Networked Embedded Systems

Practicum 2: Timers

Group number: 8

Name	Student ID	Email
Bernberger Sarah	k12112018	sarahbernberger.sb@gmail.com
Grundner Simon	k12136610	simon.grundner@gmail.com

09.05.2024

Map of Content

Theory Questions

a) Describe in your own words Timer Mode, PWM Mode, and Counter Mode. Which own use cases do you come up with?

Timer Mode

A timer can be triggered from various clock sources, such as the Internal Clock (CK_INT), an external input pin (TIx), an external trigger input (ETR) and from internal trigger inputs (ITRx).

Internal trigger inputs can be programmed as a source for a timer and allow the user to use one timer as a prescaler for another timer. For example, timer 2 is configured as a slave for Timer 3, which means that every time the Timer 3 triggers an update event (UEV) Timer 2 receives a pulse.

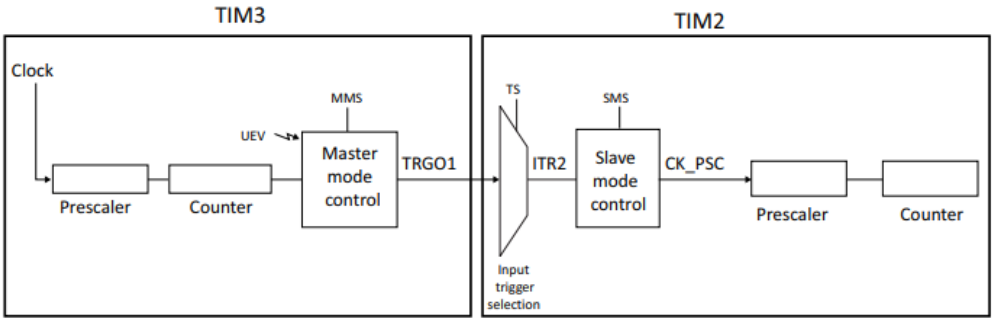

Figure 1: Master/Slave Timer Example

As mentioned, a use case for this operation would be to prescale another timer. This is needed when the user wants to implement a Timer with an extremely long counter period as the default prescaler is only 16-Bit and Timers are up to 32-Bit.

PWM Mode

Pulse width modulation mode of a timer permits to generate a signal with a frequency and a duty cycle. The frequency of the PWM-Signal can be configured with the Auto-Reload Register (TIMx_ARR) and the duty cycle with the TIMx_CRRx register. Calculating the values of these registers is part of Task B.

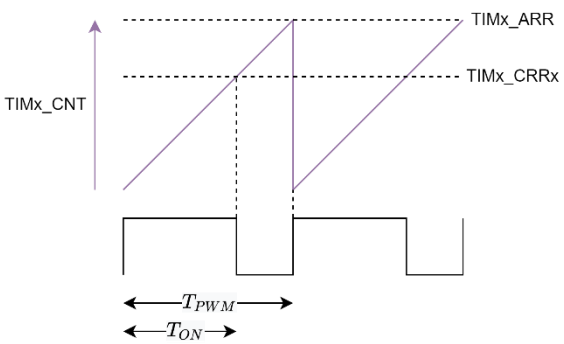

Figure 2: PWM Timing Diagram

Use Cases:

- Dimming an LED.
- Driving a servo motor.
- Control the voltage in switched mode power supplies.

Counter Mode

The counter of a timer is the core of its functionality, as all timer operations are based of measuring discretized timesteps, where the number of elapsed timesteps is contained in the counter register (TIMx_CNT) and counts with a controllable frequency f_{CNT} if the timer is started. There are several modes in which the counter operates: Up-counting Mode, down-counting mode, and center-aligned mode (up/down counting). In this exercise we will only consider the up-counting mode.

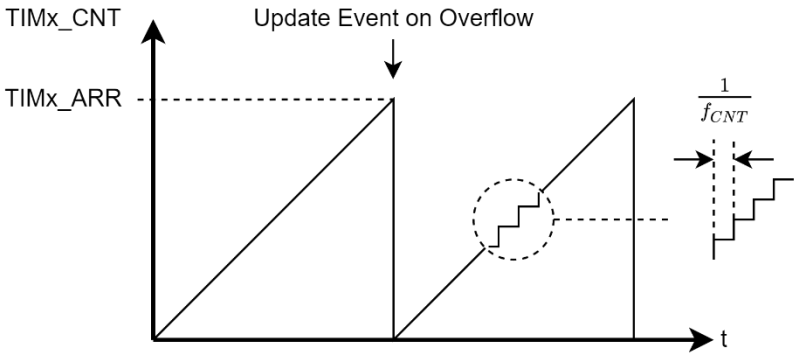

Figure 3: Timer Counter Diagram

A use case would be to repeatedly perform a certain action with a certain period of time.

b) Operation Mode: Timer with counter register

Describe verbally how this mode works. Then discuss the following example based on the diagram:

The counter register (TIMx_CNT) of a timer stores an integer number (16- or 32-Bit depending on the timer) which is incremented (in up counting mode) with the provided clock frequency. The clock frequency which triggers the count operation is previously divided by a prescaler value (TIMx_PSC) to slow down the clock speed according to the use case. When the counter reaches the auto-reload value (content of the TIMx_ARR register, by default 16- or 32 Bit), it restarts from 0 and generates a counter overflow event. The Update Interrupt Flag (UIF) which is set on an overflow event can then be read from the timer status register (TIMx_SR) and must be reset via software.

