PWM Library

Table of Contents

Introduction

This library, written by Liang Nie, should make it easy for the user to effectively use all PWM channels on the development board type C. The library, denoted from this point on as PWML gives the user the ability to modify both the duty cycle and period of the PWM. Furthermore, the PWML is able to do simple calculations to simplify finding the period. Finally, it gives 2 options for determining the duty cycle.

Advantages/Disadvantages

The PWML has advantages and disadvantages so it is important to know when to use this library and when to dig through the code to write what it does by hand.

Advantages	Disadvantages
• Easy to use • Minimal calculations needed if any	• A bit clunky

• Gives 2 modes of duty cycle modification • Allows for a change of frequency on the fly in the code • I wrote it :>	• Takes up excess storage on the development board type C • Not the most efficient running program • I wrote it :<

Consider the advantages and disadvantages of the library before implementing it into your library.

Setup

Written in C, the PWML requires some steps to set the library up:

1. Drag pwm.c in <ProjectName>/core/Src
2. Drag pwm.h in <ProjectName>/core/Inc
3. Head into the .ioc of the project

If you're using Edison's RMTemplate:

1. In Pinout & Configuration
    a. System Core
        i. DMA
            1. Add TIM4_CH3 to DMA1 with priority Low
            2. Add TIM8_CH1/CH2/CH3 to DMA2 with priority Low
            3. Add TIM1_CH1 to DMA2 with priority Low

If you're starting from a blank CubeIDE document:

1. Clock Configuration (Recommend setting Input Frequency and then resolving) [Figure 1]
    a. Input Frequency = 12MHz
    b. Enable HSE
    c. Select PLLCLK
    d. Enable CSS
    e. APB1 Peripheral = 42MHz
    f. APB1 Timer = 84MHz
    g. APB2 Peripheral = 84MHz
    h. APB2 Timer = 168 MHz
2. Pinout & Configuration [Figure 2]
    a. Pinout View
        i. PE9 = TIM1_CH1
        ii. PE11 = TIM1_CH2
        iii. PE13 = TIM1_CH3
        iv. PE14 = TIM1_CH4

        v. PD14 = TIM4_CH3

        vi. PH10 = TIM5_CH1
        vii. PH11 = TIM5_CH2
        viii. PH12 = TIM5_CH3

ix. PC6 = TIM8_CH1
x. PI6 = TIM8_CH2
xi. PI7 = TIM8_CH3

3. System Core
    a. DMA
        i. Add TIM4_CH3 to DMA1 with priority Low
        ii. Add TIM8_CH1/CH2/CH3 to DMA2 with priority Low
        iii. Add TIM1_CH1 to DMA2 with priority Low
4. Timers (Counter Period doesn't matter as much as the PWML takes care of it every time you call *PWMOutput()*)
    a. TIM1 Parameter Settings
        i. Prescaler = 335
        ii. Counter Period = 999
    b. TIM4 Parameter Settings
        i. Prescaler = 83
        ii. Counter Period = 249
    c. TIM5 Parameter Settings
        i. Prescaler = 83
        ii. Counter Period = 1999
    d. TIM8 Parameter Settings
        i. Prescaler = 335
        ii. Counter Period = 999
5. Add the library to the project by calling *#include "pwm.h"* in main.c in <ProjectName>/core/Src
6. Initialize the function by calling function *PWMInit(&htim1, &htim4, &htim5, &htim8);* Outside and in front of the For/While loop
7. At the end of the For/While loop call the function *PWMTimerStarter();*

After the following, the library should be ready to use.

Library Functions (Quick Overview)

The library comes with 3 functions:

PWMOutput(TypesThatUsePWM_t Type, int8_t Position, uint32_t desiredFrequency);

PWMInitialize(TypesThatUsePWM_t Type, msOrFullRange microsecondOrFullrange, int8_t position, float val);

PWMOff(TypesThatUsePWM_t Type, int8_t Position);

The first 2 functions are needed to run the PWML properly set the PWM signal correctly. Forgetting one or both functions while trying to set up the PWM Signal has a high likelihood of the PWM not generating correctly. (bro please don't be stupid)

The third function is used to stop the pwm in case you don't need to use it anymore. This can also be done with *PWMInitalize()* and setting float val to 0 (either MS OR FR).

