# M251 Fan Blade Lighting Effect With One RGB LED

Example Code Introduction for 32-bit NuMicro® Family

## Document Information

| | |
|---|---|
| Application | This example code uses M251 to control RGB LED lighting effects on a fan blade. |
| BSP Version | M251_Series_BSP_CMSIS_V3.02.007 |
| Hardware | NuMaker-M251SD V1.1 |

*The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.*

*Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.*

*All data and specifications are subject to change without notice.*

*For additional information or questions, please contact: Nuvoton Technology Corporation.*

www.nuvoton.com

# 1. Overview

This example code uses the M251 series microcontroller (MCU) to control a single RGB LED on a fan blade to achieve various lighting effects.

The fan used is a model with built-in three-color LED lights on the blades. This fan is mainly used as a cooling unit on graphics cards. In addition to its original cooling function, it can also enhance visual appeal through various lighting effects. The visual effects for lighting on graphics cards can generally be divided into three types. The first type places lights around the outer ring of the fan slot, using light strips to create lighting effects. The second type treats the fan itself as a lampshade, achieving lighting effects by placing lights at the hub. The last type, which is introduced in this example code, embeds an RGB LED on one of the fan blades. By coordinating the fan's rotation speed with the LED, various lighting effects can be achieved.

## 1.1 This example code utilizes the following MCU features for operation:

1. PWM: Used to directly control the color of the RGB LED lights. Since there are three fans, a total of nine output pins are required.
2. QSPI: Used to simulate PWM signals for controlling the speed of the three fans.
3. PDMA: Improves data transfer efficiency to ensure that QSPI can maintain its transmission speed.
4. Timer: Has two functions — Timer Capture is used to detect fan speed for calibrating the lighting effects position, and Timer Time-out is used to change the RGB LED lights.
5. SysTick: Used in conjunction with the Timer to change the RGB LED lights.
6. UART: Not used for fan-related control, but for data transmission between the PC and the MCU.

## 1.2 Principle of RGB LED Lighting Effects on a Fan Blade

The LED lights on the fan use three channels - red, green, and blue (the three primary colors of light) - which are ultimately combined to produce the displayed color. The lighting is controlled using PWM signals; by applying PWM signals with different duty cycles over short periods, the average voltage of each primary color LED is varied, and their combination produces the desired color output.

In addition to mixing the LED lights, the phenomenon of human visual persistence is also utilized. The human eye experiences persistence of vision with rapidly changing lights; as long as the lights are updated quickly enough, lights that do not actually appear at the same time will overlap after a full rotation of the fan, making it seem as if a continuous ring of light is always present, without any sense of discontinuity. This means that for the same lighting effect, the position where the light appears in adjacent rotations must be consistent to achieve the intended effect; otherwise, the lighting effects will overlap and fail to produce the desired result. Based on this, the lighting output must be synchronized with the fan speed to ensure the light appears at the correct position.

To determine the fan's position, it is necessary to receive and analyze the fan's TACH signal. When the fan rotates, it returns a TACH signal, which is connected to the TM_EXT pin through a pull-up resistor (10 kΩ). The TACH signal changes state once every 90 degrees of rotation, so by detecting these transitions, it is possible to determine the timing of the fan motor at the 0°, 90°, 180°, and 270° positions, as well as calculate the current fan speed.

The fan speed can be controlled by varying the PWM duty cycle. However, even with the same PWM signal, the fan speed is not perfectly constant, so the current speed must be updated every rotation. This allows the lighting to display the desired color at the specified position. If the fan speed is not continuously updated, the lighting will not appear at the correct position, resulting in chaotic lighting effects.

Some devices that require cooling may have three fans operating simultaneously. To control the lighting effects of three fans with different speeds at the same time, it is necessary to conserve the use of timers. By enabling both Capture mode and Continuous Counting mode on a single timer, one timer can be used to control the lighting for one fan. Timer Capture mode uses the TM_EXT (Timer external) pin to analyze the TACH signal, detect the signal transitions, and record the time between two adjacent transitions. By measuring the time for the fan to rotate 90 degrees (one TACH signal transition), the fan speed can be calculated. In addition to using Timer Capture mode, the timer also uses Continuous Counting mode to control the lighting. To use the timer for lighting control, it is important to understand how to coordinate with Capture mode for lighting control. Figure 1-1 explains the underlying operation of the lighting effects.

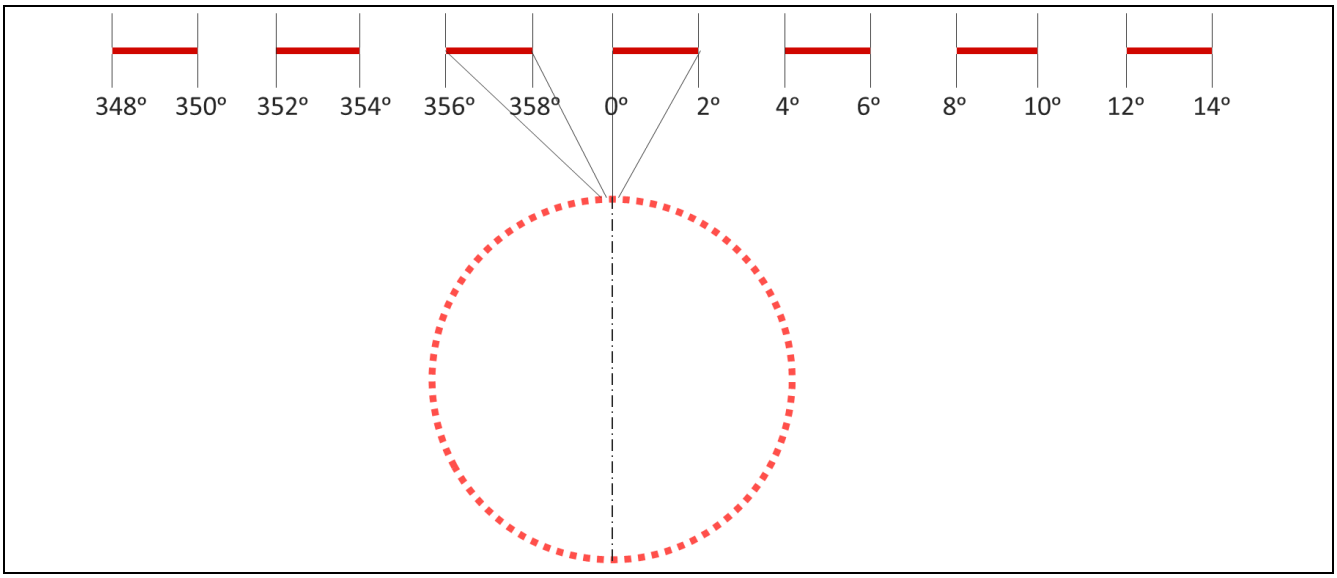



Figure 1-1 Principle of Lighting Effect on the Fan

A lighting effect is created for every 2-degree unit of rotation, as shown in Figure 1-1, which uses alternating red and white to illustrate this. In other words, the lighting is updated every 2 degrees of fan rotation. The speed calculated using Capture mode can be further used to determine the time required for a 2-degree rotation, and this value can be set as the next time-out value for Continuous Counting mode. By using Timer Continuous Counting mode, the timer interrupt is triggered every 2 degrees of fan rotation to update the lighting effect. (Since the same timer needs to work with Timer Capture mode, Periodic mode cannot be used to control

the lighting effect. Using Periodic mode would clear the TIMER_CNT (Timer counter) register, resulting in incorrect values for Capture mode.) The TIMER_CMP (Timer compare) register is updated (with the next time-out value for Continuous Counting mode) to keep the timer running. After the fan completes a full rotation, the lighting effect at the 12 o'clock position is controlled by Timer Capture mode rather than by the time-out of Timer Continuous mode. At this point, a new round of lighting display begins, the fan speed is updated, the TIF (Timer continuous counting mode time-out flag) is cleared, and the TIMER_CNT value is updated. This ensures that the lighting effect at the 12 o'clock position remains consistent between consecutive rotations and does not become mixed due to positional differences.
Once the principle of lighting generation is understood, the desired lighting effects can be implemented as needed. The following are three sample lighting effects:

- Rainbow: The entire circumference displays rainbow colors, rotating in one direction.
- Color Shift: Over time, the entire circle gradually transitions from color A to color B.
- Claws: The circle is divided into three equal sections (each 120 degrees). In each section, only the first 60 degrees are illuminated, and after a period of time, the lighting effect rotates in one direction.

The same program will produce different lighting effects depending on the fan's rotation direction. For clarity, all lighting effect explanations and illustrations in this document use clockwise fan rotation as the example.

## 1.3 Principle of Lighting Effects – Rainbow

This lighting effect displays a full circle of rainbow colors that continuously rotates in one direction. The rotation is ongoing, with an average rotation of 90 degrees every 500 ms, as shown in Figure 1-2.

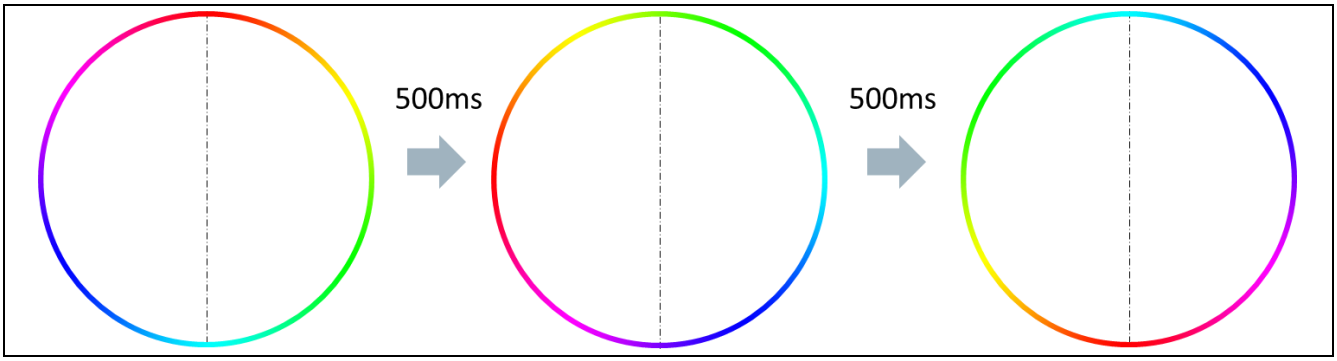



Figure 1-2 Illustration of Rainbow Lighting Effect

To avoid reducing execution efficiency, the array required for this lighting effect is created before entering the lighting control process. Since the display principle updates the lighting effect in 2-degree units, a full circle is divided into 180 segments. Therefore, the aforementioned array contains 180 elements representing PWM duty cycles, covering all the PWM duty values needed to achieve the Rainbow lighting effect (as shown in the section: Code Description). The PWM signals for the three-color LEDs are output in a continuous cycle around the circumference, forming a complete circular Rainbow lighting effect.