Figure 4: Timer with Counter Register Example

Clock frequency	Divider (Prescaler)	16-bit Counter Register at t = 0s:
10 kHz	250	65

When will the overflow occur?

Count period: $\frac{1}{10kHz} \cdot 250 = 25ms \rightarrow$ The counter is incremented every 25ms.

The time it takes to count from 65 to the maximum 16-Bit value can be calculated as follows:

$0{,}025s \cdot (2^{16} - 1 - 65) = 1636{,}75s$

What happens after this overflow?

As the counter register overflows (reaches its maximum value) an update event is triggered, setting the Update Interrupt flag high and resets the counter to 0.

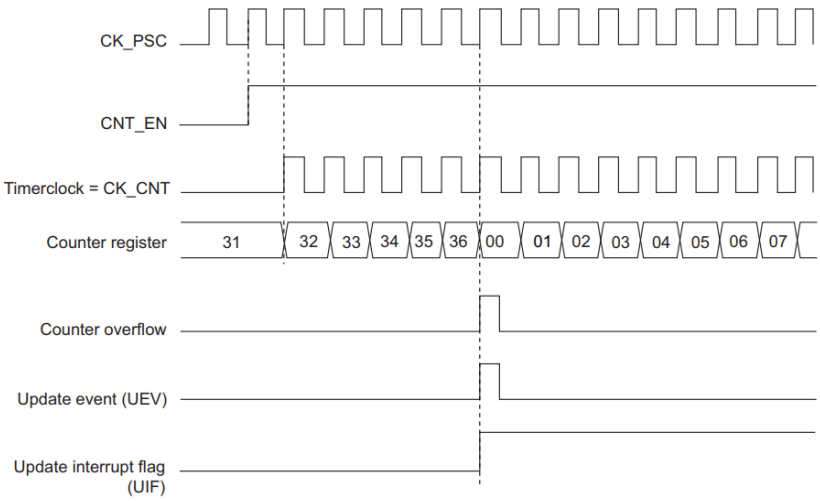

Figure 5: Counter timing diagram with PSC = 1 and ARR = 0x36

c) Operation Mode: Timer with capture event

Describe verbally how this mode works. Then discuss the following example based on the diagram:

A general purpose timer contains 4 capture and compare channels. In input capture mode the capture/compare registers (TIMx_CCRx) are used to latch the value of the counter whenever a special input capture event occurs. A capture action can be performed by checking the Capture/Compare Interrupt Flag (CCxIF) in the timer status register (TIMx_SR). CCxIF must then be cleared by software.

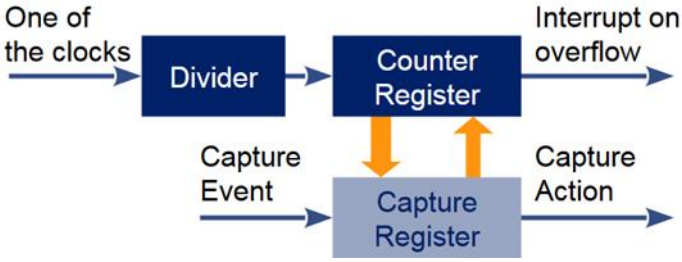

Figure 6: Timer with Capture Event Example

Clock frequency:	Divider:	Capture Event:
200 MHz	500 000	After 2s

Capture the state of all registers at t = 2s when the capture event happens. What is the state of these registers?

With a counter frequency of $f_{CNT} = \frac{f_{CLK}}{PSC} = \frac{200MHz}{500\ 000} = 400Hz$, the captured value of the counter register is $TIMx\_CCRx = \frac{400Hz}{2s} = 200$.

How can the board calculate the time span from this register - as it doesn't know that 2 s have passed?

Since the Timer Module knows at which frequency it is running, we can use backwards calculations to obtain the exact time duration.

d) Operation Mode: Timer with compare register:

Describe verbally how this mode works. Then discuss the following example based on the diagram:

The Input is captured by a catpure event in the same way as in the previous example. This time however, a second channel compares its compare register (TIMx_CCRx) against the capture register from the first channel, which – if matches – triggers a compare action. This can be again checked with the CCxIF in the TIMx_SR. Note that the capture register and the compare register are both called TIMx_CCRx and the Interrupt Flag CCxIF but from different channels.

