

**Zuzanna Godek 318373**

**MARM sprawozdanie z laboratorium 2**

Celem laboratorium było zapoznanie się z układami czasowo-licznikowymi w mikrokontrolerach STM32.

**Zadanie 1.1.**

Celem zadania było napisanie programu, który tworzy „falę” z diod, które się przełączają jedna po drugiej co 200ms.

Na początku skonfigurowano układ licznika T1, żeby zgłaszał przerwanie co 200ms:

```
auto timer_config() -> void {
    LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_TIM1);
    LL_RCC_ClocksTypeDef rcc_clocks;
    LL_RCC_GetSystemClocksFreq(&rcc_clocks);

    uint32_t timer_clock = rcc_clocks.PCLK2_Frequency;

    if (LL_RCC_GetAPB2Prescaler() != LL_RCC_APB2_DIV_1) {
        timer_clock *= 2;
    }

    const uint32_t target_freq = 10000;
    uint32_t prescaler_value = (timer_clock / target_freq) - 1;

    LL_TIM_SetPrescaler(TIM1, prescaler_value);
    LL_TIM_SetAutoReload(TIM1, 2000 - 1);
```

a także włączono obsługę przerwania od timera T1 w kontrolerze NVIC oraz w samym liczniku:

```
NVIC_SetPriority(TIM1_UP_TIM10_IRQn, 0);
NVIC_EnableIRQ(TIM1_UP_TIM10_IRQn);

LL_TIM_EnableCounter(TIM1);
LL_TIM_EnableIT_UPDATE(TIM1);
LL_TIM_GenerateEvent_UPDATE(TIM1);
}
```

Potem skonfigurowano porty GPIO w kierunku wyjścia wraz z szybkością narastania zbocza ustawioną na *low* (z racji, że częstotliwość przełączania była niska):

```
auto output_config() -> void {
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD);
```

```

LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_12, LL_GPIO_MODE_OUTPUT);
LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_13, LL_GPIO_MODE_OUTPUT);
LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_14, LL_GPIO_MODE_OUTPUT);
LL_GPIO_SetPinMode(GPIOB, LL_GPIO_PIN_15, LL_GPIO_MODE_OUTPUT);

LL_GPIO_SetPinSpeed (GPIOB, LL_GPIO_PIN_12, LL_GPIO_SPEED_FREQ_LOW);
LL_GPIO_SetPinSpeed (GPIOB, LL_GPIO_PIN_13, LL_GPIO_SPEED_FREQ_LOW);
LL_GPIO_SetPinSpeed (GPIOB, LL_GPIO_PIN_14, LL_GPIO_SPEED_FREQ_LOW);
LL_GPIO_SetPinSpeed (GPIOB, LL_GPIO_PIN_15, LL_GPIO_SPEED_FREQ_LOW);
}

```

Uzupełniono także obsługę przerwania w funkcji *tim1\_up\_tim10\_isr\_vector* (znaleziono jej nazwę w pliku nagłówkowym *f76x.h*). W tej funkcji zrealizowano logikę przełączania diod (w przerwaniu najpierw wyłączano wszystkie diody po czym włączano diodę wyznaczoną przez wartość zmiennej globalnej *step*):

```

extern "C" {

void tim1_up_tim10_isr_vector() {
    if (LL_TIM_IsActiveFlag_UPDATE(TIM1)) {
        LL_TIM_ClearFlag_UPDATE(TIM1);
        LL_GPIO_ResetOutputPin(GPIOB, LL_GPIO_PIN_12 |
LL_GPIO_PIN_13 | LL_GPIO_PIN_14 | LL_GPIO_PIN_15);
        switch(step) {
            case 0:
                LL_GPIO_SetOutputPin(GPIOB, LL_GPIO_PIN_12);
                break;
            case 1:
                LL_GPIO_SetOutputPin(GPIOB, LL_GPIO_PIN_13);
                break;
            case 2:
                LL_GPIO_SetOutputPin(GPIOB, LL_GPIO_PIN_14);
                break;
            case 3:
                LL_GPIO_SetOutputPin(GPIOB, LL_GPIO_PIN_15);
                break;
        }
        step++;
        if (step > 3) {
            step = 0;
        }
    }
}
}

```

Wykorzystywano zmienną globalną *step* do przechowywania ostatniego stanu diod:

```
namespace {
    volatile uint8_t step = 0;
}
```

Po uruchomieniu programu zweryfikowano, że diody migły po sobie poprawnie.

