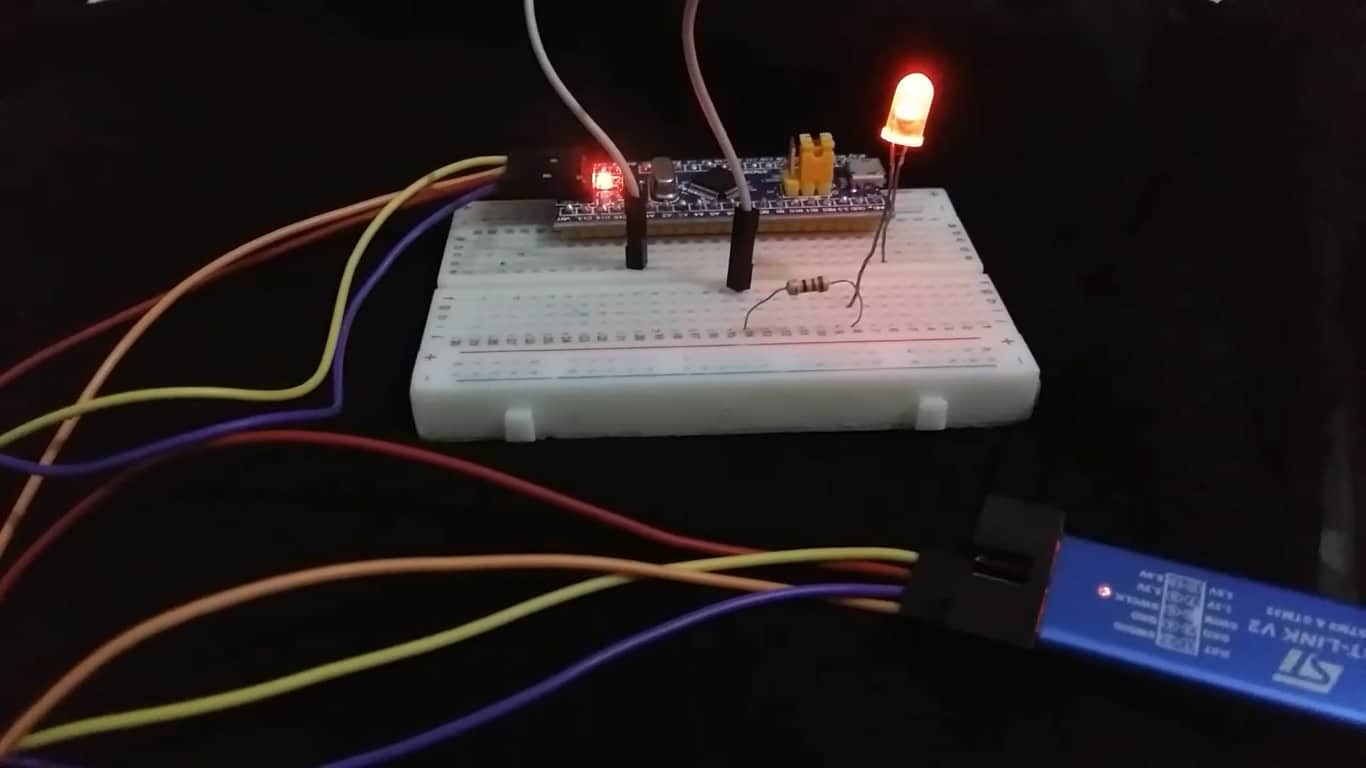
**STM32 Blue Pill Timer in PWM Mode with LED Dimmer Example**

In this tutorial, we will learn to use STM32 Blue Pill timer PWM mode and how to configure them to generate signals with different duty cycle and frequency. We will use an LED dimmer example to demonstrate how to setup the timer in PWM mode using STM32Cube IDE and HAL libraries along with a comprehensive guide about the PWM mode.