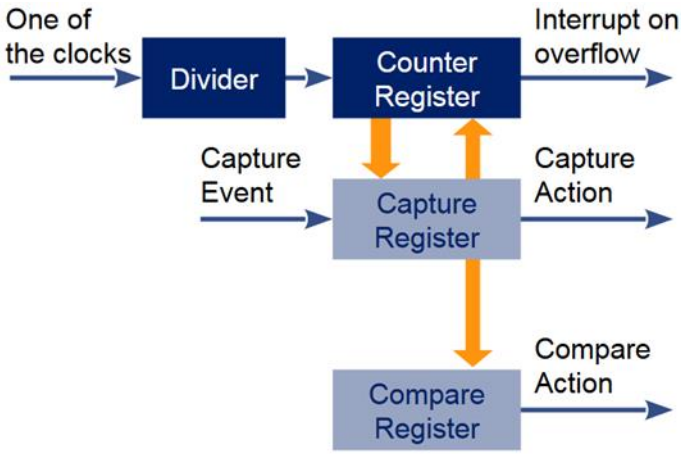

Figure 7: Timer with Compare Register

Clock frequency	Divider	Counter	Capture Register	Compare Register
20 kHz	10 000	0	0	10

$$f_{CNT} = \frac{f_{CLK}}{PSC} = \frac{20\,000Hz}{10\,000} = 2Hz$$

Specify all steps until the compare action is triggered, i.e., counter register, capture register and compare register - and relevant time stamps when any of these registers is updated.

Time	Counter	Capture	Compare	Actions
0 us	0	0	10	none
1 s	2	0	10	none
2,5 s	5	5	10	Capture Action
3,5 s	7	5	10	none
5 s	10	10	10	Capture Action Compare Action
7 s	14	10	10	none
…	…	…	10	…

Table 1: Example Timestamps with Combined Capture and Compare Channels

Task A: Blink an LED with Timer Module

In this task, you are required to toggle the LED on the Elite-Board periodically using a timer. The clock frequency of the timer module should be configured according to the requirements given below.

Requirements:

- ✓ Use **PJ7 (SEGDP)** as digital output (LED)
- ✓ **Timer 2** module should be used for blinking the LED
- ✓ Tout should be **500 ms**.

A.1. Calculations

To achieve a custom frequency f_{CNT}, the frequency from the clock source f_{TIM} can be divided with the prescaler value:

$$f_{CNT} = \frac{f_{TIM}}{PSC}$$

The goal is to trigger an update event every time the Auto Reload Register overflows within a specified amount of time. In this case the time is **500ms.** To calculate the time it takes the counter register to count from 0 to the ARR value, we simply multiply the ARR value with the count interval:

$$T_{OFL} = \frac{1}{f_{CNT}} \cdot ARR = \frac{PSC}{f_{TIM}} \cdot ARR$$

Now, a fitting combination of ARR and PSC must be chosen. Since Timer 2 is a 32-Bit timer, the max value for the ARR is high enough to use the base frequency and PSC is chosen to be 1. The value for the ARR can therefore easily be calculated:

$$ARR = T_{OFL} \cdot \frac{f_{TIM}}{PSC} = 0.5s \cdot \frac{75\ 000\ 000Hz}{1} = 37\ 500\ 000 \qquad \textit{formula 1}$$

The value for f_{TIM} can be set in the Clock configuration in Cube MX and is $75\ MHz$ by default.

Final Values	
PSC	1
ARR	37 500 000

Table 2: Task A - Final Values from Calculations

A.2. Implementation

Configuration in Cube MX

Before discussing the code implementation, timer 2 needs to be configured in Cube MX. The clock source has been set as the internal clock and the values for the PSC and ARR are adopted from the calculations above.