### Zadanie 1.2.

Kolejnym zadaniem było napisanie programu, który wykorzystując układ licznikowy T4 wygeneruje sygnał prostokątny o wypełnieniu 50% z częstotliwością 1 Hz. Następnie na diodzie LD4 miał być ten sygnał o przesunięciu fazowym 0°, na diodzie LD3 180°, a na diodzie LD5 270° (założono, że dioda LD6 ma być wyłączona do tego zadania).

Zdecydowano się wykorzystać tryb *toggle* dla 3 kanałów T4, odpowiadających odpowiednim diodom. Częstotliwość licznika ustawiono na 2 kHz, ponieważ ułatwiało to ustalenie przesunięć fazowych. Rejestr ARR ustawiono na 1000, żeby częstotliwość migania diod wynosiła 1 Hz zgodnie z polecением (diody zmieniały stan co 500 ms). Przesunięcie fazowe osiągnięto poprzez odpowiednie ustawianie rejestru CC, ale należało także wykorzystać różną polaryzację kanałów, żeby osiągnąć przesunięcie fazowe o 270°. Kanał timera odpowiadający za miganie diody zielonej (LD4) miał polarity zapisane na *high*, czyli domyślne. W rejestrze CC zapisano wartość 0, czyli dioda zmieniała stan bez przesunięcia w fazie. Polaryzację pozostałych dwóch kanałów ustawiono na *low*. Umożliwiło to odwrócenie działania diody pomarańczowej (LD3) względem zielonej przy takim samej wartości rejestrów CC (równiej 0). Natomiast w przypadku diody czerwonej(LD5) odwrócona polaryzacja była konieczna. Powodem było to, że zmianę stanu można wykonać tylko dla wartości licznika mniejszej niż 1000, ponieważ taką wartość posiada rejestr ARR. Ale wiemy że przesunięcie fazowe ma być 270°, czyli dioda ma się dopiero zapalić w trzeciej ćwiartce okresu kiedy zielona już się wyłączyla, a ma się wyłączyć w pierwszej ćwiartce okresu kiedy dioda zielona jest włączona. W związku z tym musimy ustalić zmianę stanu w pierwszej połowie okresu, ale z odwróconą polaryzacją. Czyli kiedy dioda zielona jest włączona i zostanie zliczone 500 tyknięć (pierwsza ćwiartka okresu) to dioda czerwona jest wyłączana. Natomiast po kolejnym odliczeniu do 500 ma być włączona (czyli posiada przesunięcie w fazie o 270°).

```
auto timer_config() -> void {
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM4);
    LL_RCC_ClocksTypeDef rcc_clocks;
    LL_RCC_GetSystemClocksFreq(&rcc_clocks);

    uint32_t timer_clock = rcc_clocks.PCLK1_Frequency;

    if (LL_RCC_GetAPB1Prescaler() != LL_RCC_APB2_DIV_1) {
        timer_clock *= 2;
    }
}
```

```

const uint32_t target_freq = 2000;
uint32_t prescaler_value = (timer_clock / target_freq) - 1;

LL_TIM_SetPrescaler(TIM4, prescaler_value);
LL_TIM_SetAutoReload(TIM4, 1000 - 1);

LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_TOGGLE);
LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_TOGGLE);
LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_TOGGLE);

LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH1);
LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH2);
LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH3);

LL_TIM_OC_SetCompareCH1(TIM4, 0);
LL_TIM_OC_SetPolarity(TIM4, LL_TIM_CHANNEL_CH1,
LL_TIM_OCPOLARITY_HIGH);
LL_TIM_OC_SetCompareCH2(TIM4, 0);
LL_TIM_OC_SetPolarity(TIM4, LL_TIM_CHANNEL_CH2,
LL_TIM_OCPOLARITY_LOW);
LL_TIM_OC_SetCompareCH3(TIM4, 500);
LL_TIM_OC_SetPolarity(TIM4, LL_TIM_CHANNEL_CH3,
LL_TIM_OCPOLARITY_LOW);

LL_TIM_EnableCounter(TIM4);
LL_TIM_GenerateEvent_UPDATE(TIM4);
}

```

Piny GPIO należało skonfigurować w trybie *alternate* oraz połączyć z sygnałem timera 4 ustawiając AFPin\_8\_15 na LL\_GPIO\_AF\_2:

```

auto output_config() -> void {
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD);

    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_12, LL_GPIO_MODE_ALTERNATE);
    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_13, LL_GPIO_MODE_ALTERNATE);
    LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_14, LL_GPIO_MODE_ALTERNATE);

    LL_GPIO_SetAFPin_8_15(GPIOD, LL_GPIO_PIN_12, LL_GPIO_AF_2);
    LL_GPIO_SetAFPin_8_15(GPIOD, LL_GPIO_PIN_13, LL_GPIO_AF_2);
    LL_GPIO_SetAFPin_8_15(GPIOD, LL_GPIO_PIN_14, LL_GPIO_AF_2);

    LL_GPIO_SetPinSpeed (GPIOD, LL_GPIO_PIN_12, LL_GPIO_SPEED_FREQ_LOW);
    LL_GPIO_SetPinSpeed (GPIOD, LL_GPIO_PIN_13, LL_GPIO_SPEED_FREQ_LOW);

