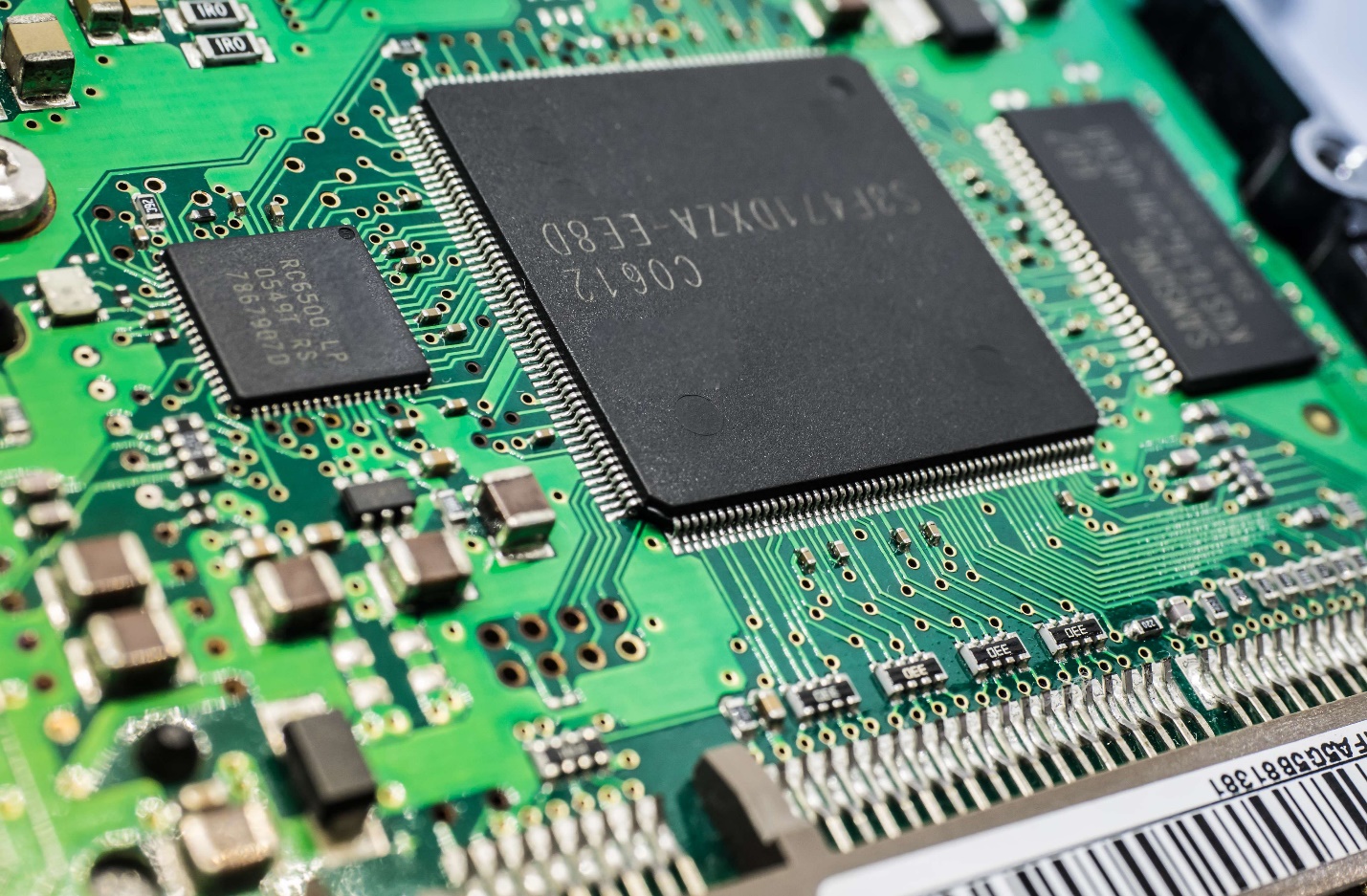
POC for Timers, WatchDog, FreeRTOS

July Trendafilov(4674693)



Contents

[Timers 3](#_Toc119368991)

[Watchdog 4](#_Toc119368992)

[FreeRTOS 6](#_Toc119368993)

# Timers

To setup a timer is STM32, some registers are required to be set. In the following example function SetTim2 is used to setup TIM2 with interrupt.