During Timer continuous counting mode, each time an interrupt occurs, the next element in the array is used as the PWM duty output. To ensure the Rainbow lighting effect continuously rotates rather than remaining fixed at the same position on the circle each time, SysTick is used. At the completion of each rotation, it checks whether the SysTick timer has reached the set time. If SysTick has reached the specified interval, the starting index for output is incremented by one, causing the Rainbow lighting effect to rotate in one direction.

## 1.4 Principle of Lighting Effects – Color Shift

This lighting effect displays a solid color around the entire circle, and during the color transition, every 40 ms the next color advances by 30 degrees in the direction of rotation. It takes a total of 480 ms to complete one color transition, as shown in Figure 1-3.

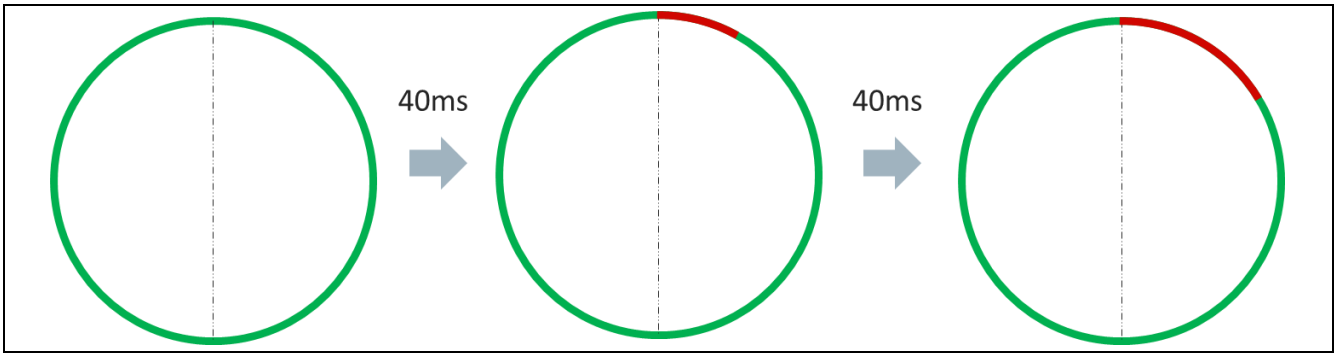



Figure 1-3 Illustration of Color Shift Lighting Effect

This figure shows a transition from green to red, with the color moving in one direction. After each color transition is completed, the same color is maintained for 1.4 seconds before the next color transition begins. Here, the SysTick clock is used to control both the timing of the color transition and the duration for displaying a single color.

## 1.5 Principle of Lighting Effects – Claws

This lighting effect displays color in three regions around the entire circle. Taking the 12 o'clock position as 0 degrees, the colored regions are 0–60 degrees, 120–180 degrees, and 240–300 degrees. Every 500 ms, the colored regions rotate 40 degrees along the circumference. After three rotations, one complete display cycle is achieved, as shown in Figure 1-4.

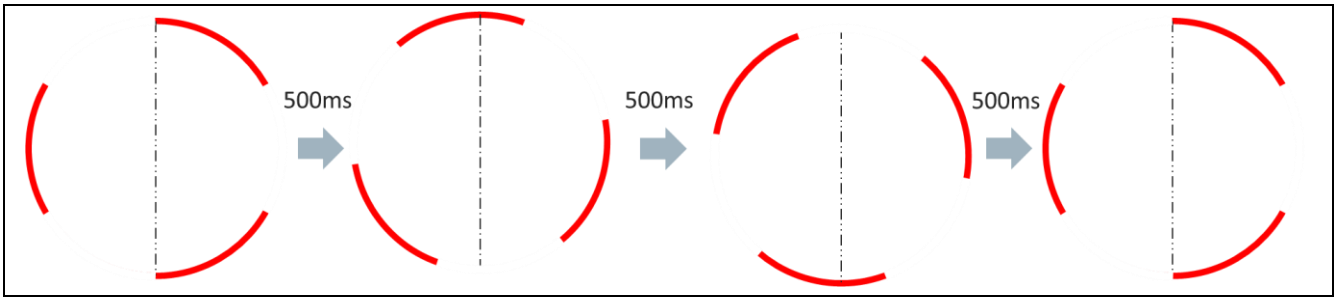



Figure 1-4 Illustration of Claws Lighting Effect (1)

On top of this, color transitions are added so that the color changes with each rotation, as shown in Figure 1-5.

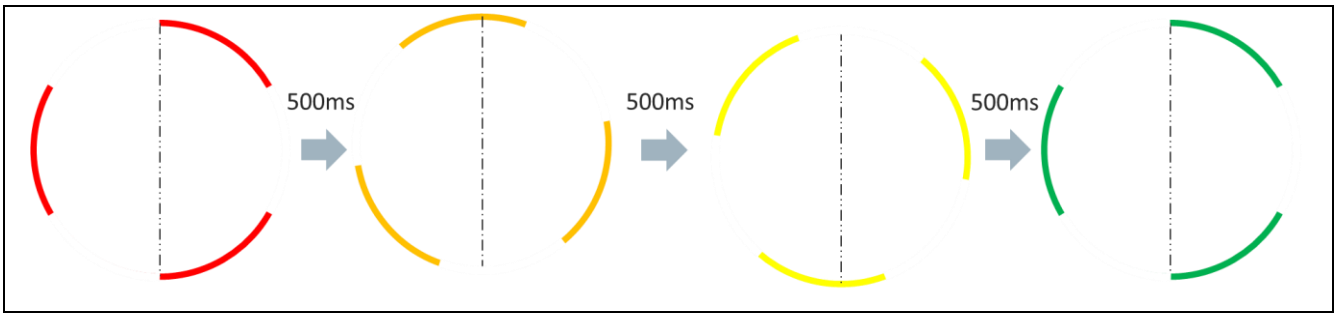



Figure 1-5 Illustration of Claws Lighting Effect (2)

## 1.6 Demo Result

The compilation and programming results of this example code are shown in Figure 1-6.

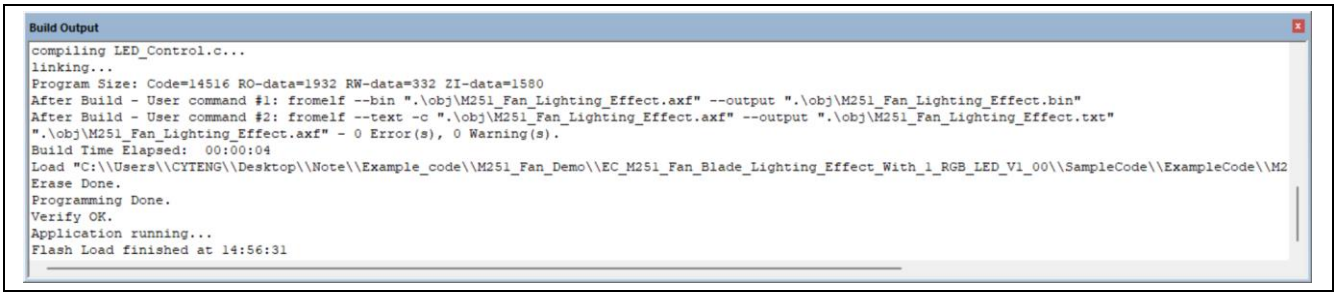



Figure 1-6 Compilation and Programming Results

The central fan rotates clockwise, while the fans on both sides rotate counterclockwise. Therefore, during operation, the lighting effects on the side fans will be opposite to those of the central fan. A photo of the fans is shown in Figure 1-7.

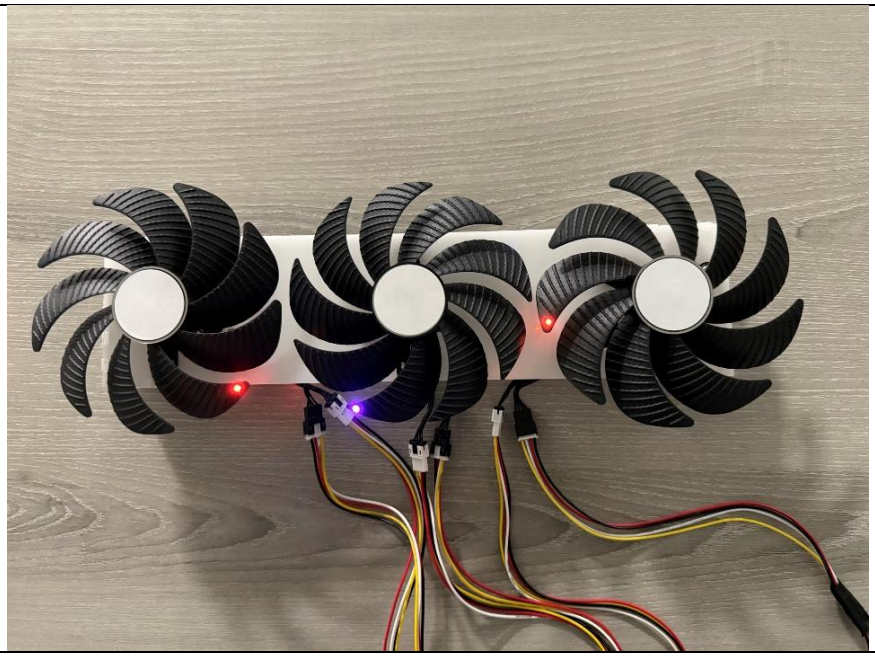


Figure 1-7 Picture of Fan

This example code displays lighting effects while the fan is rotating. However, the way a camera captures images is different from human visual perception. When taking a photo, the camera records a snapshot of the scene in a very short period of time. As a result, when photographing the fan, the captured image may show the lighting effect as incomplete around the circle, as illustrated in Figure 1-8.