```

```
    LL_GPIO_SetPinSpeed (GPIOB, LL_GPIO_PIN_14, LL_GPIO_SPEED_FREQ_LOW) ;  
}
```

Obserwacja migania diod potwierdziła poprawność programu, bo dioda zielona i pomarańczowa były włączane naprzemiennie, natomiast czerwona była włączana kiedy pomarańczowa się jeszcze świeciła i wyłączała się po tym jak zielona się zaświeciła.

### Zadanie 1.3.

Przedmiotem tego zadania było generowanie sygnałów PWM przy wykorzystaniu T4 na wejściu diod. Dioda LD4 miała się świecić z wypełnieniem 20%, dioda LD3 40%, a dioda LD5 60%. Natomiast wypełnienie o wartości zmiennej (od 0 do 100% z krokiem 10%) miało być na diodzie LD6.

Timer skonfigurowano w trybie PWM1 na 4 kanałach. Każdemu kanałowi ustawiono inną wartość CC, odpowiednią dla zadanaego wypełnienia.

```
auto timer_config() -> void {  
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM4);  
    LL_RCC_ClocksTypeDef rcc_clocks;  
    LL_RCC_GetSystemClocksFreq(&rcc_clocks);  
  
    uint32_t timer_clock = rcc_clocks.PCLK1_Frequency;  
  
    if (LL_RCC_GetAPB1Prescaler() != LL_RCC_APB1_DIV_1) {  
        timer_clock *= 2;  
    }  
  
    const uint32_t target_freq = 100*100; //200Hz  
    uint32_t prescaler_value = (timer_clock / target_freq) - 1;  
  
    LL_TIM_SetPrescaler(TIM4, prescaler_value);  
    LL_TIM_SetAutoReload(TIM4, 100 - 1);  
  
    LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);  
    LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH2, LL_TIM_OCMODE_PWM1);  
    LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_PWM1);  
    LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH4, LL_TIM_OCMODE_PWM1);  
  
    LL_TIM_OC_EnablePreload(TIM4, LL_TIM_CHANNEL_CH1);  
    LL_TIM_OC_SetCompareCH1(TIM4, 20);  
    LL_TIM_OC_SetCompareCH2(TIM4, 40);  
    LL_TIM_OC_SetCompareCH3(TIM4, 60);  
    LL_TIM_OC_SetCompareCH4(TIM4, 0);
```

```

    LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH1);
    LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH2);
    LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH3);
    LL_TIM_CC_EnableChannel(TIM4, LL_TIM_CHANNEL_CH4);

    LL_TIM_GenerateEvent_UPDATE(TIM4);
    LL_TIM_ClearFlag_UPDATE(TIM4);

    LL_TIM_EnableCounter(TIM4);
}

```

Należało także skonfigurować przerwania dla przycisku PA0:

```

auto interrupt_config() -> void {
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);

    LL_GPIO_SetPinMode(GPIOA, LL_GPIO_PIN_0, LL_GPIO_MODE_INPUT);

    LL_GPIO_SetPinSpeed(GPIOA, LL_GPIO_PIN_0, LL_GPIO_SPEED_FREQ_LOW);

    LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SYSCFG);

    LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTA, LL_SYSCFG_EXTI_LINE0);

    LL EXTI_EnableIT_0_31(LL_EXTI_LINE_0);
    LL EXTI_EnableRisingTrig_0_31(LL_EXTI_LINE_0);

    NVIC_SetPriority(EXTI0_IRQn, 5);
    NVIC_EnableIRQ(EXTI0_IRQn);
}

```

Zdecydowano się także wykorzystać timer T2 do zniwelowania drgań zestyków. Jego konfiguracja wyglądała następująco:

```

auto debounce_timer_config() -> void {
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM2);

    LL_RCC_ClocksTypeDef rcc_clocks;
    LL_RCC_GetSystemClocksFreq(&rcc_clocks);

    uint32_t timer_clock = rcc_clocks.PCLK1_Frequency;
    if (LL_RCC_GetAPB1Prescaler() != LL_RCC_APB1_DIV_1) {
        timer_clock *= 2;
    }
}