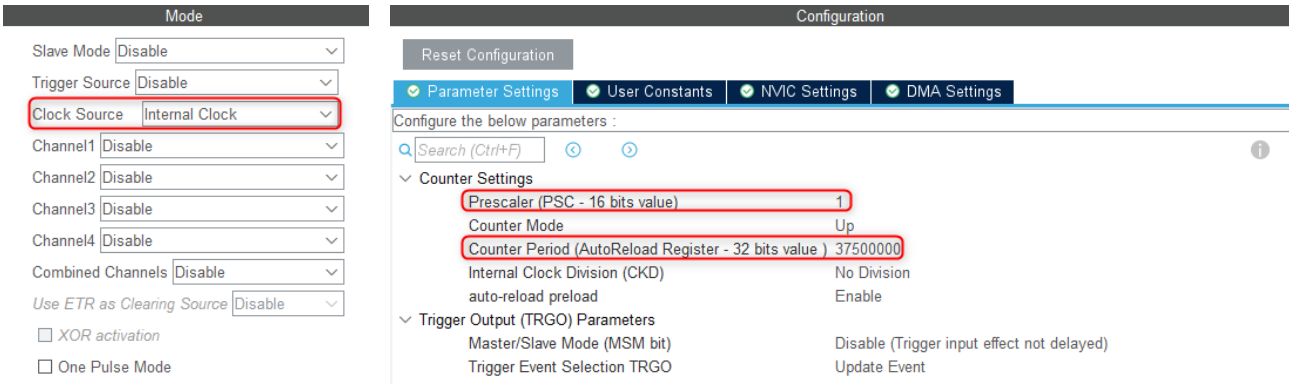

Figure 8: Task A - TIM2 Configuration in Cube MX

Code:

The timer can be started with the HAL provided function. The status register of the timer 2 is constantly checked if an update event has been triggered. The check can be performed by inspecting the status of the Update Interrupt Flag.

Timer 2 Status Register (`TIM2->SR`)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	CC4OF	CC3OF	CC2OF	CC1OF	Res.	Res.	TIF	Res.	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Figure 9: Timer Status Register

`TIM_SR_UIF = 0x0001` (Bitmask for the Flag)

	Flag is Set	Flag is Reset	Operation
`TIM2->SR`	`0b xxxx … xxx1`	`0b xxxx … xxx0`	&
`TIM_SR_UIF`	`0b 0000 … 0001`	`0b 0000 … 0001`	
check-result	1	0	=

Table 3: Check if a Bit is Set in Register

Because the UIF is not reset automatically, it must be manually reset in the software:

`TIM2->SR`	`0b xxxx … xxx1`	&
`~TIM_SR_UIF`	`0b 1111 … 1110`	
new `TIM2->SR` value	`0b xxxx … xxx0`	=

Table 4: Clear Bit in Register

Of course, the decimal point LED od the segment is also toggled.

```
int main(void)
{
  HAL_TIM_Base_Start(&htim2);
  while (1)
  {
    if (TIM2->SR & TIM_SR_UIF) // check update interrupt flag
    {
      TIM2->SR &= ~TIM_SR_UIF; // clear update interrupt flag
      HAL_GPIO_TogglePin(SEGDP_GPIO_Port, SEGDP_Pin);
    }
  }
}
```

Code Segment 1: Task A - Relevant Code in main.c

A.3. Discussion

Because this code is relatively simple, no functions or code outside of the main function is implemented. Still, the code is very easy to understand due to HAL provided macros and functions from the GPIO and TIM module.

The difference to practicum 1, where the `HAL_Delay()` function was used to achieve the blinking period, is that the timer is not part of the code but a separate unit which counts independently. This is favorable because it allows other functions (such as button press-checks) to be executed without being blocked by any delay.

Task B: Vary the Brightness of an LED

The user should be able to control the brightness of the LED by using two push buttons provided on the board. These buttons are connected to the pins PJ12 and PJ13 and labelled as BTN1 and BTN2 respectively. BTN1 button should be used to increase the brightness of the LED and BTN2 is used to decrease the brightness. The range of brightness should be between 0% and 100% and an individual step should be 10%. That means, increasing the brightness by a single step indicates a 10% increase in brightness and vice versa. The brightness should be controlled by adjusting the PWM signal which can be generated using a timer module. The configuration of the timer module and the PWM signal should be done according to the below requirements.

Requirements:

- ✓ Use **PJ7 (SEGDP)** as digital output.
- ✓ Use **PJ12 (BTN1)** and **PJ13 (BTN2)** for digital inputs.
- ✓ Frequency of the PWM signal should be **500 Hz**.
- ✓ Use **channel 2** of **TIM8** for generating PWM signal (**PJ6**)
- ✓ Max brightness = **100%**, Min brightness = **0%**, Step size = **10%**

B.1. Calculations

To vary the brightness of the LED, a PWM signal is used to control the average power delivered to the LED. The PWM mode of a timer works as depicted in Figure 2.

Calculate the frequency of the PWM-Signal:

First, the f_{TIM} frequency is divided by the prescaler PSC. The prescaled timer frequency then counts to the auto reload registers value. The formula of frequency for the whole cycle time is hereby:

$$f_{PWM} = \frac{f_{TIM}}{PSC \cdot ARR}$$

With the specification for $f_{PWM} = 500Hz$ and $f_{TIM} = 75MHz$ (defined in the Clock Configuration tab in CubeMX as APB Timer Clock), an arbitrary combination of the prescaler value PSC and the Auto-Reload-Register value can be chosen. But to achieve a higher resolution for the duty cycle, the 16-Bit ARR value should be maximized. Therefor the lowest possible prescaler value was chosen which is $PSC = 3$. Since the frequencies are given and we can freely choose the prescaler, the formula can be converted to ARR.