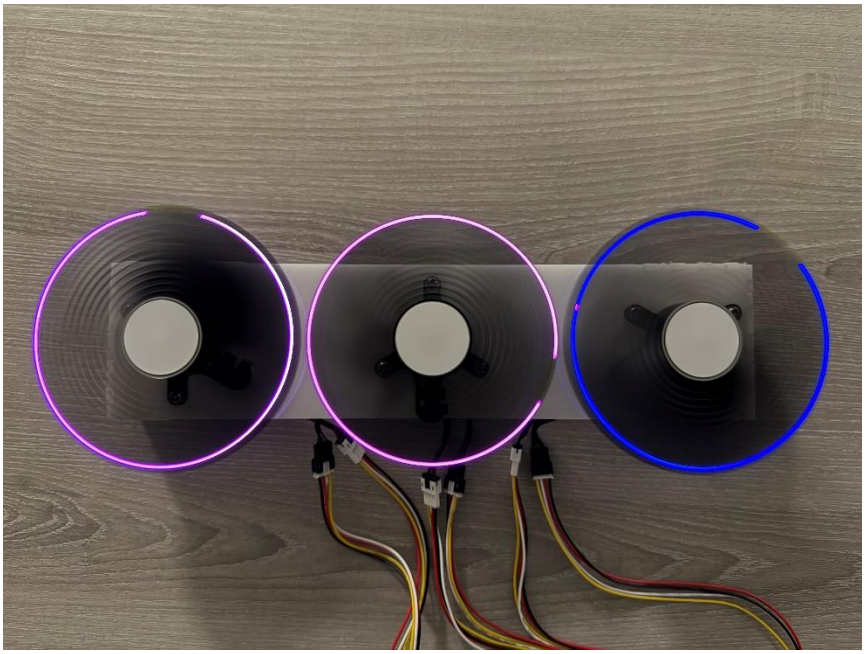


Figure 1-8 Photo Taken under Normal Conditions

At this point, increasing the exposure time can be used to improve the quality of the photographs. The results are shown in Figure 1-9 to Figure 1-13.
Although increasing the exposure time can enhance the appearance of the lighting effects in photos, the three fans do not rotate at exactly the same speed. When the slower fan completes one full rotation, the faster fan will have rotated several degrees further. As a result, during the same exposure period, some positions on certain fans may show overlapping colors from adjacent rotations. This phenomenon is especially noticeable when photographing the Rainbow lighting effect.
The Color Shift lighting effect is shown in Figure 1-9. The central fan rotates faster than the fans on either side, causing white to appear in the middle fan. Normally, to display white, all three color LEDs are output with the same PWM duty cycle. In this case, the appearance of white in the photo indicates that the colors from adjacent rotations have overlapped at that position.

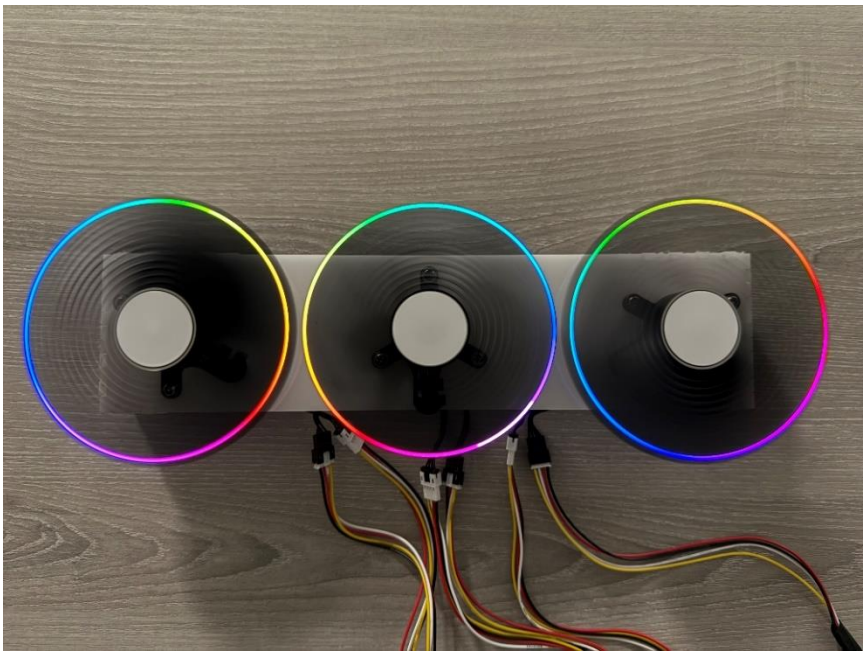


Figure 1-9 Rainbow Lighting Effect

The Color Shift lighting effect is shown in Figure 1-10 and Figure 1-11. These are before-and-after photos taken during a single lighting effect transition.

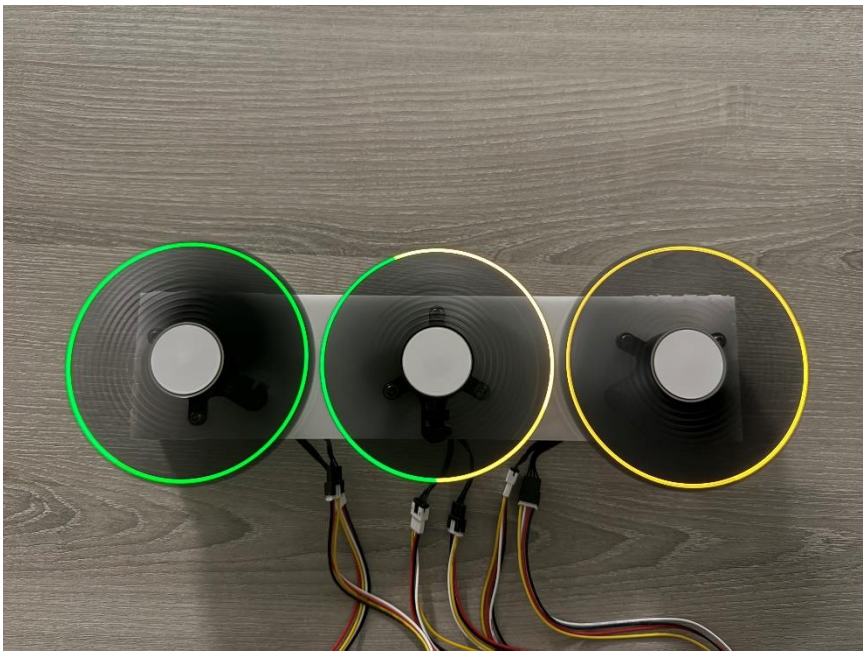


Figure 1-10 Color Shift Lighting Effect (1)

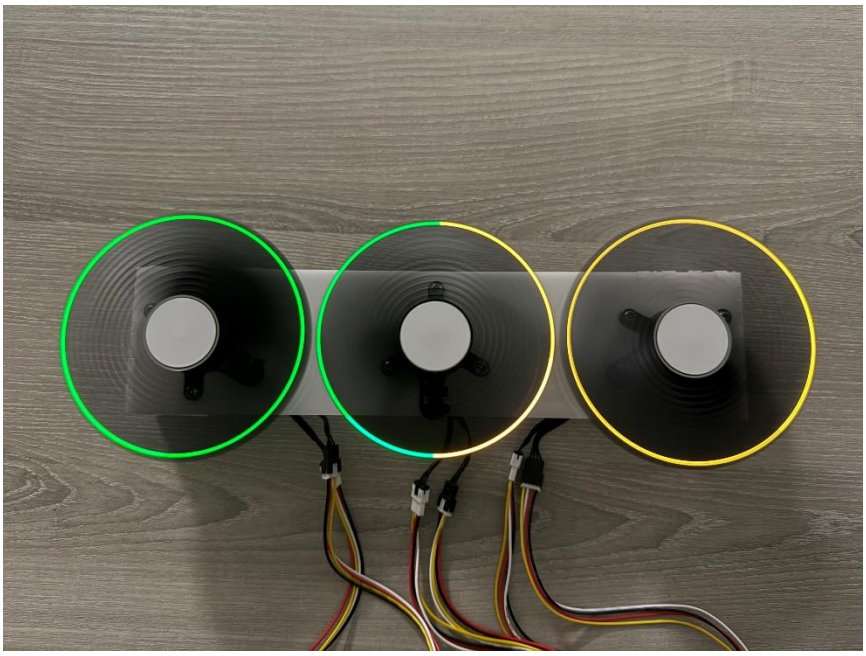


Figure 1-11 Color Shift Lighting Effect (2)

The Claws lighting effect is shown in Figure 1-12 and Figure 1-13. These are before-and-after photos taken during a single lighting effect transition.

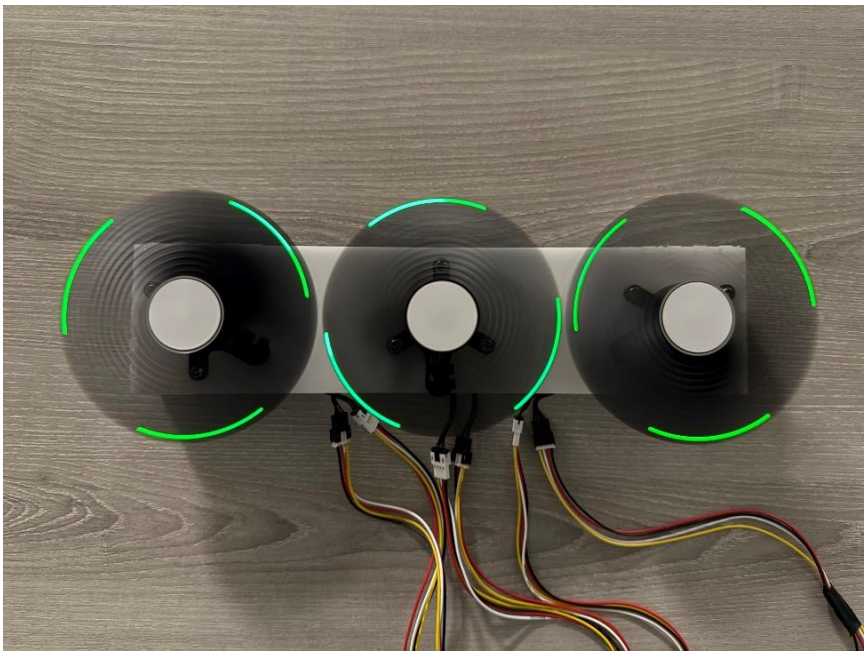


Figure 1-12 Claws Lighting Effect (1)

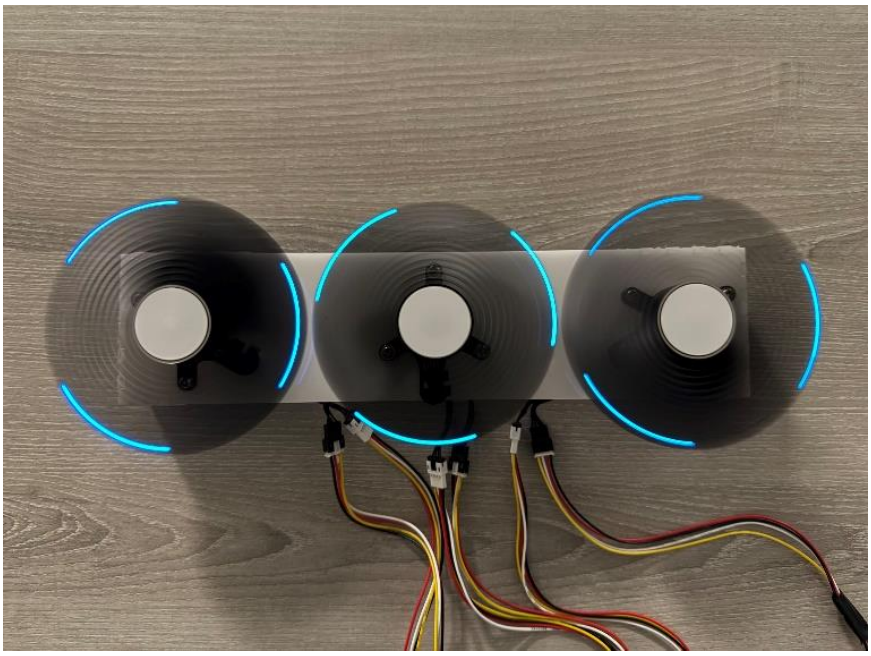


Figure 1-13 Claws Lighting Effect (2)