**STM32 Blue Pill Timer PWM Mode**

The Blue Pill STM32F103C8 comes with four timers known as TIM1, TIM2, TIM3, and TIM4. They act as a clock and are used to keep track of time based events. The timer module can work in different configurations such as timer mode, counter mode, PWM mode, output compare mode etc. This guide focuses on configuring the timer module in PWM mode.

When configuring the STM32 Blue Pill timer module in PWM mode, an internal clock source clocks the timer module and generates a digital waveform (PWM signal) on the output channel pin. The value of the output compare register which is OCR is continuously monitored and matched with the increasing values of the Timer register. At the particular moment when both the values are same, the state of the output pin is toggled until the end of the period. This occurs repeatedly.

What exactly happens is that when the timer gets clocked, the timer counter counts till the auto reload value. When the value is reached, the state of the output channel pin is set HIGH. When the timer counter counts till the CCRx register vale, the state of the output channel pin is set LOW as a match occurs. The timer counter starts again and the process repeats continuously.

The PWM signal has a specific frequency set by the internal clock, prescaler and the TIMx\_ARRx register, duty cycle held in the channel CCRx register value and resolution. The TIMx\_CCRx register controls the duty cycle and the frequency is correlated with the PWM resolution. Let us look at all these features in detail.

A PWM signal has three properties which are of utmost importance. This includes the frequency of the signal, its duty cycle and the resolution.

**STM32 PWM Frequency**

To set a particular frequency of the PWM signal, the ARR value, prescaler value and the frequency of the internal clock are used. The frequency of the output PWM signal can be calculated as follows:

**F(pwm) = F(clk) / (ARR+1)(PSC+1)**

To get your desired output PWM frequency, the user sets the prescaler and internal clock frequency and finds the ARR value that will generate the required PWM frequency.

**STM32 PWM Duty Cycle**

The CCRx register is responsible for controlling the duty cycle percentage of the PWM signal. This value is calculated by CCRx/ ARR to find the duty cycle percentage value.

**STM32 PWM Resolution**

The resolution of the PWM signal is also an important aspect of the output signal that is generated. It sets the number of discrete duty cycle levels which means the number of steps it takes for the duty cycle to get to the maximum number set. This step size can be calculated by using the frequency of the internal clock and the PWM signal.

**PWM Resolution = log( F(clk)/F(pwm) )/ log(2)**

The table below shows how the PWM resolution varies with different frequencies of the PWM signal for a STM32 Blue Pill 16 bit timer with clock frequency of 72 MHz.

|  |  |
| --- | --- |
| PWM Resolution (bits) | Approximate PWM Frequency |
| 16 | 1.1 kHz |
| 14 | 4.4 kHz |
| 12 | 17.5 kHz |
| 10 | 70 kHz |
| 8 | 281 kHz |
| 6 | 1.125 MHz |
| 4 | 4.5 MHz |

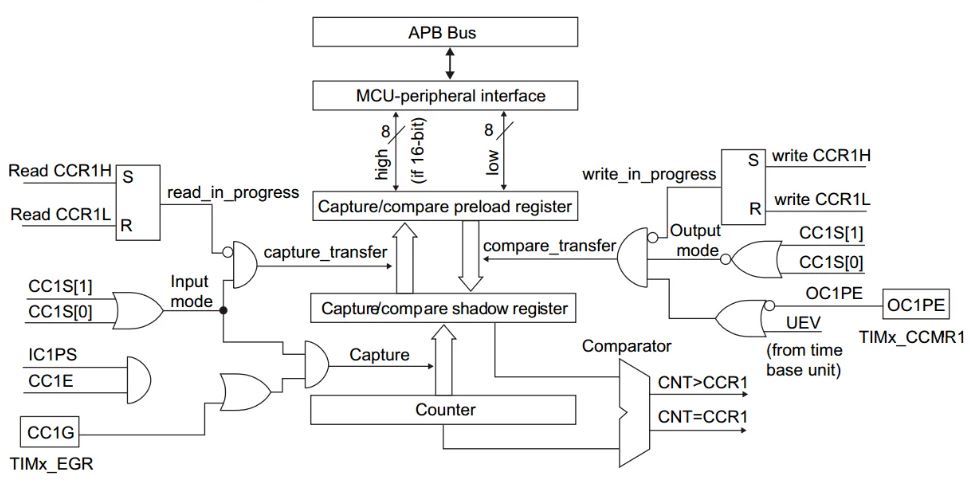
There is another formula as well which involves the ARR value instead of the frequencies. You can use this formula as well to achieve the desired PWM resolution in bits.