Text

Description automatically generated

The following code firstly sets up the prescaler divider to match the specified time frame. For example here the Timer counts to 1s. To do so the PSC register must be set to 7200 is order for the Clock rate to be prescaled:

72MHZ : 7200 = 10kHz

By that 1 : 10000 = 0,0001 and one 0,1 ms will pass as the counter ticks. When the counter reaches 10000 that’s when a second has passed and an interrupt is generated. When the Timer reaches the ARR value the interrupt is generated and the flag is set to 1.

Text

Description automatically generated

In the IRQ handler, firstly the interrupt bit must be toggled and then it is printed to the screen that TIM2 interrupt appeared at a frequency of 1 second.

Graphical user interface, text, application

Description automatically generated

# Watchdog

The watchdog is used to reset a system when a task takes too long time. The following function sets up the watchdog by writing a prescaler and a reload value. If the reload value is passes, a reset on the system is done.

Text

Description automatically generated

The watchdog timer has a 40kHz clock rate. To set a specific threshold a prescaler can be used like this:

The IWDG->PR register has the value 5 which means that the clock rate will be divided by 128:

Graphical user interface, text, application, chat or text message

Description automatically generated

40000 : 128 = 312.5 -> 1 : 312.5 = 0.0032 so each tick 3,2 ms pass.

If the value of the RLR register is reached which 1535 ticks, a reset is triggered.

This is printed to the serial monitor and it is outside the infinite loop and it is executed only once on setup.

A screenshot of a computer

Description automatically generated with medium confidence

The time when the watchdog will trigger is 3.2ms \* 1535 ticks = 4.912s if not reloaded. If the watchdog is reloaded in this time frame everything will be fine.

Graphical user interface, text

Description automatically generated

In the main writing 0xAAAA to KR reloads the timer.

Text

Description automatically generated

But if we comment the reload timer register, the watchdog will surpass the threshold. The watchdog will trigger a reset.

Graphical user interface, application

Description automatically generated

This is the serial communication. Every second the TIM2 generates an interrupt and on the 5th second IWDG is triggered.

Graphical user interface, text, application

Description automatically generated

# FreeRTOS

To use freeRTOS in a project one must start the scheduler and initialize the Kernel.

Graphical user interface, text

Description automatically generated

In this example I will use thread flags and mutex:

Text

Description automatically generated

Then is function MX\_FREERTOS\_Init I initialize the mutex and start the threads:

Text

Description automatically generated

Here the mutex has its default attributes so we must manually lock and unlock it and three function are created – one dummy and two for the workshop.

In function AddToCounter, sharedVariable is always incremented until it passes 1000 and then signals to PrintToSerial to take control. AddToCounter is guarded by a mutex and if the value of sharedVariable passes 1000, a flag is set and the mutex is released and then function PrintToScreen acquires the mutex, changes the value of sharedVariable and then releases it and returns the context to AddToCounter and this process is repeated forever.

Text

Description automatically generated

osDelay is used to switch the context.

A screenshot of a computer

Description automatically generated with medium confidence

And the output of the program is:

Graphical user interface, application

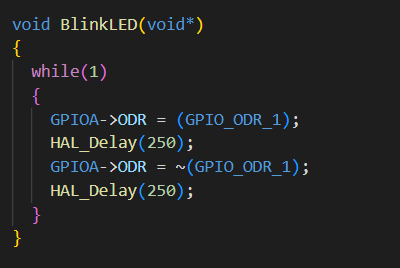
Description automatically generated

As observed when the value passes 1000, 900 is subtracted and the counting start again.