## 2. Code Description

After initializing the system, initialize each required peripheral. The code is located in *main.c.*

```
int32_t main(void)
{
    /* Unlock protected registers */
    SYS_UnlockReg();

    /* Init System, IP clock and multi-function I/O */
    SYS_Init();

    /* Lock protected registers */
    SYS_LockReg();

    /* Init UART */
    UART0_Init();

    /* Init Timer0, Timer1, Timer2 */
    TIMER0_Init();
    TIMER1_Init();
    TIMER2_Init();

    /* Init All Timer Capture function & enable interrupt */
    TIMER_Capture_Init();

    /* Init Systick */
    SysTick_Initial();

    /* Start SysTick */
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
```

Here is the content of the TIMER0_Init function. The initialization process for other timers is similar. The timer is set to Continuous mode, and the prescaler is set to 1 (since a value of 0 will result in a division by (0+1) during the prescaling operation).

```
void TIMER0_Init()
{
    /* Set TIMER0 continuous time-out frequency is 1Hz */
    TIMER_Open(TIMER0, TIMER_CONTINUOUS_MODE, 1);

    /* Set TIMER0 prescale = 1 */
    TIMER_SET_PRESCALE_VALUE(TIMER0, 0);
}
```

Here is the content of the TIMER_Capture_Init function. While controlling the lighting, the timer also performs capture operations for the fan motor's TACH signal. This function configures the timer for capture and enables interrupts. Since the timer needs to perform capture operations simultaneously, it must use Continuous mode so that the timer does not reset the counter value after a time-out occurs.

```
void TIMER_Capture_Init()
{
    //Enable Timer capture, capture interrupt, interrupt, NVIC
    TIMER_EnableCapture(TIMER0, TIMER_CAPTURE_FREE_COUNTING_MODE,
TIMER_CAPTURE_EVENT_RISING_FALLING);
    TIMER_EnableCaptureInt(TIMER0);
    TIMER_EnableInt(TIMER0);
```

```
    TIMER_EnableCapture(TIMER1, TIMER_CAPTURE_FREE_COUNTING_MODE,
TIMER_CAPTURE_EVENT_RISING_FALLING);
    TIMER_EnableCaptureInt(TIMER1);
    TIMER_EnableInt(TIMER1);
    TIMER_EnableCapture(TIMER2, TIMER_CAPTURE_FREE_COUNTING_MODE,
TIMER_CAPTURE_EVENT_RISING_FALLING);
    TIMER_EnableCaptureInt(TIMER2);
    TIMER_EnableInt(TIMER2);
    NVIC_EnableIRQ(TMR0_IRQn);
    NVIC_EnableIRQ(TMR1_IRQn);
    NVIC_EnableIRQ(TMR2_IRQn);
}
```

Here is the content of the SysTick_Initial function. Initialize SysTick by selecting the clock source, updating the target count value (this value will be set again during lighting effect initialization), clearing the counter value, and enabling the SysTick interrupt.

```
void SysTick_Initial(void)
{
    /* Set SysTick clock source to HCLK */
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk;

    /* Set load value to make SysTick period is 250ms */
    SysTick->LOAD = 48000000 / 4 ;

    /* Clear SysTick counter */
    SysTick->VAL = 0;

    /* Enable SysTick interrupt */
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;

    /* Enable SysTick NVIC */
    NVIC_EnableIRQ(SysTick_IRQn);
}
```

Continuing with the content of main function, in order to control the fan speed, the MCU needs to provide a PWM signal to regulate the fan's rotation speed. In this program, QSPI is used to emulate the PWM signal. To use QSPI for PWM signal emulation, a PWM duty cycle array for the fan speed must first be created to serve as the data source for QSPI output, and PDMA (Peripheral Direct Memory Access) is used to transfer the data. For more information on the principle and application, you can refer to another Nuvoton example code on NuForum: "[M251] Using QSPI with PDMA to Simulate PWM" In this example, the duty cycle of the PWM signal simulated by QSPI is fixed at 50%. Similar code will not be repeated here.

```
    /* Setting PDMA Structure for QSPI */
    /* DSCT0 Description*/
    DMA_DESC_QSPI[0].u32Ctl =
        ((40 - 1) << PDMA_DSCT_CTL_TXCNT_Pos) | /* Transfer count is 40 */ \
        PDMA_WIDTH_32 |   /* Transfer width is 32 bits(one word) */ \
        PDMA_SAR_INC |    /* Source increment size is 32 bits(one word) */ \
        PDMA_DAR_FIX |    /* Destination increment size is 32 bits(one word) */ \
        PDMA_REQ_SINGLE |  /* Transfer type is burst transfer type */ \
        PDMA_BURST_128 |  /* Burst size is 128. No effect in single transfer type */ \
        PDMA_TBINTDIS_DISABLE |   /* Disable transfer done and table empty interrupt */ \
        PDMA_OP_SCATTER;  /* Operation mode is scatter-gather mode */

    /* Configure source address */
```

```
    DMA_DESC_QSPI[0].u32Src = (uint32_t)Duty_array;

    /* Configure destination address */
    DMA_DESC_QSPI[0].u32Dest = (uint32_t)&QSPI0->TX;

    /* Configure next descriptor table address */
    DMA_DESC_QSPI[0].u32Offset = (uint32_t)&DMA_DESC_QSPI[1] - (PDMA->SCATBA); /* next
descriptor table is table 2 */
    /* next descriptor table is table 2 */

    /* DSCT1 Description*/
    DMA_DESC_QSPI[1].u32Ctl =
        ((40 - 1) << PDMA_DSCT_CTL_TXCNT_Pos) | /* Transfer count is 40 */ \
        PDMA_WIDTH_32 |   /* Transfer width is 32 bits(one word) */ \
        PDMA_SAR_INC |    /* Source increment size is 32 bits(one word) */ \
        PDMA_DAR_FIX |    /* Destination increment size is 32 bits(one word) */ \
        PDMA_REQ_SINGLE |  /* Transfer type is burst transfer type */ \
        PDMA_BURST_128 |  /* Burst size is 128. No effect in single transfer type */ \
        PDMA_TBINTDIS_DISABLE |   /* Disable transfer done and table empty interrupt */ \
        PDMA_OP_SCATTER;  /* Operation mode is scatter-gather mode */

    /* Configure source address */
    DMA_DESC_QSPI[1].u32Src = (uint32_t)Duty_array;

    /* Configure destination address */
    DMA_DESC_QSPI[1].u32Dest = (uint32_t)&QSPI0->TX;

    /* Configure next descriptor table address */
    DMA_DESC_QSPI[1].u32Offset = (uint32_t)&DMA_DESC_QSPI[0] - (PDMA->SCATBA); /* next
descriptor table is table 1 */
    /* next descriptor table is table 1 */

    /* Calculate PWM output duty array */
    Calculate_Duty_array(Duty_right_now);

    /* Init QSPI & PDMA, Start QSPI output */
    QSPI_and_PDMA_Init();
```

Continuing with the content of main function, initialize the PWM for controlling the LED and create the array needed for the Rainbow lighting effect, which contains all the PWM duty values required for a complete rainbow cycle. Then, start the Timer and PWM, assign the lighting effects for the three fans, and perform the lighting effect initialization.

```
/* Init PWM output */
    PWM_Init();

    /* Create Rainbow lighting effect array */
    for (uint8_t i = 0; i < 180; i++)
    {
        if (i < 40)
        {
            Rainbow_array[i] = 100;
        }
        else if ((i > 39) && (i < 80))
        {
            Rainbow_array[i] = 100 - 2.5 * (i - 40);
        }
        else if ((i > 79) && (i < 140))
        {
            Rainbow_array[i] = 0;
        }
        else if ((i > 139) && (i < 180))
```

```
        {
            Rainbow_array[i] = 2.5 * (i - 140);
        }
    }

    //Timer start
    TIMER_Start(TIMER0);
    TIMER_Start(TIMER1);
    TIMER_Start(TIMER2);

    /* Start PWM counter */
    PWM_Start(fan_left.PWM_generator, fan_left.PWM_RMSK);
    PWM_Start(fan_left.PWM_generator, fan_left.PWM_GMSK);
    PWM_Start(fan_left.PWM_generator, fan_left.PWM_BMSK);
    PWM_Start(fan_middle.PWM_generator, fan_middle.PWM_RMSK);
    PWM_Start(fan_middle.PWM_generator, fan_middle.PWM_GMSK);
    PWM_Start(fan_middle.PWM_generator, fan_middle.PWM_BMSK);
    PWM_Start(fan_right.PWM_generator, fan_right.PWM_RMSK);
    PWM_Start(fan_right.PWM_generator, fan_right.PWM_GMSK);
    PWM_Start(fan_right.PWM_generator, fan_right.PWM_BMSK);

    /* Init lighting effect: Rainbow lighting mode */
    Lighting_Mode_Setting = RAINBOW_LIGHTING_MODE;
    fan_left.light_mode_choose = Lighting_Mode_Setting ;
    fan_middle.light_mode_choose = Lighting_Mode_Setting ;
    fan_right.light_mode_choose = Lighting_Mode_Setting ;
    Rainbow_initial(&fan_left);
    Rainbow_initial(&fan_middle);
    Rainbow_initial(&fan_right);
```

The Rainbow_array created as described above is shown in the table below. When applying this array for the Rainbow lighting effect, the duty value for red is Rainbow_array[i], for green is Rainbow_array[i+60], and for blue is Rainbow_array[i+120]. If the array index exceeds 179, the value is taken modulo 180. In this way, as shown in Table 2-1, after going through all 180 elements, a complete cycle of rainbow colors is achieved.

| i | 0 | 20 | 40 | 60 | 80 | 100 | 120 | 140 | 160 |
|---|---|---|---|---|---|---|---|---|---|
| Rainbow_array[i] (R) | 100 | 100 | 100 | 50 | 0 | 0 | 0 | 0 | 50 |
| G | 0 | 0 | 50 | 100 | 100 | 100 | 50 | 0 | 0 |
| B | 50 | 0 | 0 | 0 | 0 | 50 | 100 | 100 | 100 |

Table 2-1 Rainbow_array Reference Table

Here is the content of the PWM_Init function. Set the initial frequency and duty cycle for the PWM, and configure the output signal as inverted to simplify subsequent calculations. Please note that it is recommended for the PWM frequency to be higher than the frequency corresponding to the time it takes the fan to rotate by 1 degree. The reason is that if the PWM frequency is not high enough, the PWM-controlled lighting will not complete a full cycle during the fan's rotation, resulting in visible banding in the lighting display and failing to achieve the desired lighting effect.