**PWM Resolution in bits = log(ARR+1)/ log(2)**

**STM32 Blue Pill Timer PWM Output Channels**

As mentioned previously, the STM32 Blue Pill consists of four timer where each timer has multiple channels that can be configured. This allows the user to set up multiple PWM signals with varying duty cycles by using the same timer. As the timer will be same, therefore all the PWM signals generated like this will be synchronized.

The figure below shows the main circuit for capture channel 1.



As you may see in the circuit above, the capture channel consists of a capture register with a shadow register, an input stage consisting of a digital filter, a multiplexer, and Prescaler for capturing, and an output stage consisting of output control and a comparator. Here the output stage is responsible for creating a reference waveform.

**PWM Modes**

The PWM mode (mode 1 or mode 2) can be configured separately for each output channel. Therefore, each OCx output can have a PWM signal associated with it. The mode is selected by writing the OCxM bits which are present in the TIMx\_CCMRx register. For PWM mode 1 bits ‘110’ are written and for PWM mode 2, bits ‘111’ are written. The OCxPE bit present in the TIMx\_CCMRx register and the ARPE bit present in the TIMx\_CR1 register is enabled by the user according to the particular preload register and the auto preload register respectively.

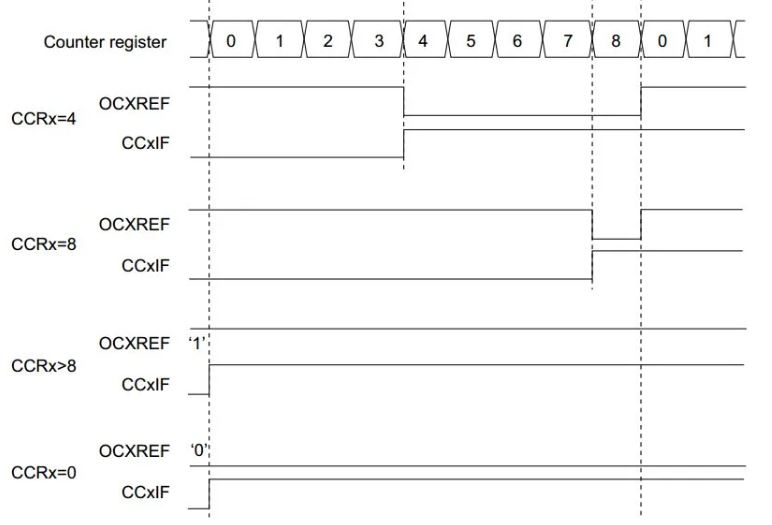
When setting the timer module operation in counter mode, there are different counting modes the user can choose from. This includes the up-counting/down-counting (edge aligned) and center aligned mode etc.

**Edge Aligned Mode**

The edge aligned mode consists of configurations where the PWM mode may be set up in up-counting or down-counting mode.