Function 1 *PWMOutput()* Parameters:

PWMOutput(TypesThatUsePWM_t Type, int8_t Position, uint32_t desiredFrequency);

- *TypesThatUsePWM_t Type* (typedef enum) [Select which of the 3 you want to set]
  - 0 = Motor
  - 1 = LED
  - 2 = Buzzer
- *int8_t Position* (int8_t) [Indexed at 1]
  - Motor :
    - (1-7 in correspondence to PWM Pins 1-7 on the board)
  - LED :
    - (1-3 in correspondence to BGR (Blue = 1, Green = 2, Red = 3)
  - Buzzer
    - Position doesn't matter
- *uint32_t desiredFrequency* (uint32_t)
  - Set the desired frequency of the PWM signal here. Typically, devices are at 50 or 500 Hz
  - Warning, don't do 0 Hz or like some massive fuckin number please (I would write a warning but tbh that's just natural selection)

Function 2 *PWMInitialize()* Parameters

PWMInitialize(TypesThatUsePWM_t Type, msOrFullRange microsecondOrFullrange, int8_t position, float val);

- *TypesThatUsePWM_t Type* (typedef enum) [Select which of the 3 you want to set]
  - 0 = Motor
  - 1 = LED
  - 2 = Buzzer
- *msOrFullRange microsecondOrFullrange* (typedef enum) [Choose whether you want to set the µs duty cycle or a percentage of the full range]
  - 0 = MS
  - 1 = FR
- *int8_t Position* (int8_t) [Indexed at 1]
  - Motor :
    - (1-7 in correspondence to PWM Pins 1-7 on the board)
  - LED :
    - (1-3 in correspondence to BGR (Blue = 1, Green = 2, Red = 3)
  - Buzzer
    - Position doesn't matter
- *float val* (float)
  - input either the µs duty cycle or percentage of the full range (50% = 0.5) here.

Function 3 *PWMOff()* Parameters

PWMOff(TypesThatUsePWM_t Type, int8_t Position);

- *TypesThatUsePWM_t Type* (Typedef enum) [Select which of the 3 you want to set]
  - 0 = Motor

    - 1 = LED
    - 2 = Buzzer
- *int8_t Position* (int8_t) [Indexed at 1]
    - Motor :
        - (1-7 in correspondence to PWM Pins 1-7 on the board)
    - LED :
        - (1-3 in correspondence to BGR (Blue = 1, Green = 2, Red = 3)
    - Buzzer
        - Position doesn't matter

Library Function (More in depth)

Function 1 *PWMInit()*

PWMInit() it tasked to give the pointers of the timers to the library. This is so that *PWMOutput()*, *PWMOff()*, and *PWMInitialize()* can run properly.

Furthermore, *PWMInit()* runs *HAL_TIME_Base_Start()* for the 4 timers the PWML covers.

This is so that the user doesn't have to manually start the base timers whilst also giving the library pointers to the timers location in memory from the main.c file.

Function 2 *PWMOutput()*

PWMOutput() is tasked with setting the counter period of the PWM signal. Normally the signal is determined beforehand in the .ioc and is not touched. But in the case of the buzzer, to play different frequencies, the counter period has to be able to change. As to not limit the abilities of the other timers too, I've incorperated the ability to almost all of the channels.

Unfortunately, not all timers have access to the DMA through all channels, this means that only some of the timer channels have access to the feature.

Timer channels that are excluded from the feature:

- TIM1_CH2/CH3/CH4
- TIM5_CH1/CH2/CH3

** Testing needed**
Does changing the counter period of TIM1_CH1 change the counter period of TIM1_CH2 as well?
Can you set TIM1_CH2 or does it crash?
** Testing needed**

PWMOutput() has 2 main jobs, calculating the counter period from the frequency given and setting a global array at the given position and type to 1.

For the counter period calculator, the task is relayed to *initializePeriod()* which oversees both the calculation process and the DMA setting.

InitializePeriod() calls *calculateOutputPeriodToGetFrequency()* to calculate and return the calculated counter period as a uint32_t. After that it uses DMA to put the calculated counter period directly into the timer mem location using ARR.