$$ARR = \frac{f_{TIM}}{f_{PWM} \cdot PSC} = \frac{75\,000\,000Hz}{500Hz \cdot 3} = 50\,000$$

Calculate the duty cycle:

The duty cycle is the *on*- to *cycle* time ratio. Since CCR spans over the *on* period of the counter and ARR over the whole period, the duty cycle can be calculated as follows:

$$DC = \frac{CCR}{ARR}$$

To calculate the duty cycle from 0 to 100 with a step size of 10%, set the capture/compare register to

$$CCR = 0.1 \cdot n \cdot ARR \qquad \textit{formula 2}$$

where the integer number n is in the interval of $[0,10]$. This register is updated on every button push.

Final Values	
PSC	3
ARR	50 000

Table 5: Task B - Final Values from Calculations

B.2. Implementation

Configuration in Cube MX:

Conveniently, the *GPIO PJ6* which is wired to segment G is also internally connected to channel 2 of Timer 8. The first thing to be done, is to select it in the pinout of the microcontroller as shown in Figure 10. After selecting PWM-Generation (non-inverted with output) for channel 2, the values for the prescaler and auto reload register can be inserted from the calculations.

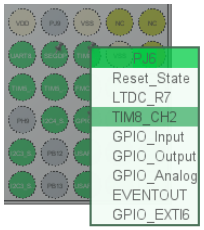

Figure 10: PJ6 as TIM8 CH2 in Pinout

The *Pulse* value, which determines the duty cycle in the *PWM Generation Channel 2* configuration, can be initially set to 0, because it will be dynamically adjusted to the state of the program.

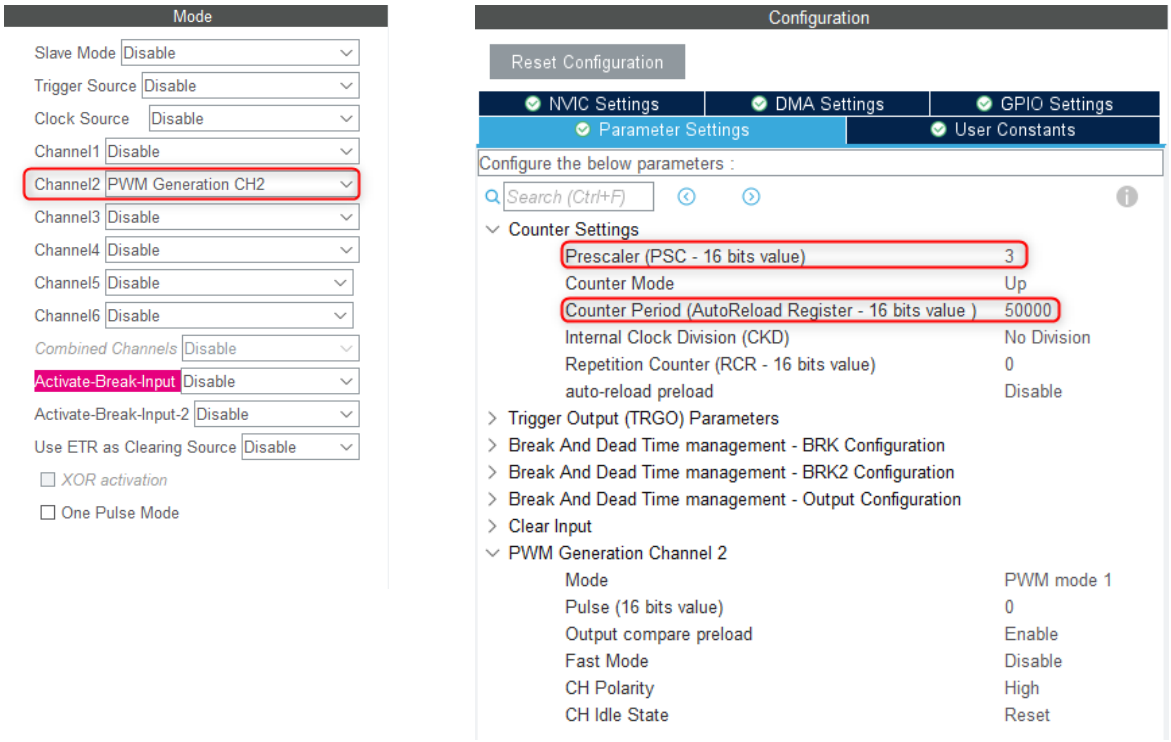

Figure 11: Task B - Config for TIM 8

Code:

For this task, the job of the main function is to firstly initialize the two buttons used to step up and down with the duty cycle, as well as starting the PWM Output of Timer 8 channel 2. An integer variable is used to keep track of the current step.