For example, if you want to display purple light (R,G,B) = (128,0,255) at the 0–2 degree position, and the PWM output frequency's period matches the time it takes the fan motor to rotate 2 degrees, then starting a new PWM cycle at the 0-degree position will cause the light to display bright purple (R,G,B) = (255,0,255) from 0 to 1 degree, and blue (R,G,B) = (0,0,255) from 1 to 2 degrees. This is a result of the PWM frequency not being fast enough, as shown in Figure 2-1.

```
void PWM_Init()
{
    //For PWM Initial Setting
    PWM_ConfigOutputChannel(fan_left.PWM_generator, fan_left.PWM_Rchannel, 50000,
100);//PWM0 ch0
    PWM_ConfigOutputChannel(fan_left.PWM_generator, fan_left.PWM_Gchannel, 50000,
100);//PWM0 ch1
    PWM_ConfigOutputChannel(fan_left.PWM_generator, fan_left.PWM_Bchannel, 50000,
100);//PWM0 ch2
    PWM_ConfigOutputChannel(fan_middle.PWM_generator, fan_middle.PWM_Rchannel, 50000,
100);//PWM0 ch3
    PWM_ConfigOutputChannel(fan_middle.PWM_generator, fan_middle.PWM_Gchannel, 50000,
100);//PWM0 ch4
    PWM_ConfigOutputChannel(fan_middle.PWM_generator, fan_middle.PWM_Bchannel, 50000,
100);//PWM0 ch5
    PWM_ConfigOutputChannel(fan_right.PWM_generator, fan_right.PWM_Rchannel, 50000,
100);//PWM1 ch3
    PWM_ConfigOutputChannel(fan_right.PWM_generator, fan_right.PWM_Gchannel, 50000,
100);//PWM1 ch4
    PWM_ConfigOutputChannel(fan_right.PWM_generator, fan_right.PWM_Bchannel, 50000,
100);//PWM1 ch5

    /* Enable output of PWM0, PWM1 */
    PWM_EnableOutput(fan_left.PWM_generator, fan_left.PWM_RMSK);
    PWM_EnableOutput(fan_left.PWM_generator, fan_left.PWM_GMSK);
    PWM_EnableOutput(fan_left.PWM_generator, fan_left.PWM_BMSK);
    PWM_EnableOutput(fan_middle.PWM_generator, fan_middle.PWM_RMSK);
    PWM_EnableOutput(fan_middle.PWM_generator, fan_middle.PWM_GMSK);
    PWM_EnableOutput(fan_middle.PWM_generator, fan_middle.PWM_BMSK);
    PWM_EnableOutput(fan_right.PWM_generator, fan_right.PWM_RMSK);
    PWM_EnableOutput(fan_right.PWM_generator, fan_right.PWM_GMSK);
    PWM_EnableOutput(fan_right.PWM_generator, fan_right.PWM_BMSK);

    //PWM Inverse
    PWM0->POLCTL |= 0x3FUL; //ch0~ch5
    PWM1->POLCTL |= 0x38UL; //ch3~ch5
}
```

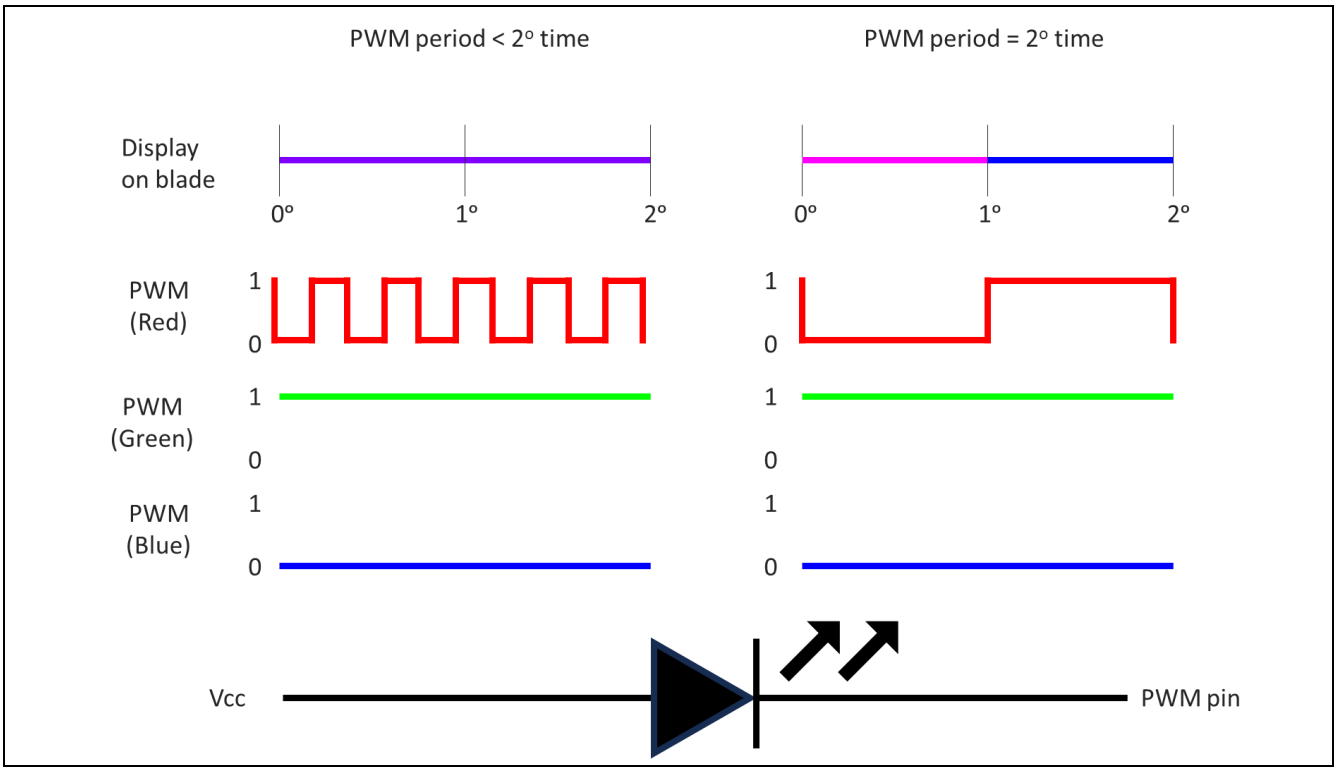



Figure 2-1 Explanation of the Impact of PWM Frequency on Lighting Effects

Other lighting effect initialization functions, such as the Rainbow_initial function, are used to initialize certain variables when switching between different lighting effects. This ensures that variables from the previous effect do not affect the next one. At the beginning of the function, PWM_EnableOutput is called because the Claws lighting effect may have turned off the current PWM output, so it needs to be explicitly enabled during initialization. Finally, the SysTick target value is updated and SysTick is reset to restart the timing.

```
void Rainbow_initial(struct fan* fan_setting)
{
    PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_RMSK);
    PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_GMSK);
    PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_BMSK);

    /* Init variable*/
    fan_setting->update_count = fan_setting->Capture_Init_count_Offset;
    fan_setting->fan_count = 0;
    fan_setting->count_initial = 0;
    fan_setting->R_duty = 0;
    fan_setting->G_duty = 0;
    fan_setting->B_duty = 0;
    fan_setting->systick_flag = 0;

    /* Setting SysTick time-out value */
    SysTick->LOAD = RAINBOW_SYSTICK_LOAD_VALUE * fan_setting->Speed;

    /* Reset SysTick */
    SysTick->VAL = 0;
}
```

Here is the content of the Color_shift_initial function. Compared to the Rainbow_initial function, this function additionally fills in the color values for Color_R_Duty and the other two color channels. This is because, unlike the Rainbow lighting effect which uses the Rainbow_array, this lighting effect uses the Default_Color array to calculate the transitional colors for the lighting effect. It also converts the RGB values from 0–255 to PWM duty values from 0–100 (the brightness can be adjusted by modifying the structure according to the user's preference, but normally there is no need to change it). The Color_Transition variable indicates whether the fan is currently displaying the initial color or is in the process of transitioning between colors. By setting different initial values for Color_Transition on the three fans, the lighting effects can be performed in sequence.

```
void Color_shift_initial(struct fan* fan_setting)
{
    PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_RMSK);
    PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_GMSK);
    PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_BMSK);

    /* Init Basic Color */
    for (uint8_t i = 0; i < fan_setting->Color_Num; i++)
    {
        fan_setting->Color_R_Duty[i] =  Default_Color[i][0] * DUTY_MAX / COLOR_VALUE_MAX *
fan_setting->Brightness / BRIGHTNESS_MAX;
        fan_setting->Color_G_Duty[i] =  Default_Color[i][1] * DUTY_MAX / COLOR_VALUE_MAX *
fan_setting->Brightness / BRIGHTNESS_MAX;
        fan_setting->Color_B_Duty[i] =  Default_Color[i][2] * DUTY_MAX / COLOR_VALUE_MAX *
fan_setting->Brightness / BRIGHTNESS_MAX;
    }

    /* Init variable*/
    fan_left.Color_Transition = 2;
    fan_middle.Color_Transition = 1;
    fan_right.Color_Transition = 0;
    fan_setting->update_count = fan_setting->Capture_Init_count_Offset;
    fan_setting->fan_count = 0;
    fan_setting->count_initial = 0;
    fan_setting->color_select = 0;
    fan_setting->change_count = 0;
    fan_setting->systick_flag = 0;

    /* Setting SysTick time-out value */
    SysTick->LOAD = COLOR_SHIFT_SYSTICK_LOAD_VALUE * fan_setting->Speed;

    /* Reset SysTick */
    SysTick->VAL = 0;
}
```

Before introducing each lighting effect individually, it is necessary to first explain the SysTick interrupt and Timer interrupt, as both are closely related to lighting control. The former is used to help time the transition to the next frame of the lighting effect, while the latter is responsible for capturing the TACH signal from the fan motor and outputting the PWM signal to control the LED at each position on the fan.

In the SysTick interrupt handler, every time the interrupt occurs twice, the flag for switching the lighting effect is set to 1. These flags are cleared during the execution of the lighting effect.

