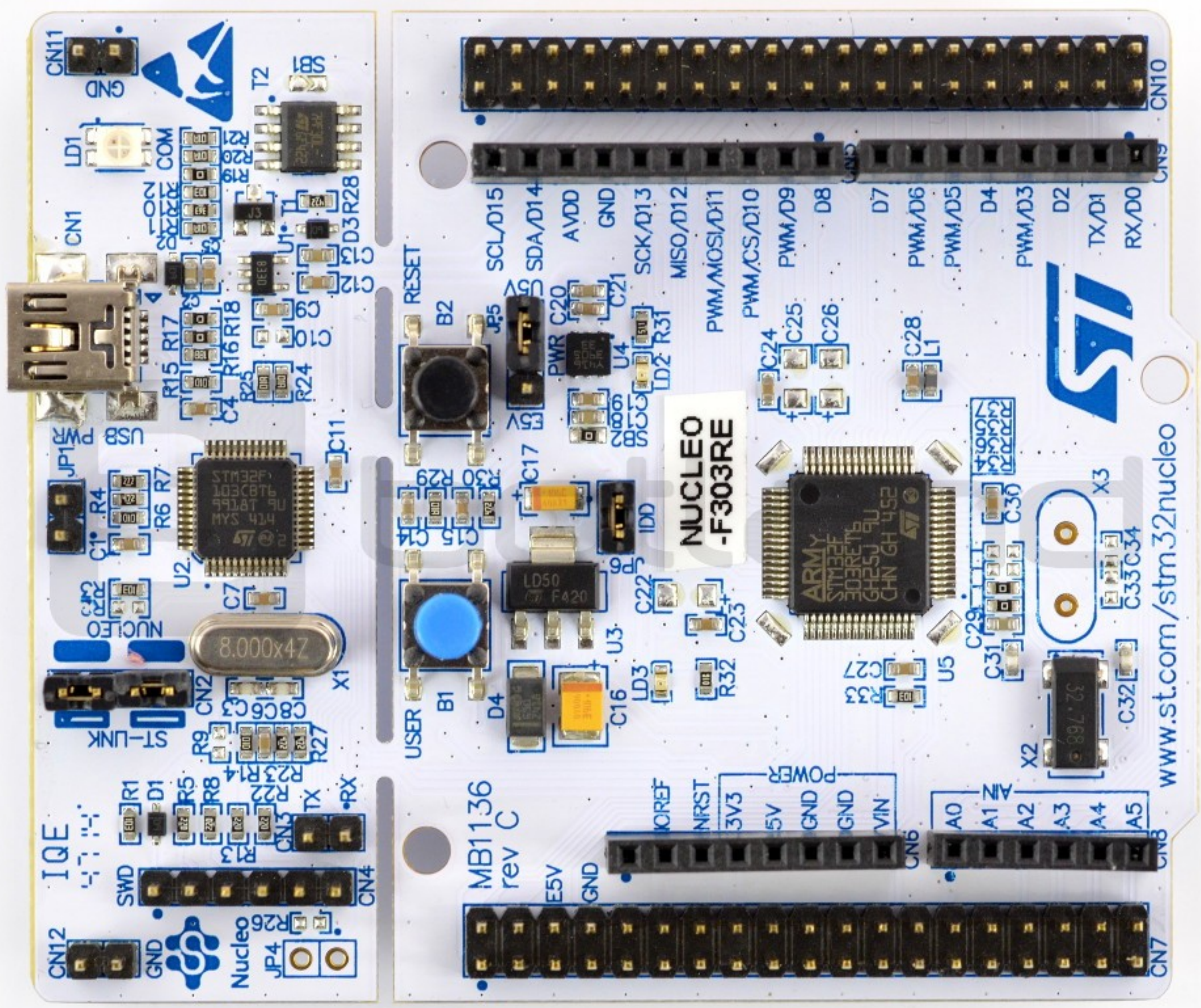
Programming the STM32F303RET6



CMSIS

1. Program

The program initialized two pins BTN (13C) and LED (5A) as follows:

- BTN: function=input, speed=medium, pull=pull_up
- LED: function=output, speed=medium, pull=no_pull