The functions for the button module are reused from practicum 1, where they are also documented in detail. Therefore, there won't be any code snippets in this document but are still available in the appended code.

In the infinite while loop, both buttons are checked for updates. With the results of these checks a sign `sgn` variable is computed to determine the direction of change. The conditions for this variable are listed in following table:

step_up.has_changed	step_dn.has_changed	sgn (step_up.has_changed - step_dn.has_changed)
0	0	0
0	1	-1
1	0	1
1	1	0

Table 6: Conditions for sgn variable

This variable can then be directly added to the current step (`curr_step`) and checked if it exceeds the interval from `MIN_STEP` (0) to `MAX_STEP` (10). In case it does, the value is capped to either `MIN_VALUE` or `MAX_VALUE`. Finally, the Capture/Compare Register which determines the pulse width can be updated with the new value derived from formula 2.

```
int main(void)
{
  Button_t step_up;
  Button_t step_dn;
  USR_BTN_Init(&step_up, BTN1_GPIO_Port, BTN1_Pin);
  USR_BTN_Init(&step_dn, BTN2_GPIO_Port, BTN2_Pin);

  uint8_t curr_step = 0;
  HAL_TIM_PWM_Start(&htim8, TIM_CHANNEL_2);

  while (1)
  {
    USR_BTN_Check(&step_up);
    USR_BTN_Check(&step_dn);

    int8_t sgn = step_up.has_changed - step_dn.has_changed;
    if (sgn == 0) continue; // none or both buttons pressed

    curr_step = LIMIT(curr_step + sgn, MIN_STEP, MAX_STEP);
    TIM8->CCR2 = (uint16_t)(0.1f * TIM8->ARR * curr_step);
  }
}
```

Code Segment 2: Task B - Relevant Code in main.c

B.3. Results

Following Figures show results of the PWM controlled G-Segment of the display.

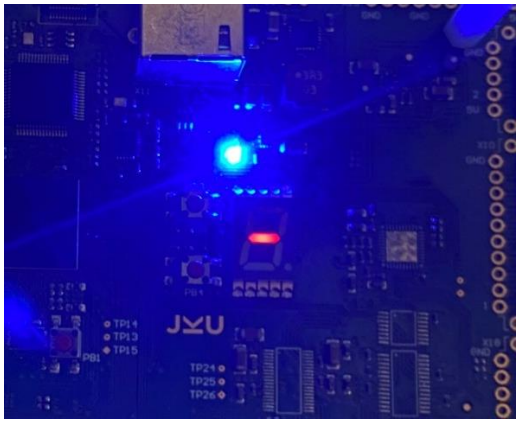

Figure 12: Task B Result - Low Intensity (10%)

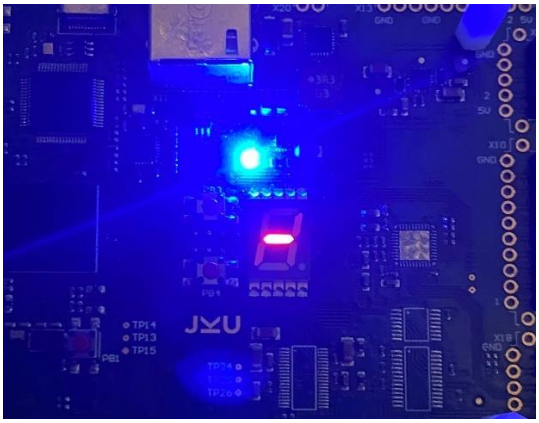

Figure 13: Task B Result - High Intensity (100%)

B.4. Discussion

To emphasize previous feedback regarding the use of macros:

The reason a macro instead of a function was implemented in the last practicum is because the functionality of this macro was needed for different data types, the issues and potential problems of macro implementations were also discussed in the protocol. The macro used:

```
#define LIMIT(x, min, max) ((x) < (min) ? (min) : ((x) > (max) ? (max) : (x)))
```

Code Segment 3: LIMIT Macro

For example, x = LIMIT(x, max, min) can be interpreted as

```
if (x < min) x = min;
else if (x > max) x = max;
else x = x;
```

for better readability. Since exactly this functionality was needed multiple times in the code, we decided to create a macro for this operation. Even though this implementation works perfectly fine for this case and will still be used in the code, a more secure version could be implemented as follows:

```
#define LIMIT(x, min, max) \
  ({ \
    typeof(x) _x = (x); \
    typeof(min) _min = (min); \
    typeof(max) _max = (max); \
    ((_x < _min) ? _min : ((_x > _max) ? _max : _x)); \
  })
```

Code Segment 4: Secure LIMIT Macro

Implementing it as demonstrated ensures that each input parameter is only used once, and its value is copied to local parameters instead.