For *calculateOutputPeriodToGetFrequency(),* Held at the top of the library is a set of constants which relates to the prescalers set in the initialization period. To calculate the constants, the following formula was used.

$$constantPrescaler = \frac{prescaler}{APBx}$$

APBx:
APB1 pertains to TIM1 and TIM8
APB2 pertains to TIM4 and TIM5

Because the constant won't change with different frequencies, they were calculated beforehand and put into the library. With those constants, the following formula is derived:

$$counterPeriod = \frac{1}{constantPrescaler * frequency\ given}$$

Currently the calculations end as such, and it puts the counterPeriod into the timers ARR (counter period). Later on, there is a plan to add a rounding feature to the final number and index it to 0.
The reason why indexing it to 0 isn't needed at this moment is because a counter period change of 1 has minimal effect on the actual frequency of the signal. Furthermore, the conversion from floats to uint32_t normally rounds down as floats are generally a tiny bit smaller than the actual number. This rounding down quark helps us.

After returning, *initializePeriod()* then saves the calculated value into the corresponding position of an array uint32_t period[11] . This value is used later on in the *PWMInitialize()* process.

As for the second task of the function. There's an array *int8_t whichPWMisOn[11]* which keeps track of every PWM channel that is on or off. This function is used to change the corresponding position to a 1.

Function 3 *PWMInitialize()*

PWMInitialize() has two tasks like *PWMOutput().* To find the proper duty cycle from a specified µs step or a percentage of a full range and to set the duty cycle in CCRx. Unlike *PWMOutput()* though, the duty cycle can't be determined beforehand in the .ioc, which makes sense. Therefore, *PWMInitialize()* is the library to dissect if the user plans to learn from the PWML.

PWMInitialize() has two ways to determine the duty cycle that the user wants. MS or FR.

MS or microseconds is used when the pwm signal has only works from a set microsecond range. For instance, normal servos use a range from 1000µs to 2000µs. This duty cycle is irrelevant

from the counter period of the signal. And will always find the closest duty cycle to the specified amount of microseconds.
FR on the other hand is a percentage of the total period. What this means that if you input 0.1 (10%) in the val section, your duty cycle will be 10% of the entire period. This is useful for all cases in which you want access to the entire bandwidth of the period like an LED or Buzzer.

To calculate the duty cycle, *PWMInitialize()* calls upon *calculateOutputPeriodValue()* to return a uint32_t for the duty cycle.

calculateOutputPeriodValue() breaks into calculating either MS or FR.

To calculate MS, the following equation is used

$$dutyCycle = \frac{\mu sVal}{\mu sPerStep}$$

µsPerStep is the amount of µs a single step of the duty cycle is. This calculations is pretty easy.

As for FR, the task is given to a different function *safeOutputPeriodValueCalculator()*.In said function, the equation is just as simple

$$dutyCycle = ValPercent * counterPeriod$$

Counter period was saved in an array *period[11]* by *InitializePeriod()* as stated earlier and is used here. This calculation also has a safeguard to prevent numbers above the counter period.

After the following duty cycle is calculated, the value is then returned to *PWMInitalize()* and then the duty cycle is set using CCRx where x corresponds to the channel number of the timer.

Function 4 *PWMOff()*

PWMOff() as a function is quite simplistic. All it does is it sets the corresponding position in the array *whichPWMisOn[11]* to 0.

Function 5 PWMTimerStarter()

PWMTimerStarter() should be left at the end of a while loop and should be called once a cycle. This function goes through the entirety of the array *whichPWMisOn[11]* and will either *HAL_TIM_PWM_Start()* or *HAL_TIM_PWM_Stop()* in response to whether it is a 1 or 0.

Figures

Figure 1: Clock Configuration

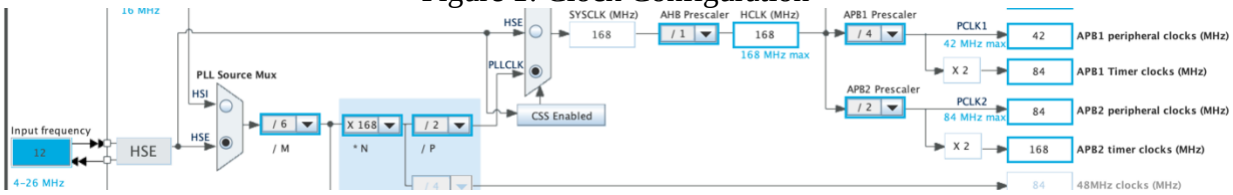

Figure 2: Pinout Configuration