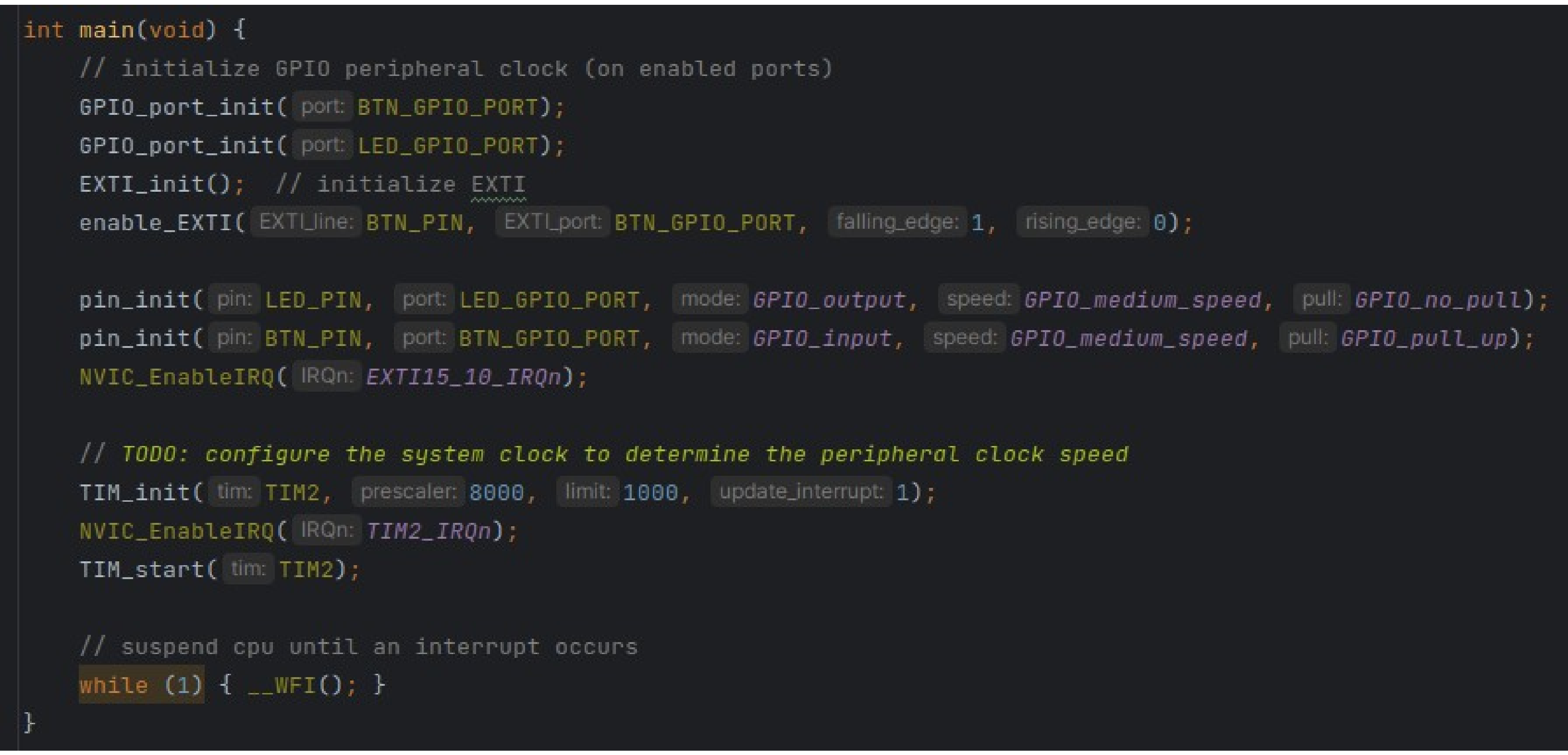
After this the EXTI line is enabled for the BTN pin (EXTI15_10_IRQn)

Then a timer is configured TIM2 with prescaler=8000 and limit=1000

An (update) interrupt is added to this clock so that an interrupt is triggered when the timer exceeds its limit (rolls over)

The timer is started and in the while loop the cpu is set to wait until interupt (suspend cpu / low power mode)