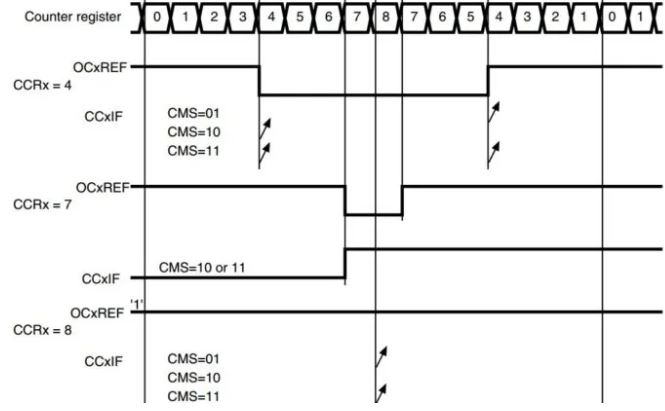
As seen in the figure below for edge aligned waveforms, the reference PWM signal which is known as OCxREF has a HIGH state when TIMx\_CNT is less than TIMx\_CCRx. When this condition is not met, its state goes to LOW. Moreover, when the compare value in TIMx\_CCRx register has a value greater than the auto-reload value held in TIMx\_ARR register then the refence PWM signal stays HIGH. Otherwise, if the compare value is 0 then it stays LOW.

Note that in both PWM modes, either 1 or 2, the values TIMx\_CNT and TIMx\_CCRx are matched to check which value (TIMx\_CNT or TIMxCCRx) is lesser than equal the other, according to the direction of the counter set.

Edge Aligned PWM Waveforms

**Center Aligned Mode**

In case of center aligned mode, the compare flag may be set according to different counting ways (count up, count down or count up and down). This counter’s counting mechanism depends on the configuration of the CMS bits in the TIMx\_CR1 register. In the figure below, you can view the center aligned PWM signals where TIMx\_ARR=8 in PWM mode 1.

Center Aligned PWM Waveforms

**STM32 Blue Pill LED Dimmer Code**

We will set up a code in which the duty cycle is varied by writing the value to the CCR1 register.

Now let us look at our main.c file that was generated. Inside the main.c file, make sure the following code is part of your script by including the lines of code given below.

#include "main.h"

TIM\_HandleTypeDef htim2;

void SystemClock\_Config(void);

static void MX\_GPIO\_Init(void);

static void MX\_TIM2\_Init(void);

int main(void)

{

int32\_t dutyCycle = 0;

HAL\_Init();

SystemClock\_Config();

MX\_GPIO\_Init();

MX\_TIM2\_Init();

HAL\_TIM\_PWM\_Start(&htim2, TIM\_CHANNEL\_1);

while (1)

{

while(dutyCycle < 65535)

{

TIM2->CCR1 = dutyCycle;

dutyCycle += 100;

HAL\_Delay(1);

}

while(dutyCycle > 0)

{

TIM2->CCR1 = dutyCycle;

dutyCycle -= 100;

HAL\_Delay(1);

}

}

}

**Working of the Code**

First we create a variable called ‘duty\_cycle’ which initially holds the value 0. Then we initialize all the configured peripherals and set the system clock configuration. After that we start the PWM signal generation by calling HAL\_TIM\_PWM\_Start() function. This function takes in two parameters. The first parameter is the pointer to the TIM\_HandleTypeDef structure that holds the configuration parameters for the timer module. The second parameter is the timer channel that will be enabled. It can take either of the values listed below:

* TIM\_CHANNEL\_1: TIM Channel 1 is enabled
* TIM\_CHANNEL\_2: TIM Channel 2 is enabled
* TIM\_CHANNEL\_3: TIM Channel 3 is enabled
* TIM\_CHANNEL\_4: TIM Channel 4 is enabled

HAL\_TIM\_PWM\_Start(&htim2, TIM\_CHANNEL\_1);

Inside the infinite while() loop, we will run two while loops that will write the timer2 value to the CCR1 register which holds the duty cycle. The duty cycle will vary from 0 to 100% as its value is increased/decreased by 100 in each loop, that will alter the brightness of the LED according to the duty cycle.

while (1)

{

while(dutyCycle < 65535)

{

TIM2->CCR1 = dutyCycle;

dutyCycle += 100;

HAL\_Delay(1);

}

while(dutyCycle > 0)

{

TIM2->CCR1 = dutyCycle;

dutyCycle -= 100;

HAL\_Delay(1);

}

}

**main.c file**

This is how a complete main.c file will be after modification.