Task C: Count Pulses of a Blinking LED

In this task, you are required to generate a pulse counter with Timer 3. A clock (Tout = 1 sec approx.) signal shall be generated from Timer 2, which shall be used as an internal trigger for Timer 3. Timer 3 shall count repeatedly from 0 to 9. The current count of the Timer 3 should be displayed on the Seven Segment Display of the Eliteboard.

Requirements:

- ✓ Use the **7-segment display** for digital output.
- ✓ **Timer 2** should provide an internal trigger; $\mathbf{T_{out}}$ should be **1s**.
- ✓ **Timer 3** should be used to count (**0-9**) pulses.

C.1. Implementation

In this task, two timers have to be configured in such a way, that they are connected as shown in Figure 14. This is a case where a master timer acts as a prescaler for a slave timer.

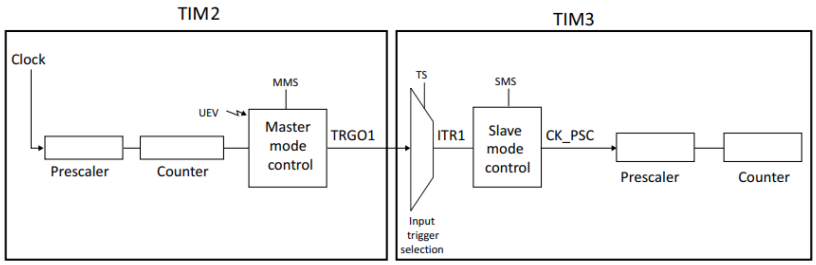

Figure 14: Task C - Timer Interconnection Diagram

Configuration in Cube MX:

Master Timer 2 Configuration:

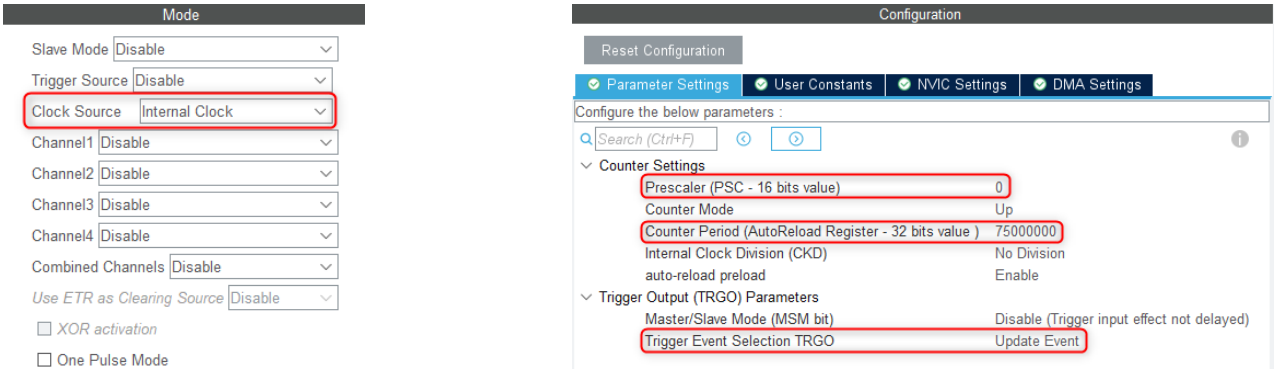

Figure 15: Task C - Master Timer Configuration

The ARR value is calculated in the same way as in formula 1 but with $T_{OFL} = 1s$. A prescaler of 0 means, that no prescaler is used and is equal to a prescaler of 1. Since the Trigger output (TRGO) causes the internal trigger to fire, we want it to respond to an overflow update event.

Slave Timer 3 Configuration:

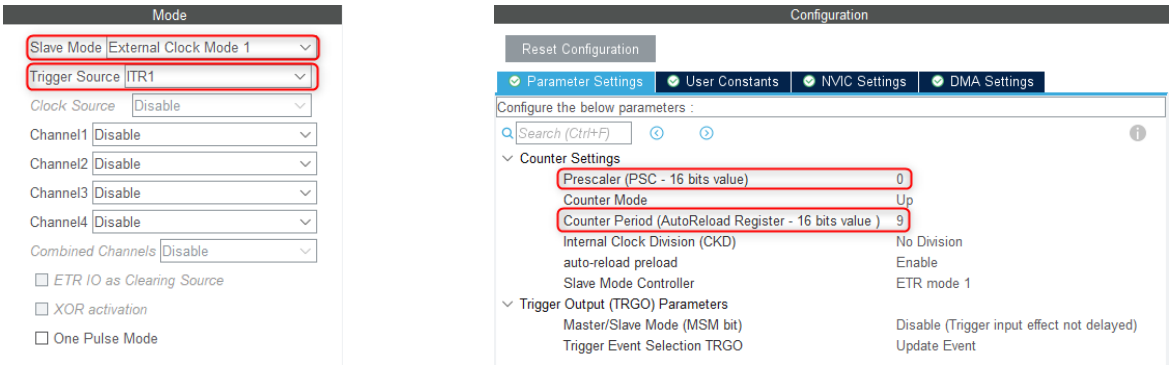

Figure 16: Task C - Slave Timer Configuration

Timer 3 is configured in slave mode. Instead of the internal clock, external clock mode 1 is now selected because as depicted in Figure 17 from the datasheet [RM0433, p. 1646], the Internal Trigger (ITRx) is connected to this clock mode.