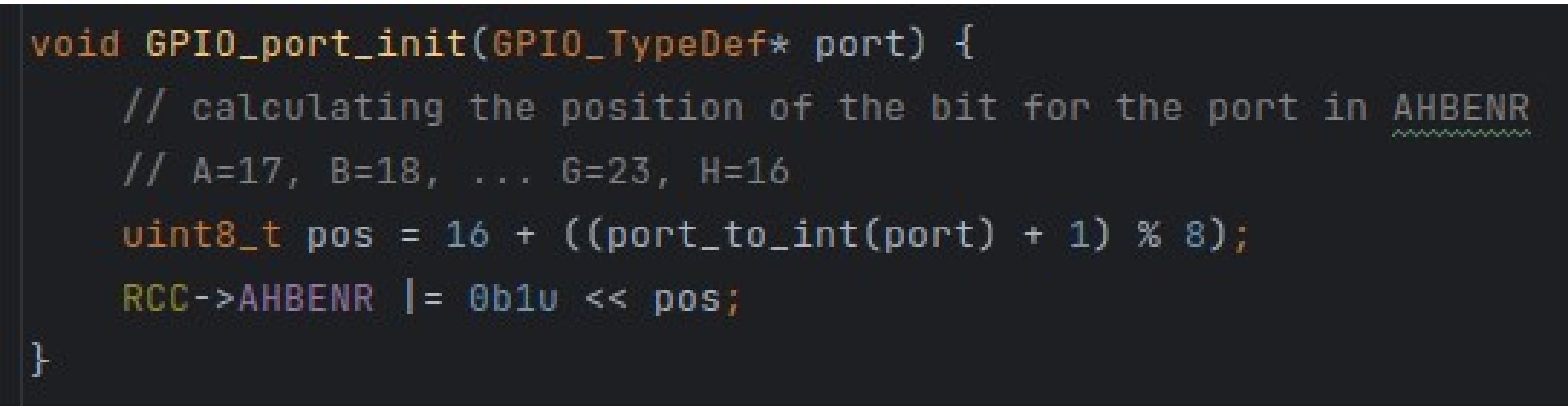
2. Code



```
int main(void) {
    // initialize GPIO peripheral clock (on enabled ports)
    GPIO_port_init( port: BTN_GPIO_PORT);
    GPIO_port_init( port: LED_GPIO_PORT);
    EXTI_init();  // initialize EXTI
    enable_EXTI( EXTI_line: BTN_PIN,  EXTI_port: BTN_GPIO_PORT,  falling_edge: 1,  rising_edge: 0);

    pin_init( pin: LED_PIN,  port: LED_GPIO_PORT,  mode: GPIO_output,  speed: GPIO_medium_speed,  pull: GPIO_no_pull);
    pin_init( pin: BTN_PIN,  port: BTN_GPIO_PORT,  mode: GPIO_input,  speed: GPIO_medium_speed,  pull: GPIO_pull_up);
    NVIC_EnableIRQ( IRQn: EXTI15_10_IRQn);

    // TODO: configure the system clock to determine the peripheral clock speed
    TIM_init( tim: TIM2,  prescaler: 8000,  limit: 1000,  update_interrupt: 1);
    NVIC_EnableIRQ( IRQn: TIM2_IRQn);
    TIM_start( tim: TIM2);

    // suspend cpu until an interrupt occurs
    while (1) { __WFI(); }
}
```

This code implements the behaviour described under 1. Program.

Now lets look at how this is done within these functions.