# Challenge FreeRTOS and Timers

For the challenge I have one LED, one servo controlled by PWM and on rotary encoder. The rotary encoder is reused from a previous project, while there are some new concepts here (PWM) and RTOS. The LED is blinking at 0.5 Hz on a separate thread and the servo is controlled by the rotary encoder on a different thread. The threads don’t share resources, so we don’t have synchronization.

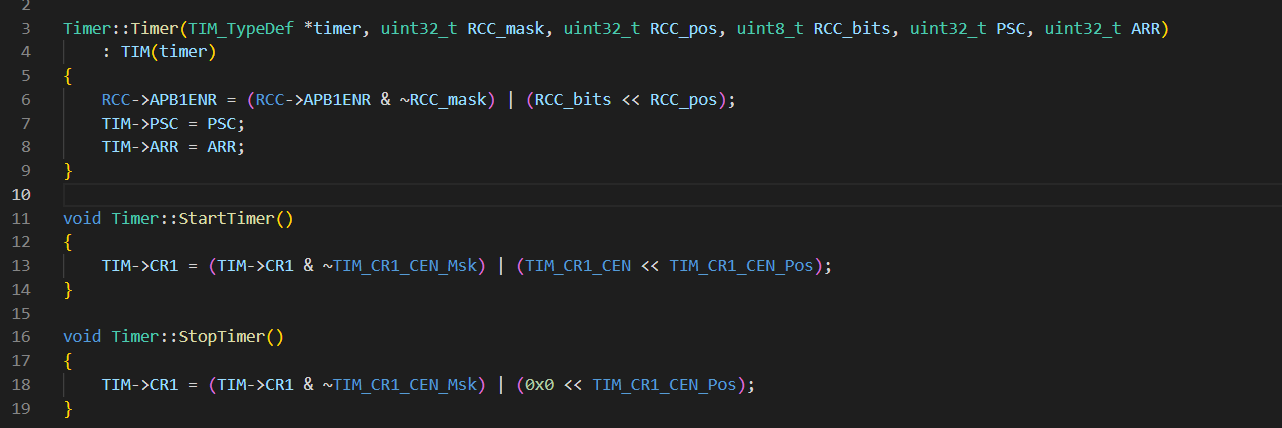
The first part is the Thread for the LED:



Here there is an infinite loop which toggles a LED which is not blocking the other thread.

The second part is the Servo. The new concepts introduced here are timers and PWM. This is done with a class for timers and servos which look like:

The timers class member functions are:



There is a constructor and a start stop method. In the constructor we have 3 important lines:  
1. To enable the timer we want to use in register RCC->APB1ENR (I use TIM2 in this example).