#include "main.h"

TIM\_HandleTypeDef htim2;

void SystemClock\_Config(void);

static void MX\_GPIO\_Init(void);

static void MX\_TIM2\_Init(void);

int main(void)

{

int32\_t dutyCycle = 0;

HAL\_Init();

SystemClock\_Config();

MX\_GPIO\_Init();

MX\_TIM2\_Init();

HAL\_TIM\_PWM\_Start(&htim2, TIM\_CHANNEL\_1);

while (1)

{

while(dutyCycle < 65535)

{

TIM2->CCR1 = dutyCycle;

dutyCycle += 100;

HAL\_Delay(1);

}

while(dutyCycle > 0)

{

TIM2->CCR1 = dutyCycle;

dutyCycle -= 100;

HAL\_Delay(1);

}

}

}

/\*\*

\* @brief System Clock Configuration

\* @retval None

\*/

void SystemClock\_Config(void)

{

RCC\_OscInitTypeDef RCC\_OscInitStruct = {0};

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = {0};

/\*\* Initializes the RCC Oscillators according to the specified parameters

\* in the RCC\_OscInitTypeDef structure.

\*/

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSE;

RCC\_OscInitStruct.HSEState = RCC\_HSE\_ON;

RCC\_OscInitStruct.HSEPredivValue = RCC\_HSE\_PREDIV\_DIV1;

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON;

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_ON;

RCC\_OscInitStruct.PLL.PLLSource = RCC\_PLLSOURCE\_HSE;

RCC\_OscInitStruct.PLL.PLLMUL = RCC\_PLL\_MUL9;

if (HAL\_RCC\_OscConfig(&RCC\_OscInitStruct) != HAL\_OK)

{

Error\_Handler();

}

/\*\* Initializes the CPU, AHB and APB buses clocks

\*/

RCC\_ClkInitStruct.ClockType = RCC\_CLOCKTYPE\_HCLK|RCC\_CLOCKTYPE\_SYSCLK

|RCC\_CLOCKTYPE\_PCLK1|RCC\_CLOCKTYPE\_PCLK2;

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_PLLCLK;

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV2;

RCC\_ClkInitStruct.APB2CLKDivider = RCC\_HCLK\_DIV1;

if (HAL\_RCC\_ClockConfig(&RCC\_ClkInitStruct, FLASH\_LATENCY\_2) != HAL\_OK)

{

Error\_Handler();

}

}

/\*\*

\* @brief TIM2 Initialization Function

\* @param None

\* @retval None

\*/

static void MX\_TIM2\_Init(void)

{

/\* USER CODE BEGIN TIM2\_Init 0 \*/

/\* USER CODE END TIM2\_Init 0 \*/

TIM\_ClockConfigTypeDef sClockSourceConfig = {0};

TIM\_MasterConfigTypeDef sMasterConfig = {0};

TIM\_OC\_InitTypeDef sConfigOC = {0};

/\* USER CODE BEGIN TIM2\_Init 1 \*/

/\* USER CODE END TIM2\_Init 1 \*/

htim2.Instance = TIM2;

htim2.Init.Prescaler = 0;

htim2.Init.CounterMode = TIM\_COUNTERMODE\_UP;

htim2.Init.Period = 65535;

htim2.Init.ClockDivision = TIM\_CLOCKDIVISION\_DIV1;

htim2.Init.AutoReloadPreload = TIM\_AUTORELOAD\_PRELOAD\_ENABLE;

if (HAL\_TIM\_Base\_Init(&htim2) != HAL\_OK)

{

Error\_Handler();