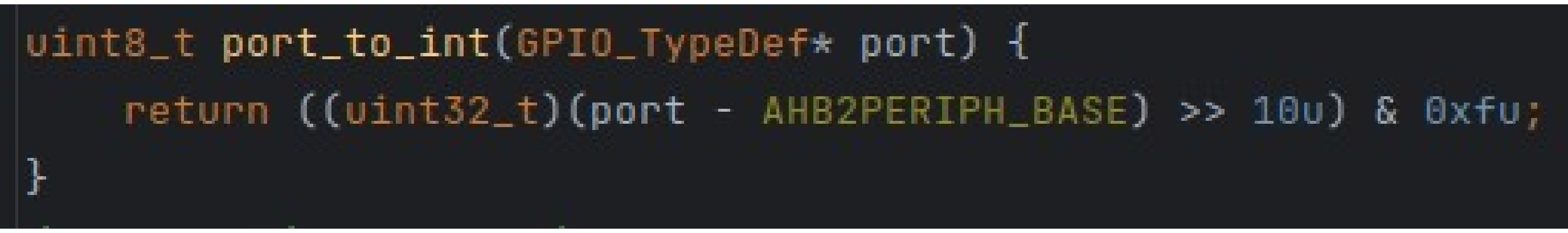
```c
void GPIO_port_init(GPIO_TypeDef* port) {
    // calculating the position of the bit for the port in AHBENR
    // A=17, B=18, ... G=23, H=16
    uint8_t pos = 16 + ((port_to_int(port) + 1) % 8);
    RCC->AHBENR |= 0b1u << pos;
}
```

This code calculates the position for the bit in AHBENR that activates the port clock. The positions are defiened as follows:

- PORTA: BIT-17
- PORTB: BIT-18
- PORTC: BIT-19
- PORTD: BIT-20
- PORTE: BIT-21
- PORTF: BIT-22
- PORTG: BIT-23
- PORTH: BIT-16   !!!

The bit position for PORTH is different than usual so that is why I had to add “16 + (x + 1) % 8”

It calls upon the function port_to_int, it works as follows:

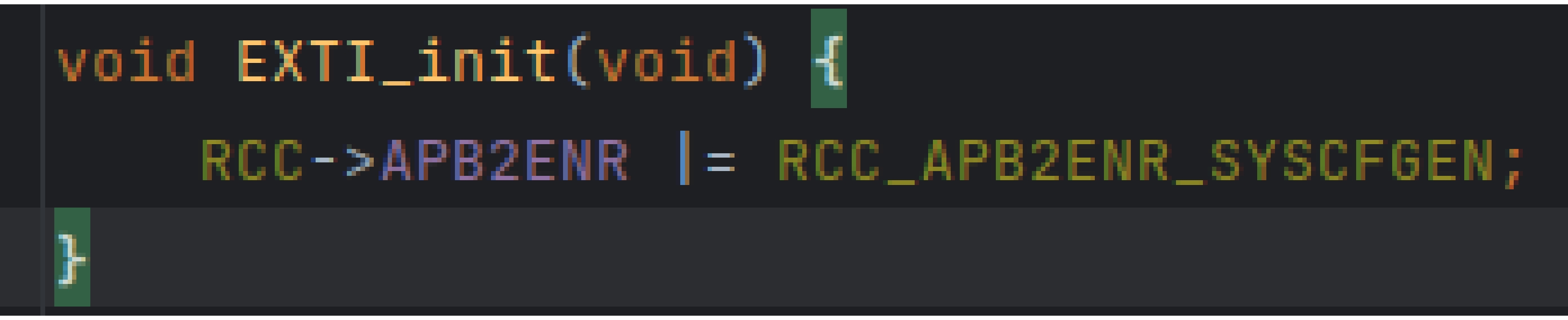


```c
uint8_t port_to_int(GPIO_TypeDef* port) {
    return ((uint32_t)(port - AHB2PERIPH_BASE) >> 10u) & 0xfu;
}
```

This function calculates the port number based on the offset compared to AHB2PERIPH_BASE. The memory layout for the port registers is as follows:

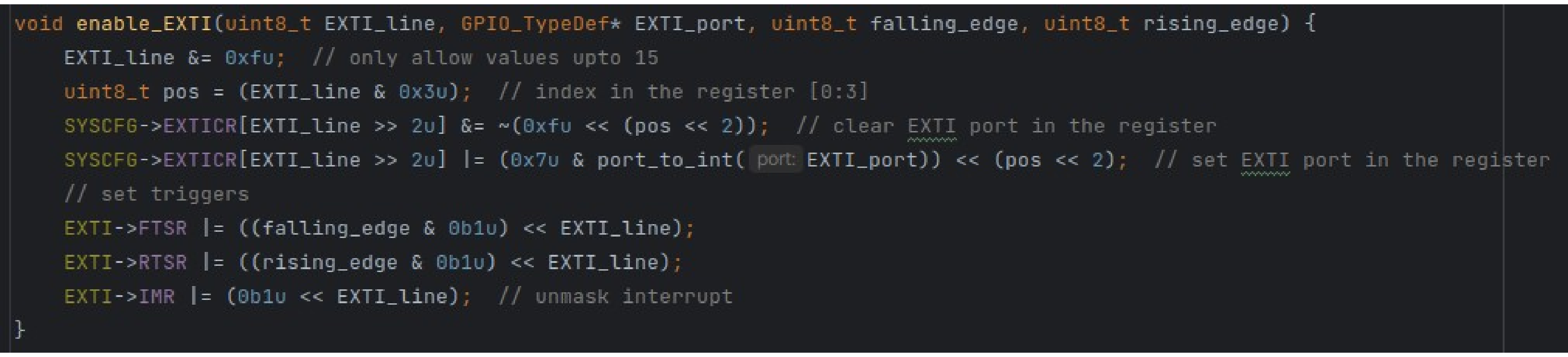
- PORTA: (AHB2PERIPH_BASE + 0x00000000UL)
- PORTB: (AHB2PERIPH_BASE + 0x00000400UL)
- PORTC: (AHB2PERIPH_BASE + 0x00000800UL)
- PORTD: (AHB2PERIPH_BASE + 0x00000C00UL)
- PORTE: (AHB2PERIPH_BASE + 0x00001000UL)
- PORTF: (AHB2PERIPH_BASE + 0x00001400UL)
- PORTG: (AHB2PERIPH_BASE + 0x00001800UL)
- PORTH: (AHB2PERIPH_BASE + 0x00001C00UL)

You can probably see that the ports are at a distance 0x400 from eachother to convert this distance into one I have used “>> 10”. after this an and operation is preformed to ensure that the value does not include any errors outside the lower 4 bits.



```c
void EXTI_init(void) {
    RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
}
```

EXTI_init simply enables its clock



```c
void enable_EXTI(uint8_t EXTI_line, GPIO_TypeDef* EXTI_port, uint8_t falling_edge, uint8_t rising_edge) {
    EXTI_line &= 0xfu;  // only allow values upto 15
    uint8_t pos = (EXTI_line & 0x3u);  // index in the register [0:3]
    SYSCFG->EXTICR[EXTI_line >> 2u] &= ~(0xfu << (pos << 2));  // clear EXTI port in the register
    SYSCFG->EXTICR[EXTI_line >> 2u] |= (0x7u & port_to_int( port: EXTI_port)) << (pos << 2);  // set EXTI port in the register
    // set triggers
    EXTI->FTSR |= ((falling_edge & 0b1u) << EXTI_line);
    EXTI->RTSR |= ((rising_edge & 0b1u) << EXTI_line);
    EXTI->IMR |= (0b1u << EXTI_line);  // unmask interrupt
}
```

The enable_EXTI function allows you to pass:

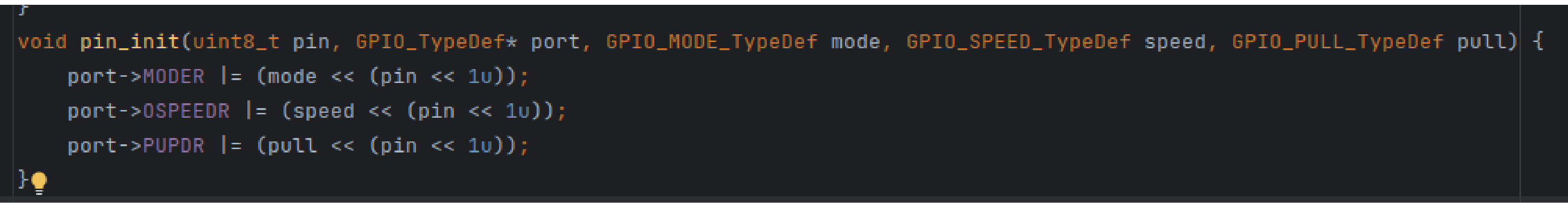
- pin_number (as EXTI_line)
- EXTI_port
- falling/rising_edge (0 or 1)

Firtsly the function filters off all errors (or unsuported values) in the pin number outside the lower 4 bits.