Since each fan operates at a different speed, the flags should not be shared among the three fans. For this reason, it is not recommended to directly check the SysTick time-out flag within the lighting effect function, as clearing the flag there may cause some fans to miss the lighting effect transition.

```
void SysTick_Handler(void)
{
    /* Reset SysTick */
    SysTick->VAL = 0;

    /* Set flag = 1 when enter SysTick interrupt twice */
    if (SysTick_count > 1)
    {
        fan_left.systick_flag = 1;
        fan_middle.systick_flag = 1;
        fan_right.systick_flag = 1;
        SysTick_count = 0;
    }

    /* Calculate SysTick count */
    SysTick_count++;
}
```

Here is the content of the Timer interrupt function. TMR0_IRQHandler is used as an example; the interrupt routines for Timer1 and Timer2 are similar. First, the Capture flag is checked. This flag is used to detect the TACH signal returned by the fan motor during operation. Capturing this signal serves two purposes: first, it allows you to determine the current speed of the fan, which is used as a timing reference for updating the lighting effects. Second, it enables you to determine the fan's current "relative position" in its rotation. Since the starting position of the fan motor can vary each time it is powered on, the position where the TACH signal begins is also variable, making it impossible to use the TACH signal to indicate the absolute position of the fan motor. By knowing the relative position of the fan and recording the number of times the Capture interrupt occurs, you can determine whether the fan has completed a full rotation.

After a full rotation, the fan speed is updated to provide a reference for the next Timer lighting effect update and to establish a starting reference point for lighting effects that cycle once per rotation. This not only makes it easier to calculate lighting effect parameters, but also ensures that the lighting effects between adjacent rotations remain consistent, preserving the intended visual result. Finally, any pending Time-out interrupt flags are cleared.

```
void TMR0_IRQHandler(void)
{
    /* Check capture flag=1, otherwise is time out interrupt occur */
    if (TIMER_GetCaptureIntFlag(TIMER0) == 1)
    {
        /* Clear capture interrupt flag */
        TIMER_ClearCaptureIntFlag(TIMER0);

        /* Calculate capture count, if perform four times capture, it means the fan motor
has completed one revolution.  */
        fan_left.Capture_Init_count++;

        /* Record the capture position when the motor starts rotating after being powered
on.*/
```

```
        fan_left.Capture_Init_count = fan_left.Capture_Init_count % (4 +
fan_left.Capture_Init_count_Start_position) + fan_left.Capture_Init_count_Start_position;

        /* If the fan motor has completed one revolution. */
        if ((fan_left.update_count >  3) && (fan_left.Capture_Init_count ==
+fan_left.Capture_Init_count_Offset))
        {
            /* Update fan speed */
            Update_Timer_Capture(&fan_left);

            /* Initial fan_count & determine PWM output  */
            Timer_capture_function(&fan_left);

            /* Init update count(capture count) */
            fan_left.update_count = 0;

            /* Setting next Timer time-out value */
            TIMER0->CMP = TIMER0->CNT + fan_left.CalCMP[3];

            /* Clear time out interrupt flag(TIF) to prevent re-entering the interrupt
immediately after exiting the interrupt  */
            TIMER_ClearIntFlag(TIMER0);
            NVIC_ClearPendingIRQ(TMR0_IRQn);

        }

        /* Calculate update count(capture count) */
        fan_left.update_count++;
    }
```

Here is the content of the Update_Timer_Capture function. The actual count value is obtained by calculating the difference between two consecutive Timer capture values for a full rotation. This value represents the time required for a 360-degree rotation. Since the lighting effect is updated every 2 degrees, this value is divided by 180 and stored for use in calculating the next time-out interval in Continuous mode.

```
void Update_Timer_Capture(struct fan* fan_setting)
{
    //Put t-1 count value in CalCMP[1],  Scan Capture and put data in CalCMP[0],
Difference value Calculate in CalCMP[2]
    fan_setting->CalCMP[1] = fan_setting->CalCMP[0];
    fan_setting->CalCMP[0] = TIMER_GetCaptureData(fan_setting->Timer_Generator);
    fan_setting->CalCMP[2] = fan_setting->CalCMP[0] - fan_setting->CalCMP[1];
    fan_setting->CalCMP[2] = fan_setting->CalCMP[2] & 0xffffff;
    fan_setting->CalCMP[3]=(fan_setting->CalCMP[2]/180)& 0xffffff;
}
```

Continuing with the content of Timer interrupt function, when the Timer interrupt occurs without the Capture flag being set, it indicates that the fan has rotated 2 degrees and the lighting effect needs to be updated. At this point, the Timer_Light_active_function is called to change the LED display. After updating the LED, since the Timer is in Continuous mode, the TIMER_CMP value must be reset so that the Timer knows when to trigger the next interrupt (i.e. after the fan rotates another 2 degrees). Figure 2-2 illustrates the types of Timer interrupts triggered around the entire circumference.

```
    /* Time out interrupt */
    else if ((TIMER_GetCaptureIntFlag(TIMER0) == 0))
    {
        /* Clear time out interrupt flag */
        TIMER_ClearIntFlag(TIMER0);

        /* Active lighting effect */
        Timer_Light_active_function(&fan_left);

        /* Setting next Timer time-out value */
        TIMER0->CMP = TIMER0->CMP + fan_left.CalCMP[3];
    }
}
```

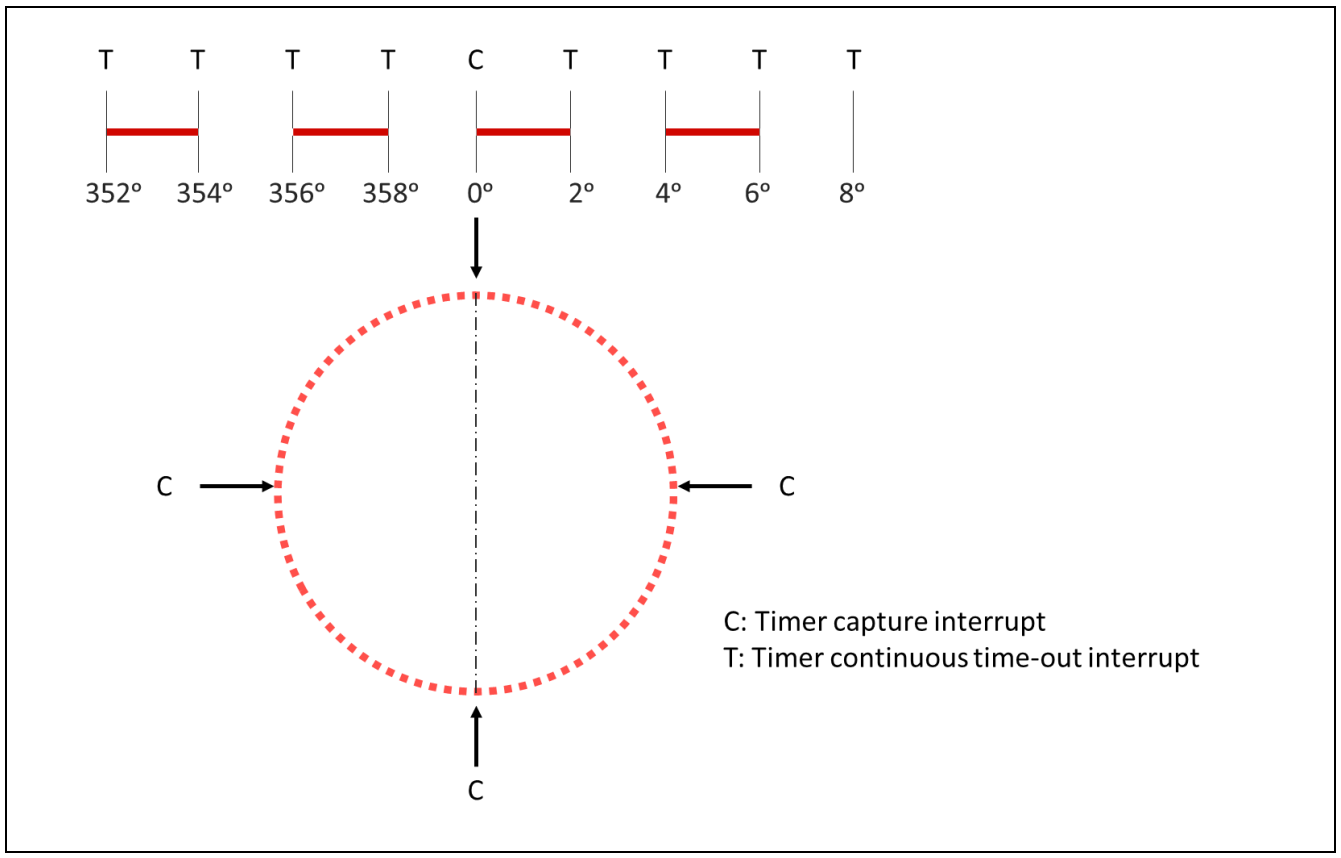



Figure 2-2 Explanation of Circumference and Timer Interrupt

Finally, the lighting effect functions located in LED_Control.c are used in the Timer interrupt routines described above. These functions are divided into two parts: one updates the lighting effect after the fan completes a full rotation, and the other updates the lighting effect after the fan rotates 2 degrees.

After a full rotation of the fan, the Timer_capture_function is used to reset the main variable, fan_count, which continuously tracks the fan's lighting effect. This reset provides a reference point for each rotation, preventing significant fluctuations in the lighting effect that could occur if the fan_count values differ between adjacent rotations (which would happen if the reference point were not reset).

In the Color Shift lighting effect, in addition to resetting the reference point, the first lighting effect display after the reset is also performed. As described in the Color_shift_initial function, the Color_Transition variable indicates the state of the color transition: when its value is 3, it means the fan is in the process of transitioning colors, so the updated color should be displayed at the reset point; when its value is less than 3, it means the fan is still showing the previous color, so the previous color should be displayed at the reset point.

```
void Timer_capture_function(struct fan* fan_setting)
{
    switch (fan_setting->light_mode_choose)
    {
        case RAINBOW_LIGHTING_MODE:
        {
            /* Reset fan_count at capture point */
            fan_setting->fan_count = fan_setting->count_initial;
        }
        break;

        case COLOR_SHIFT_LIGHTING_MODE:
        {
            /* Determine the value of the capture point based on the value of
Color_Transition */
            if (fan_setting->Color_Transition == 3)
            {
                fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->Color_R_Duty[(fan_setting->color_select + 1) %
fan_setting->Color_Num], 100);
                fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->Color_G_Duty[(fan_setting->color_select + 1) %
fan_setting->Color_Num], 100);
                fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->Color_B_Duty[(fan_setting->color_select + 1) %
fan_setting->Color_Num], 100);
            }
            else if (fan_setting->Color_Transition < 3)
            {
                fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->Color_R_Duty[fan_setting->color_select ], 100);
                fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->Color_G_Duty[fan_setting->color_select ], 100);
                fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->Color_B_Duty[fan_setting->color_select ], 100);
            }

            /* Reset fan_count at capture point */
            fan_setting->count_initial = 0;
            fan_setting->fan_count = fan_setting->count_initial;
        }
        break;

        case CLAWS_LIGHTING_MODE:
        {
            /* Reset fan_count at capture point */
            fan_setting->fan_count = fan_setting->count_initial;
        }
        break;
```

```
    }

    /* Setting comparator value, to change PWM duty */
    PWM_SET_CMR(fan_setting->PWM_generator, fan_setting->PWM_Rchannel, fan_setting-
>u32NewCMR_R);
    PWM_SET_CMR(fan_setting->PWM_generator, fan_setting->PWM_Gchannel, fan_setting-
>u32NewCMR_G);
    PWM_SET_CMR(fan_setting->PWM_generator, fan_setting->PWM_Bchannel, fan_setting-
>u32NewCMR_B);
}
```

Here is the content of the Timer_Light_active_function function. This function is used in most cases when entering a Timer interrupt. It determines the current lighting effect and performs the corresponding calculations, which will be explained one by one below.