}

sClockSourceConfig.ClockSource = TIM\_CLOCKSOURCE\_INTERNAL;

if (HAL\_TIM\_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL\_OK)

{

Error\_Handler();

}

if (HAL\_TIM\_PWM\_Init(&htim2) != HAL\_OK)

{

Error\_Handler();

}

sMasterConfig.MasterOutputTrigger = TIM\_TRGO\_RESET;

sMasterConfig.MasterSlaveMode = TIM\_MASTERSLAVEMODE\_DISABLE;

if (HAL\_TIMEx\_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL\_OK)

{

Error\_Handler();

}

sConfigOC.OCMode = TIM\_OCMODE\_PWM1;

sConfigOC.Pulse = 0;

sConfigOC.OCPolarity = TIM\_OCPOLARITY\_HIGH;

sConfigOC.OCFastMode = TIM\_OCFAST\_DISABLE;

if (HAL\_TIM\_PWM\_ConfigChannel(&htim2, &sConfigOC, TIM\_CHANNEL\_1) != HAL\_OK)

{

Error\_Handler();

}

/\* USER CODE BEGIN TIM2\_Init 2 \*/

/\* USER CODE END TIM2\_Init 2 \*/

HAL\_TIM\_MspPostInit(&htim2);

}

/\*\*

\* @brief GPIO Initialization Function

\* @param None

\* @retval None

\*/

static void MX\_GPIO\_Init(void)

{

/\* GPIO Ports Clock Enable \*/

\_\_HAL\_RCC\_GPIOD\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

}

/\* USER CODE BEGIN 4 \*/

/\* USER CODE END 4 \*/

/\*\*

\* @brief This function is executed in case of error occurrence.

\* @retval None

\*/

void Error\_Handler(void)

{

/\* USER CODE BEGIN Error\_Handler\_Debug \*/

/\* User can add his own implementation to report the HAL error return state \*/

\_\_disable\_irq();

while (1)

{

}

/\* USER CODE END Error\_Handler\_Debug \*/

}

#ifdef USE\_FULL\_ASSERT

/\*\*

\* @brief Reports the name of the source file and the source line number

\* where the assert\_param error has occurred.

\* @param file: pointer to the source file name

\* @param line: assert\_param error line source number

\* @retval None

\*/

void assert\_failed(uint8\_t \*file, uint32\_t line)

{

/\* USER CODE BEGIN 6 \*/

/\* User can add his own implementation to report the file name and line number,

ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) \*/

/\* USER CODE END 6 \*/

}

#endif /\* USE\_FULL\_ASSERT \*/

Save the main.c file after modifying it. Now we are ready to build our project.

**Building the Project**

To build our project press **Ctrl + B or go to Project > Build All**.

Your project will start building. After a few moments, your project will be successfully built if there are no errors.