Secondly the bit position is calculated this is determinded only by the lowest two bits because the configuration of all EXTI_lines is split across multiple registers in an array. The index in this array is determined by the remaining two bits.

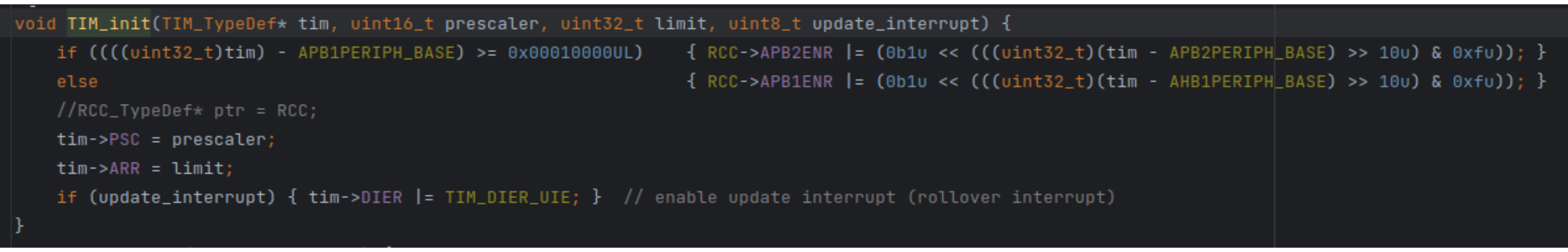
Then the plot (4 bits) in the register is cleared after which the port_number is written to it so that the EXTI peripheral knows what pin to listen to. Note that pos is shifted left by 2 to multiply it by 4 because the plots are 4 bits wide.

Lastly the falling and rising edge configuration is set and the interrupt is unmasked.



```c
void pin_init(uint8_t pin, GPIO_TypeDef* port, GPIO_MODE_TypeDef mode, GPIO_SPEED_TypeDef speed, GPIO_PULL_TypeDef pull) {
    port->MODER |= (mode << (pin << 1u));
    port->OSPEEDR |= (speed << (pin << 1u));
    port->PUPDR |= (pull << (pin << 1u));
}
```

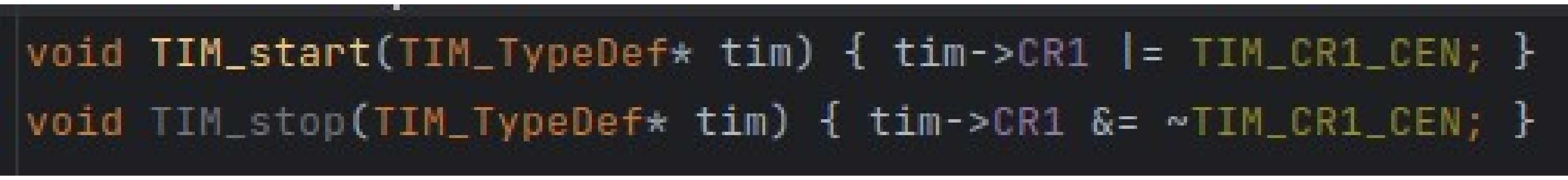
The pin_init function simply writes the configurations to the registers



```c
void TIM_init(TIM_TypeDef* tim, uint16_t prescaler, uint32_t limit, uint8_t update_interrupt) {
    if ((((uint32_t)tim) - APB1PERIPH_BASE) >= 0x00010000UL)    { RCC->APB2ENR |= (0b1u << (((uint32_t)(tim - APB2PERIPH_BASE) >> 10u) & 0xfu)); }
    else                                                        { RCC->APB1ENR |= (0b1u << (((uint32_t)(tim - AHB1PERIPH_BASE) >> 10u) & 0xfu)); }
    //RCC_TypeDef* ptr = RCC;
    tim->PSC = prescaler;
    tim->ARR = limit;
    if (update_interrupt) { tim->DIER |= TIM_DIER_UIE; }  // enable update interrupt (rollover interrupt)
}
```