Rainbow lighting effect: After a SysTick time-out occurs, the value of count_initial is updated so that, when entering Timer_capture_function, fan_count is set to this value. Then, based on the value of fan_count (where the values for red, green, and blue are different), the corresponding element from the Rainbow_array is selected as the PWM duty cycle. This allows the correct color to be displayed at each position, and when connected around the entire circumference, a full Rainbow lighting effect is achieved, as shown in Figure 2-3.

```
void Timer_Light_active_function(struct fan* fan_setting)
{
    switch (fan_setting->light_mode_choose)
    {
        case RAINBOW_LIGHTING_MODE:
         {
            /* When systick_flag =1, change next lighting effect frame */
            if (fan_setting->systick_flag == 1) // flag 0->1 : 11ms
            {
                fan_setting->systick_flag = 0;
                fan_setting->count_initial++;
                fan_setting->count_initial = fan_setting->count_initial % (360 /
COUNT_ANGLE_UNIT);
            }

            /* Change PWM output each time when enter interrupt */
            fan_setting->R_duty = Rainbow_array[fan_setting->fan_count];
            fan_setting->G_duty = Rainbow_array[fan_setting->fan_count_G];
            fan_setting->B_duty = Rainbow_array[fan_setting->fan_count_B];
            fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->R_duty, 100);
            fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->G_duty, 100);
            fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->B_duty, 100);

            fan_setting->fan_count++;
            fan_setting->fan_count = fan_setting->fan_count % (COUNT_ANGLE_360);

            /* Update fan_count_G & fan_count_B value */
            fan_setting->fan_count_G = fan_setting->fan_count + 120;
            fan_setting->fan_count_G = fan_setting->fan_count_G % (COUNT_ANGLE_360);
            fan_setting->fan_count_B = fan_setting->fan_count + 60;
            fan_setting->fan_count_B = fan_setting->fan_count_B % (COUNT_ANGLE_360);
```

```
        }
        break;
    }
```

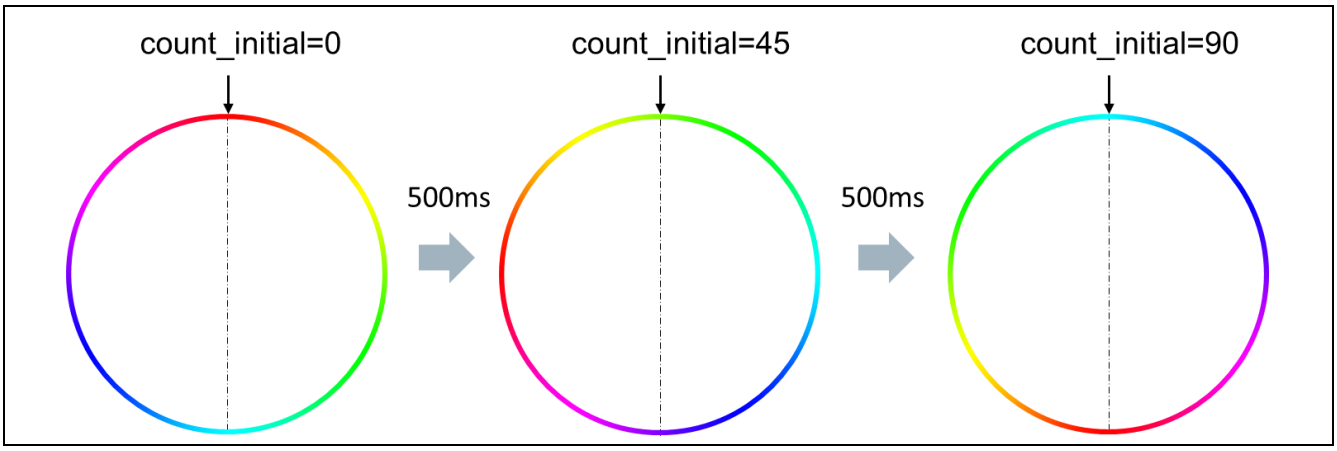



Figure 2-3 Explanation of Rainbow Lighting Effect

Color flow lighting effect: After a SysTick time-out occurs, the value of change_count is updated for subsequent calculations and for use in Timer_capture_function. The change_count variable marks the point in each rotation where color 1 transitions to color 2. Since the color flow effect only advances the color transition at certain intervals, change_count is used to keep track of the current transition position around the circumference. With continuous lighting updates, this creates the visual impression of a gradual transition from color 1 to color 2. The Color_Transition variable functions as described in Timer_capture_function.

During the color transition (Color_Transition = 3), change_count is used to determine whether to output color 1 or color 2. At other times (Color_Transition < 3), color 1 is output.

During the color transition (Color_Transition = 3), as shown in Figure 2-4, change_count increases by 15 (COUNT_ANGLE_30) each time, representing a 30-degree increment with each SysTick time-out. In this way, a full rotation completes one lighting effect cycle.

```
        case COLOR_SHIFT_LIGHTING_MODE:
        {
            /* When systick_flag =1, change next lighting effect frame */
            if (fan_setting->systick_flag == 1)
            {
                fan_setting->systick_flag = 0;
                fan_setting->change_count = fan_setting->change_count + (COUNT_ANGLE_30);

                if (fan_setting->change_count > (COUNT_ANGLE_30 * 11))
                {
                    fan_setting->change_count = 0;

                    if (fan_setting->Color_Transition < 3)
                    {
                        fan_setting->Color_Transition++;
                    }
                    else
                    {
```

```
                        fan_setting->color_select = (fan_setting->color_select + 1) %
fan_setting->Color_Num;
                        fan_setting->Color_Transition = 0;
                    }
                }
            }

            /* Set the lighting effect according to Color_Transition */
            if (fan_setting->Color_Transition == 3)
            {
                if (fan_setting->fan_count < fan_setting->change_count)
                {
                    fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->Color_R_Duty[(fan_setting->color_select + 1) %
fan_setting->Color_Num], 100);
                    fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->Color_G_Duty[(fan_setting->color_select + 1) %
fan_setting->Color_Num], 100);
                    fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->Color_B_Duty[(fan_setting->color_select + 1) %
fan_setting->Color_Num], 100);
                }
                else if (fan_setting->fan_count > (fan_setting->change_count - 1))
                {

                    fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->Color_R_Duty[fan_setting->color_select], 100);
                    fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->Color_G_Duty[fan_setting->color_select], 100);
                    fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->Color_B_Duty[fan_setting->color_select], 100);

                }
            }
            else if (fan_setting->Color_Transition < 3)
            {
                fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->Color_R_Duty[fan_setting->color_select], 100);
                fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->Color_G_Duty[fan_setting->color_select], 100);
                fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->Color_B_Duty[fan_setting->color_select], 100);
            }

            fan_setting->fan_count++;
            fan_setting->fan_count = fan_setting->fan_count % (COUNT_ANGLE_360);
        }
        break;
```

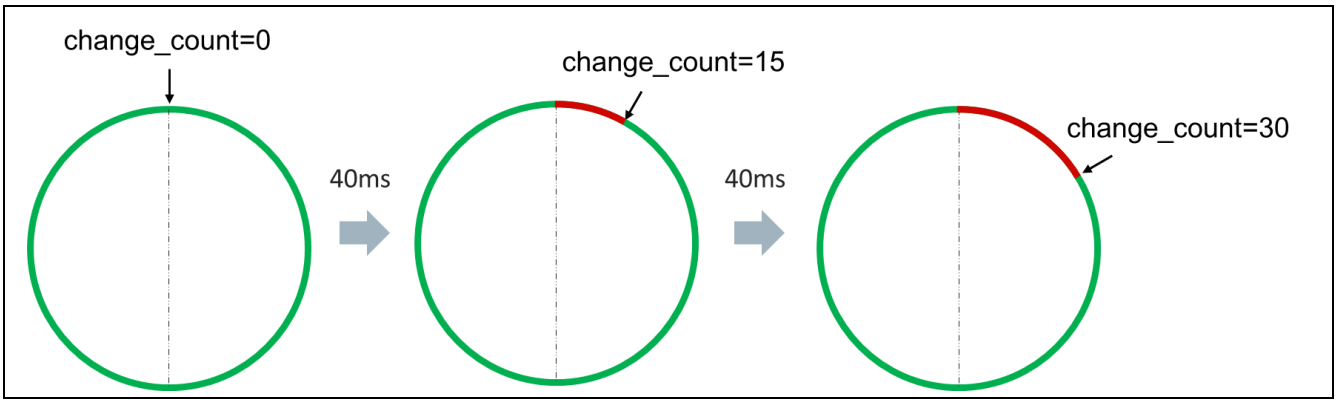



Figure 2-4 Explanation of Color Shift Lighting Effect

Claws lighting effect: After a SysTick time-out occurs, the value of count_initial is updated so that, when entering Timer_capture_function, fan_count is set to this value. Next, the LED color is changed to the next color. Here, color_select indicates which color in the array should be displayed, and Color_Num represents the total number of colors in the array. When color_select exceeds the number of colors in the array, it is reset to zero.

The Claws lighting effect is achieved by disabling PWM output in specific regions and enabling PWM output in others. For example, as shown in Figure 2-5, on the far left with count_initial = 0, the brightness changes every 60 degrees clockwise starting from the 12 o'clock position. When count_initial = 20, the entire lighting effect shifts 40 degrees counterclockwise. When count_initial = 40, the effect shifts another 40 degrees counterclockwise. After three shifts, one complete cycle of the lighting effect is achieved.