**Connecting ST-Link Programmer with STM32**

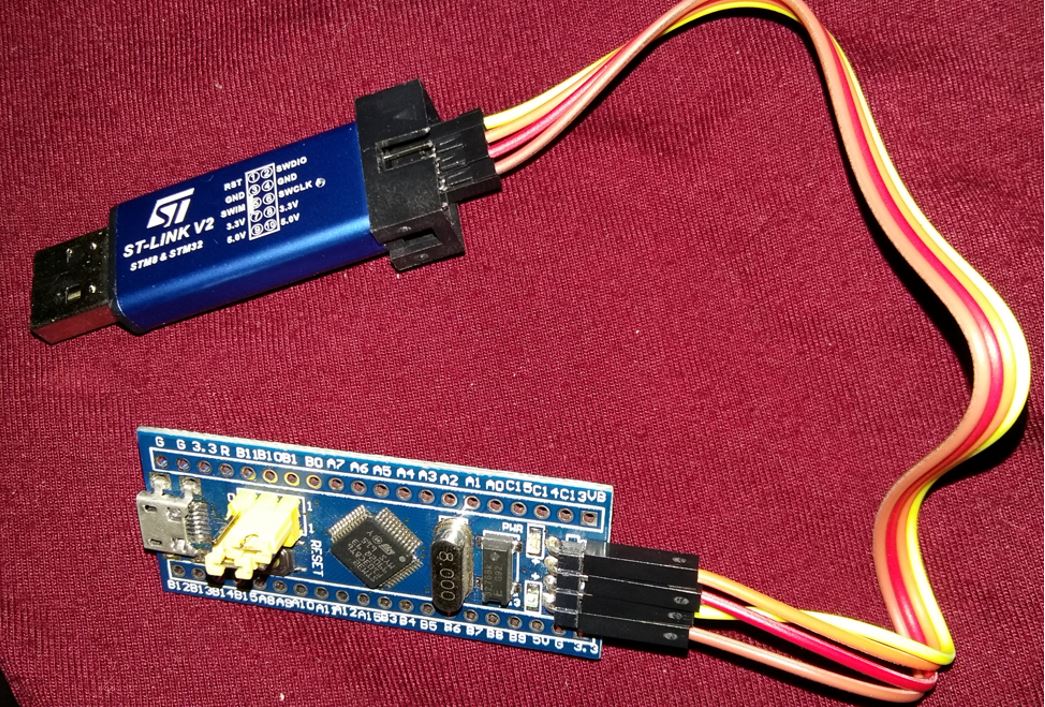
Now as we have successfully built our project let us move ahead and upload the code to our STM32 board. To do that, first we will have to connect our Blue Pill STM32 with a ST-Link programmer. We will be using ST-Link V2.



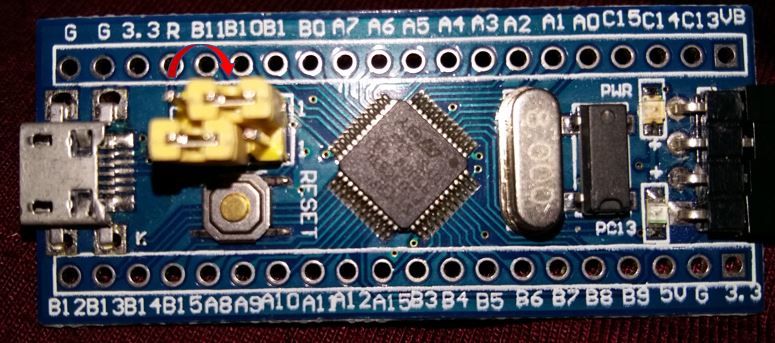
This will provide an interface between our computer and our STM32 board. It consists of 10 pins. We will be using pin2 SWDIO, pin6 SWCLK, pin4 GND, and pin8 3.3V to connect with our STM32 board. The SWDIO is the data input/output pin and the SWCLK is the clock pin. Follow the pin configuration given on the ST-LINK V2 to identify each pin.

Follow the table below to connect both devices correctly.

|  |  |
| --- | --- |
| STM32 | ST-LINK V2 |
| VCC 3.3V pin | pin8 3.3V |
| SWDIO pin | pin2 SWDIO |
| SWCLK pin | pin6 SWCLK |
| GND pin | pin4 GND |



Additionally, move the BOOT jumper to the right to enable the microcontroller to go into programming mode.