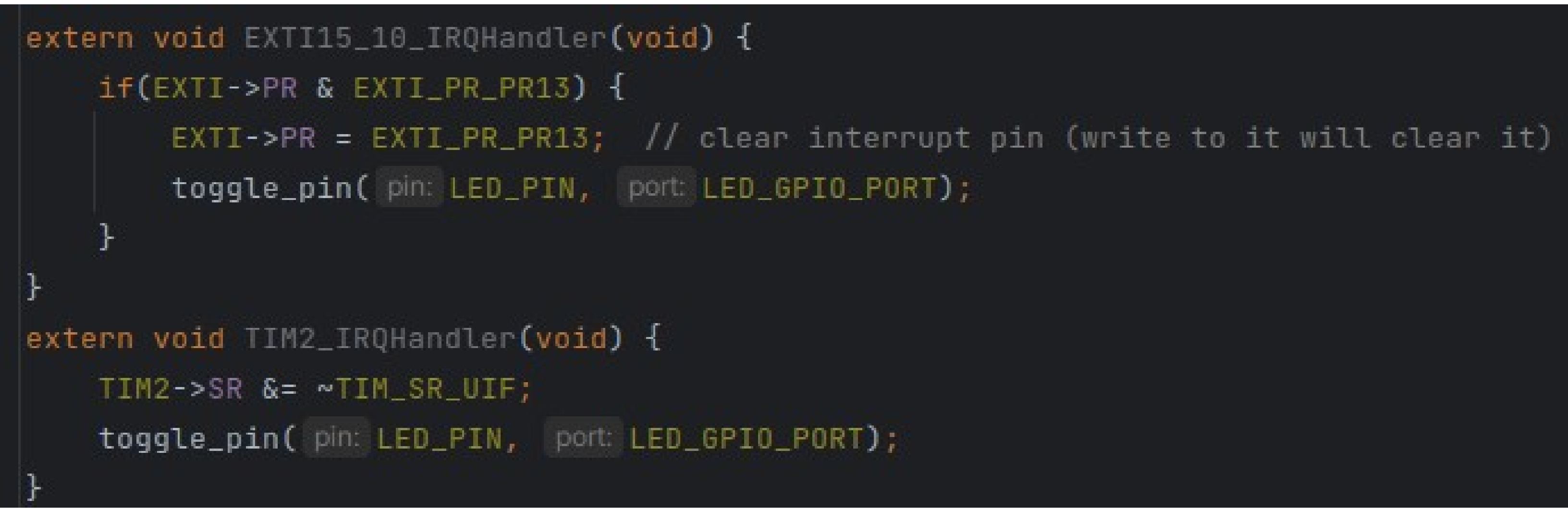
The TIM_init function turns on the correct peripheral clock by looking at the address of the timer.

Then the prescaler, limit and interrupt configuration is set in the timer registers.



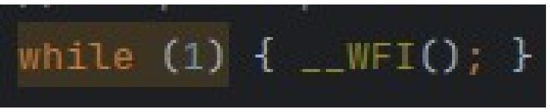
```c
void TIM_start(TIM_TypeDef* tim) { tim->CR1 |= TIM_CR1_CEN; }
void TIM_stop(TIM_TypeDef* tim) { tim->CR1 &= ~TIM_CR1_CEN; }
```

TIM_start/stop simply set/unset the enable bit in CR1 (control register 1)



```c
extern void EXTI15_10_IRQHandler(void) {
    if(EXTI->PR & EXTI_PR_PR13) {
        EXTI->PR = EXTI_PR_PR13;  // clear interrupt pin (write to it will clear it)
        toggle_pin( pin: LED_PIN,  port: LED_GPIO_PORT);
    }
}
extern void TIM2_IRQHandler(void) {
    TIM2->SR &= ~TIM_SR_UIF;
    toggle_pin( pin: LED_PIN,  port: LED_GPIO_PORT);
}
```

In the interrupt handlers the led pin is simply toggled.



```c
while (1) { __WFI(); }
```

In this last bit of code the CPU is forced to suspend until an interrupt occurs (after which it is immediately suspended again)

3. Result

The led is toggled every second or when the button is pressed

// TODO: add video