By coordinating the colors and the positions of the lighting effect, the Claws lighting effect is displayed.

```
        case CLAWS_LIGHTING_MODE:
        {
            /* When systick_flag =1, change next lighting effect frame */
            if (fan_setting->systick_flag == 1)
            {
                fan_setting->systick_flag = 0;
                fan_setting->count_initial = (fan_setting->count_initial + 20) % 60;

                fan_setting->color_select = (fan_setting->color_select + 1) %
fan_setting->Color_Num;

                fan_setting->u32NewCMR_R = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Rchannel, fan_setting->Color_R_Duty[fan_setting->color_select], 100);
                fan_setting->u32NewCMR_G = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Gchannel, fan_setting->Color_G_Duty[fan_setting->color_select], 100);
                fan_setting->u32NewCMR_B = CalNewDutyCMR(fan_setting->PWM_generator,
fan_setting->PWM_Bchannel, fan_setting->Color_B_Duty[fan_setting->color_select], 100);


                PWM_SET_CMR(fan_setting->PWM_generator, fan_setting->PWM_Rchannel,
fan_setting->u32NewCMR_R);
                PWM_SET_CMR(fan_setting->PWM_generator, fan_setting->PWM_Gchannel,
fan_setting->u32NewCMR_G);
                PWM_SET_CMR(fan_setting->PWM_generator, fan_setting->PWM_Bchannel,
fan_setting->u32NewCMR_B);
```

```
            }

            //Claws bright zone in first frame(angle): 0~60  120~180, 240~300
            if ((fan_setting->fan_count < COUNT_ANGLE_60) | (((COUNT_ANGLE_120 - 1) <
fan_setting->fan_count) & (fan_setting->fan_count < COUNT_ANGLE_180)) | \
                            (((COUNT_ANGLE_240 - 1) < fan_setting->fan_count) &
(fan_setting->fan_count < COUNT_ANGLE_300)))
            {
                PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_RMSK);
                PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_GMSK);
                PWM_EnableOutput(fan_setting->PWM_generator, fan_setting->PWM_BMSK);

            }
            //Claws dark zone in first frame(angle): 60~120, 180~240, 300~360
            else if ((((COUNT_ANGLE_60 - 1) < fan_setting->fan_count) &
(fan_setting->fan_count < COUNT_ANGLE_120)) | \
                            (((COUNT_ANGLE_180 - 1) < fan_setting->fan_count) &
(fan_setting->fan_count < COUNT_ANGLE_240)) | (((COUNT_ANGLE_300 - 1) <
fan_setting->fan_count) & (fan_setting->fan_count < COUNT_ANGLE_360)))
            {
                PWM_DisableOutput(fan_setting->PWM_generator, fan_setting->PWM_RMSK);
                PWM_DisableOutput(fan_setting->PWM_generator, fan_setting->PWM_GMSK);
                PWM_DisableOutput(fan_setting->PWM_generator, fan_setting->PWM_BMSK);
            }
            fan_setting->fan_count++;
            fan_setting->fan_count = fan_setting->fan_count % (COUNT_ANGLE_360);
        }
        break;
```

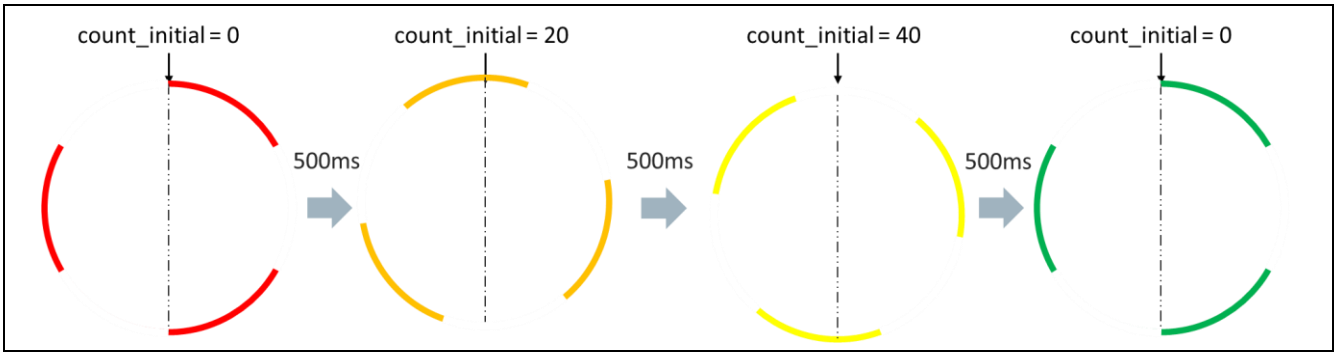



Figure 2-5 Explanation of Claws Lighting Effect

# 3. Software and Hardware Requirements

## 3.1 Software Requirements

- BSP version
  - M251_Series_BSP_CMSIS_V3.02.007
- IDE version
  - Keil uVersion 5.39

## 3.2 Hardware Requirements

- Circuit components
  - NuMaker-M251SD V1.1
- Other
  - Three fans with LED lights
- Pin Connect
  - Connect the UART0 TX (PB.13) pin to the PC UART RX and connect the UART0 RX (PB.12) pin to the PC UART TX. Also, turn on the TXD and RXD switches on the Nu-Link2-Me to show the execution results of this example code.
  - Each fan has several wires, and each fan's pins need to be connected to the designated pins. If the number of fans is limited, it is also possible to connect fewer than three fans.

| | Fan 1 pin | Fan 2 pin | Fan 3 pin |
|---|---|---|---|
| TACH(FG) | PA.11 | PA.10 | PA.9 |
| PWM(QSPI MOSI) | PA.0 | PA.0 | PA.0 |
| LED_R | PB.5 | PB.2 | PC.2 |
| LED_G | PB.4 | PB.1 | PC.1 |
| LED_B | PB.3 | PB.0 | PC.0 |

  - A pull-up resistor is required on the TACH line to ensure the fan can correctly receive the capture signal.

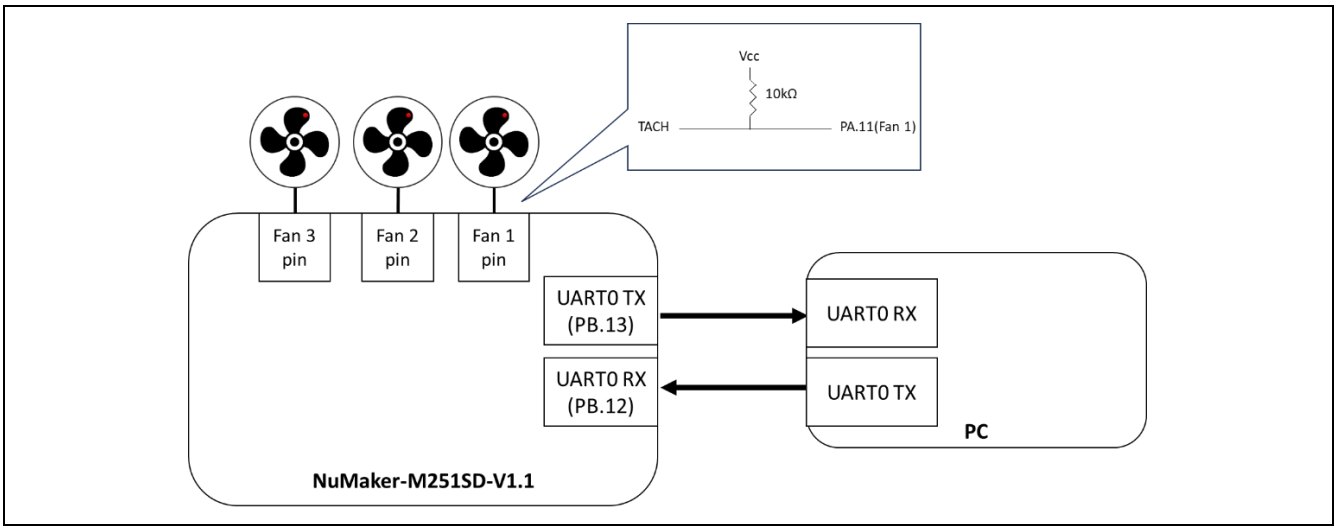



Figure 3-1 Pin Connect

# 4. Directory Information

The directory structure is shown below.

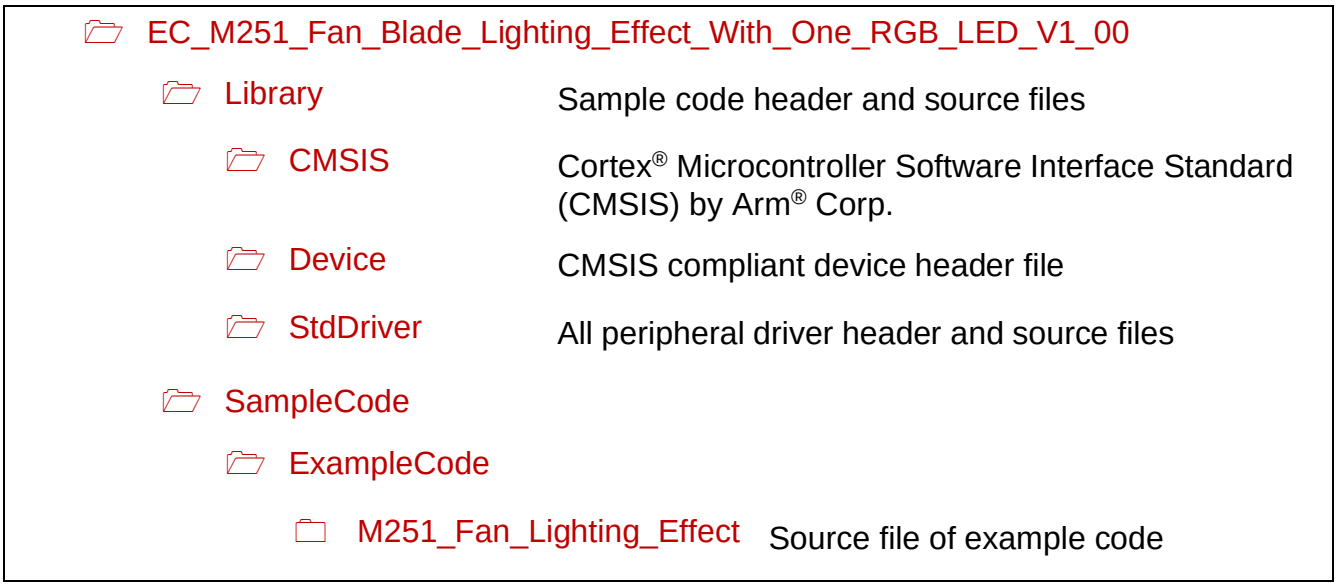



Figure 4-1 Directory Structure

# 5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *M251_Fan_Lighting_Effect.uvprojx*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run
4. Use a terminal program (e.g., Tera Term) to send commands based on the messages returned via UART in order to switch the lighting effects.

# 6. Revision History

| Date | Revision | Description |
|---|---|---|
| 2025.07.01 | 1.00 | Initial version. |

## Important Notice

**Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".**

**Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.**

**All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.**

*Please note that all data and specifications are subject to change without notice.*
*All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.*