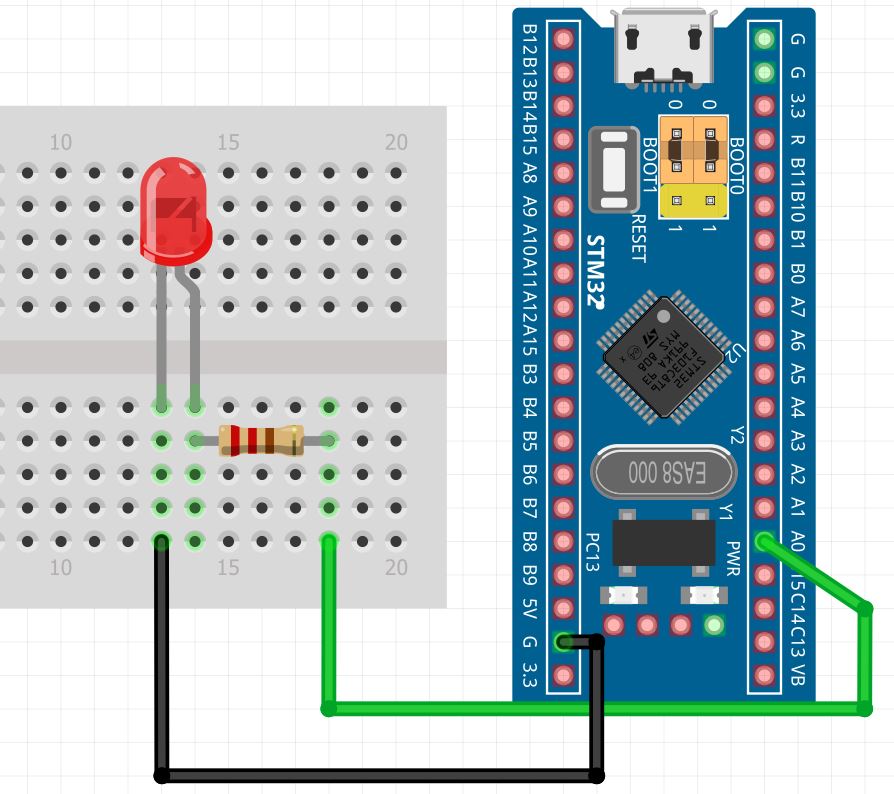


**Hardware Setup for STM32 Blue Pill LED Dimmer**

For this project we will require the following components:

1. STM32 Blue Pill
2. One 5mm LED
3. One 220 ohm current limiting resistor
4. Connecting Wires
5. Breadboard

Connect the components as shown in the schematic diagram below:



Connect the LED’s anode pin to PA0 through a 220-ohm resistor. The cathode pin will be grounded.

**Demonstration**

* Now connect your ST-LINK V2 with your computer via the USB port. Both devices will power ON.
* Next press the RUN button in the IDE. The ‘Edit configuration’ window will open up. Click ‘OK’.
* After a few moments, the code will be successfully sent to the STM32 board. Otherwise, press the RESET button on your STM32 board.
* Now to bring the Blue pill back to normal mode make sure you bring the BOOT jumper back at its place.

Press the Reset button of STM32. The brightness of the LED increases from 0 to 100% then decreases from 100% to 0. The process repeats endlessly.

You may also like to read:

* [STM32 Blue Pill Timer Input Capture Mode with Frequency Measurement Example](https://microcontrollerslab.com/stm32-blue-pill-timer-input-capture-mode-frequency-measurement/)
* [STM32 Blue Pill Timer Encoder Mode with Rotary Encoder Example](https://microcontrollerslab.com/stm32-blue-pill-timer-encoder-mode-rotary-encoder/)
* [STM32 Blue Pill Timer in Counter Mode with STM32Cube IDE and HAL Libraries](https://microcontrollerslab.com/stm32-blue-pill-timer-counter-mode-stm32cube-ide/)
* [STM32 Blue Pill Timer Interrupt with STM32Cube IDE and HAL Libraries](https://microcontrollerslab.com/stm32-blue-pill-timer-interrupt-stm32cube-ide-hal-libraries/)
* [STM32 Blue Pill External Interrupts with STM32Cube IDE and HAL Libraries](https://microcontrollerslab.com/stm32-blue-pill-external-interrupts-stm32cube-ide/)

Categories[STM32 Blue Pill](https://microcontrollerslab.com/category/stm32-blue-pill/)