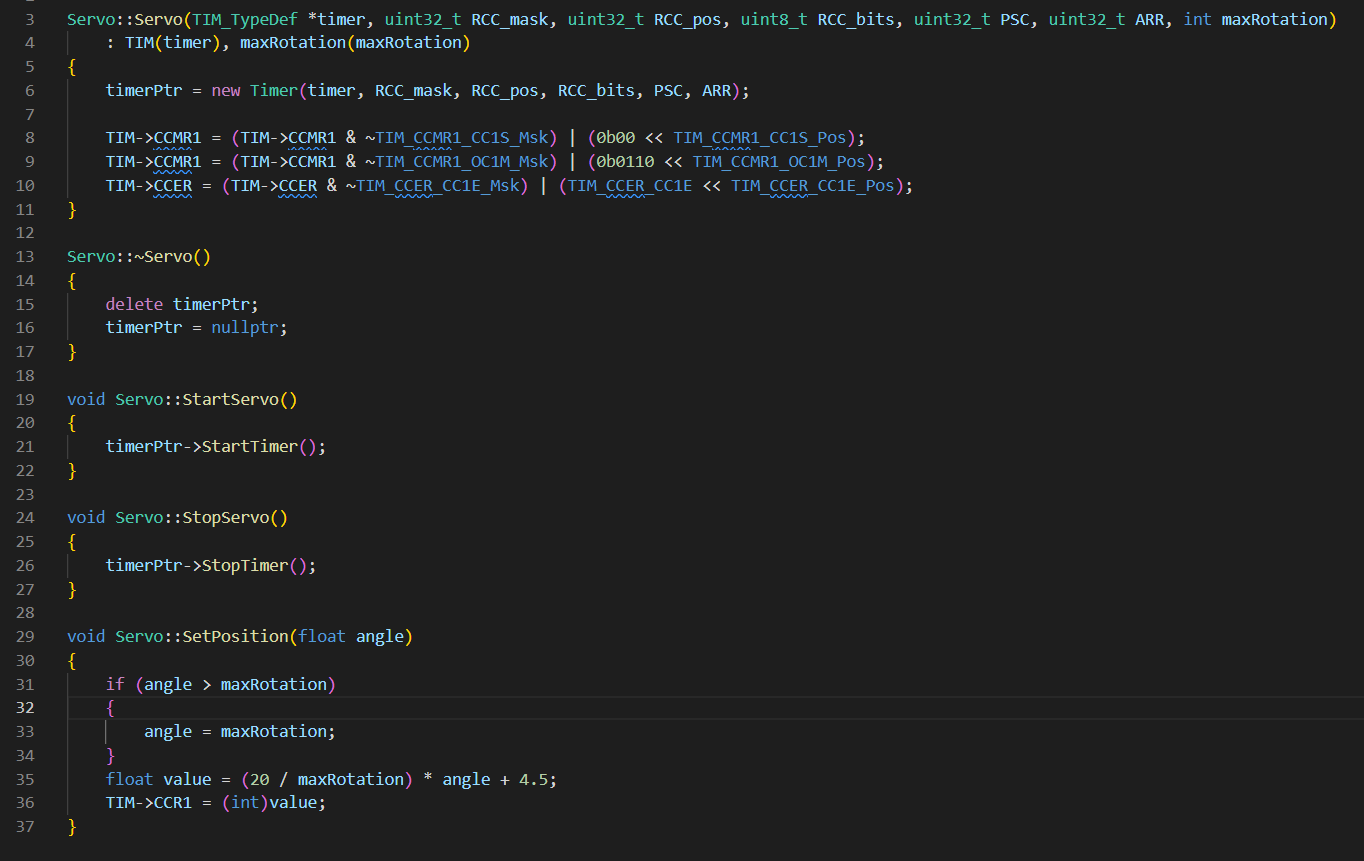
2. To set a prescaler for the timer so that it doesn’t count very fast. In this example I will set it to 1kHz.

3. To set a value at which the timers goes back to 0 and starts counting again

Then we have two functions that start and stop the timer object:

1. To do that we either write 1 or 0 to the first bit of register CR1.

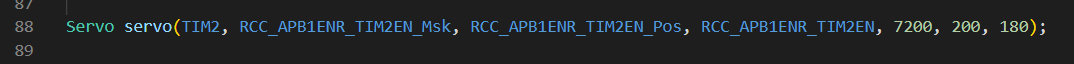
Then the other class servo looks like:

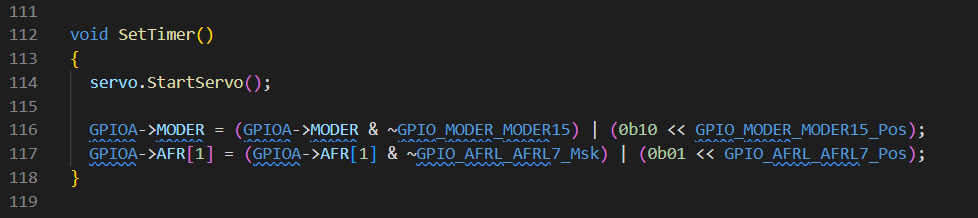


There is a constructor, start and stop and “SetPosition” function. In the constructor we initialize the timer with an object of the timer class. Then we extend the setup of this timer to enable PWM. This is done with 3 operations:

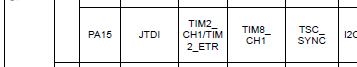
1. First, we must configure that we will use the register as output. (line 8).
2. Second, we set the timer in PWM mode 1 which is up-counting(line 9).
3. Third, we set the capture/compare channel 1 to output by writing to the corresponding bit.

After we have the classes for the timer and PWM generation for the servo we can initialize our PWM timer as:

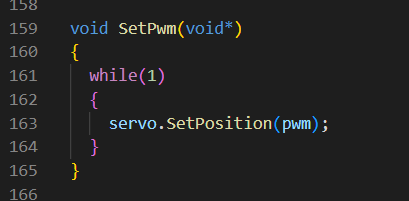




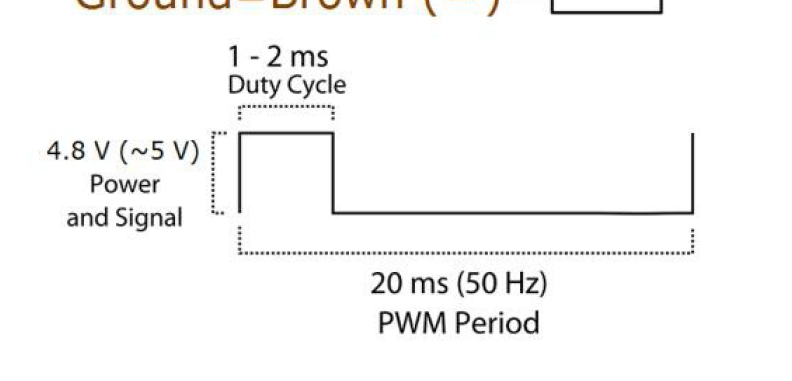
We have a global variable servo which we start in this function. Then in order for the PWM to work, a pin with the correct alternating function must be set. In this case we use GPIOA pin 15 as it can be connected to TIM2 CH1 which we previously have set. To use the tim function, we toggle the second bit of the register (line 117).

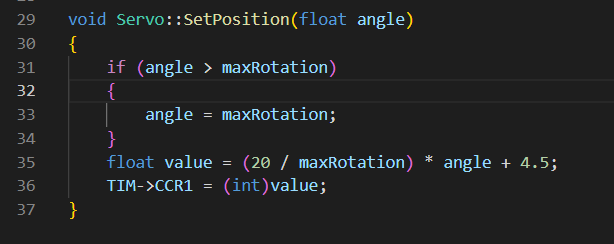


Lastly we have our thread for setting the angle using PWM.



Here in a different thread we use set the angle using our PWM variable which has values [0 – 180] because we use a servo which can rotate only 180°. To control the servo we must generate the correct PWM. To do so in the spec for the servo there is written that the PWM duty cycle must be between 1-2 ms while the whole duty cycle is 20ms (50Hz). The SetPwm is in a thread and when an interrupt occurs, the interrupt modifies the “pwm” variable and then the thread sets the position of the servo which actually sets the duty cycle.





Here the function calculates the correct value for the PWM. If we want to rotate the servo to 90°, the duty cycle must be 1.5ms. and in this case it will be around 14.5. The in the CCR1 register we set the value and when the counter reaches this value the value goes to zero until CNT > ARR. After that the generation starts again. Here we have 10kHz clock so 14.5 will be reached in 1.5ms.

# Conclusion

In this challenge I have learned to control timers and use PWM generation to control actuators. Further, I have learned to use FreeRTOS to have multiple threads and synchronize when necessary.