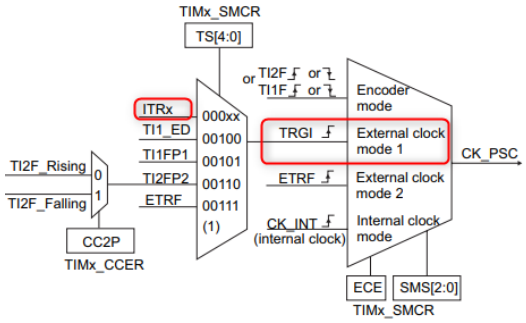

Figure 17: External Clock Connection Diagram

To know which internal trigger is connected to which master timer TRGO, Table 7 can be found in the datasheet [RM0433, p. 1682]. With timer 3 as the slave timer and timer 2 as the master timer, the internal trigger is ITR1.

Slave TIM	ITR0	ITR1	ITR2	ITR3	ITR4	ITR5	ITR6	ITR7	ITR8
TIM2	TIM1	TIM8	TIM3	TIM4	ETH PPS	USB1 OTG_ HS_ SOF	USB2 OTG_ FS_ SOF	-	-
TIM3	TIM1	TIM2	TIM15	TIM4	ETH PPS	-	-	-	-
TIM4	TIM1	TIM2	TIM3	TIM8	-	-	-	-	-
TIM5	TIM1	TIM8	TIM3	TIM4	-	-	fdcan1_ soc	USB1 OTG_ HS_ SOF	USB2 OTG_ FS_ SOF

Table 7: TIMx Internal Trigger Connection

Finally, since we want the slave timer to count from 0 to 9, the ARR is set correspondingly to 9.

Code:

In the main method for this task, the previously configured timers are started, and a segment display is instantiated. It is checked in the infinite while loop, if the counter register changed its value compared to the currently displayed number. If so, the segment display is directly written with the counter register value of the slave timer 3, where `seg.curr_num` is updated as well.

```
int main(void)
{
  HAL_TIM_Base_Start(&htim2); // Start Master Timer
  HAL_TIM_Base_Start(&htim3); // Start Slave Timer

  Segment_t seg;
  USR_SEG_Init(&seg);

  while (1)
  {
    if (seg.curr_num != TIM3->CNT)
    {
      USR_SEG_Write(&seg, TIM3->CNT);
    }
  }
}
```

Code Segment 5: Task C - Relevant Code in main.c

The write function for the segment display limits the value to the set values (0 and 9) and writes the number to the Output Data Register. `MODIFY_REG` is a macro provided by the Hardware Abstraction Layer Library.

```
void USR_SEG_Write(Segment_t* seg, int num)
{
  seg->curr_num = LIMIT(num, seg->min_num, seg->max_num);
  MODIFY_REG(
      seg->GPIOx->ODR,
      seg->pin_mask,
      seg->segments[seg->curr_num]);
}
```

Code Segment 6: Segment Write Function

- `seg->GPIOx` is the port the segment display is connected to (Port J)
- `seg->pin_mask` is the or-ation of every SEGx_Pin (x … A-G)
- `seg->segments[]` is an array containing the pin mask of each number where the index is that corresponding number (0-9).

C.2. Discussion

We noticed that the difficulty in this task was not the actual coding but finding relevant information in the datasheet. Additionally, the ability to reuse a lot of functions from practicum 1 made it easier to focus on the timer implementation, which was very convenient.

Index

CCxIF.....*Capture/Compare x Interrupt Flag (x = channel)*
CK_INT.....*Internal Clock*
ETR.....*External Trigger*
GPIO.....*General Purpose Input/Output*
HAL.....*Hardware Abstraction Layer*
ITRx.....*Internal Trigger (Nr x)*
LED.....*Light Emitting Diode*
TIM.....*Timer*
TIMx_ARR.....*Timer Auto-Reload Register*
TIMx_CNT.....*Timer Counter Register*
TIMx_CRRx.....*Capture Compare Register*
TIMx_PSC.....*Prescaler Register*
TIMx_SR.....*Timer Status Register*
TIx.....*External Input Pin (Nr x)*
TRGO.....*Trigger Output*
UEV.....*Update Event*
UIF.....*Update Interrupt Flag*

Figures

Code Segments

Tables