```

```

    uint32_t prescaler = (timer_clock / 10000) - 1;
    LL_TIM_SetPrescaler(TIM2, prescaler);
    LL_TIM_SetAutoReload(TIM2, debounce_ms * 10 - 1);

    LL_TIM_EnableIT_UPDATE(TIM2);

    NVIC_SetPriority(TIM2_IRQn, 6);
    NVIC_EnableIRQ(TIM2_IRQn);
}

```

Samo drganie zestyków zostało wyeliminowane poprzez uruchomianie timera w przerwaniu przycisku i wyłączaniu przerwań do czasu ustania drgań. Wypełnienie było zwiększane przy obsłudze przerwania przycisku.

```

extern "C" {
    //! Exti 0 vector
    void exti0_isr_vector() {
        if(LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_0)) {
            LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_0);

            LL_EXTI_DisableIT_0_31(LL_EXTI_LINE_0);
            if(counter >= 100) {
                counter = 0;
            } else {
                counter += 10;
            }
            LL_TIM_OC_SetCompareCH4(TIM4, counter);
            LL_TIM_SetCounter(TIM2, 0);
            LL_TIM_EnableCounter(TIM2);
        }
    }

    void tim2_isr_vector() {
        if(LL_TIM_IsActiveFlag_UPDATE(TIM2)) {
            LL_TIM_ClearFlag_UPDATE(TIM2);

            static uint8_t stable_low_cnt = 0;

            if(LL_GPIO_IsInputPinSet(GPIOA, LL_GPIO_PIN_0)) {
                stable_low_cnt = 0;
            } else {
                stable_low_cnt++;
            }
        }
    }
}

```

```

        if (stable_low_cnt >= 3) {
            LL_TIM_DisableCounter(TIM2);
            stable_low_cnt = 0;

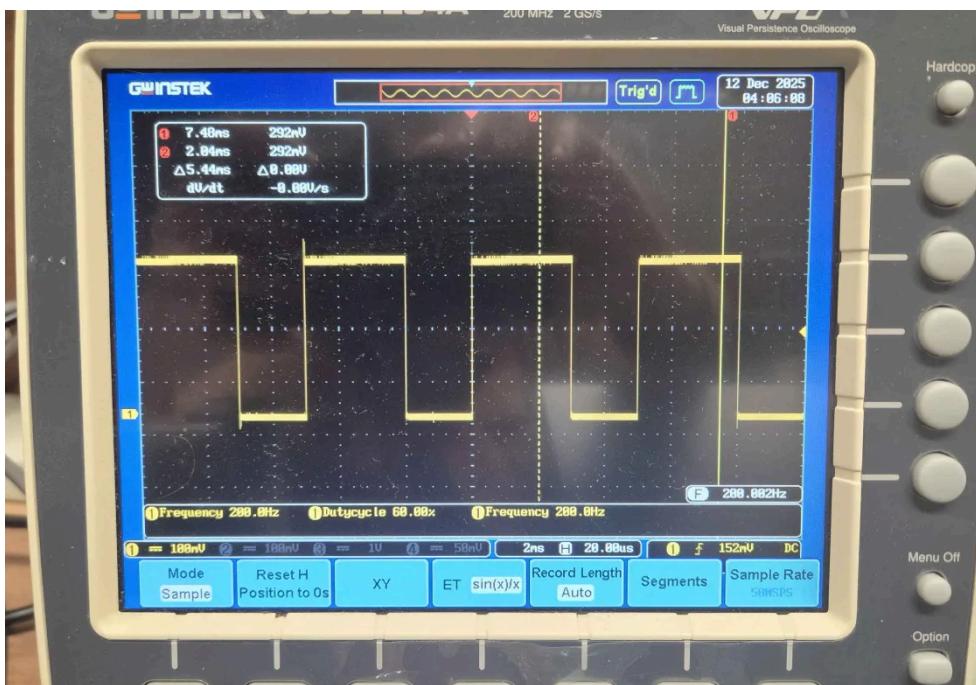
            LL_EXTI_ClearFlag_0_31(LL EXTI_LINE_0);
            LL_EXTI_EnableIT_0_31(LL EXTI_LINE_0);
        }
    }
}
}
}

```

Założono że drgania ustały, przycisk był w stanie niskim 3 razy. Po każdym upływie czasu timera sprawdzano czy przycisk jest „odciśnięty” i jeśli był w stanie niskim 3 razy to wyłączało timer 2 i włączano spowrotem przerwania na przycisku PA0.

Zbadano oscyloskopem wyjście dla diody LD5:

W trybie PWM1:

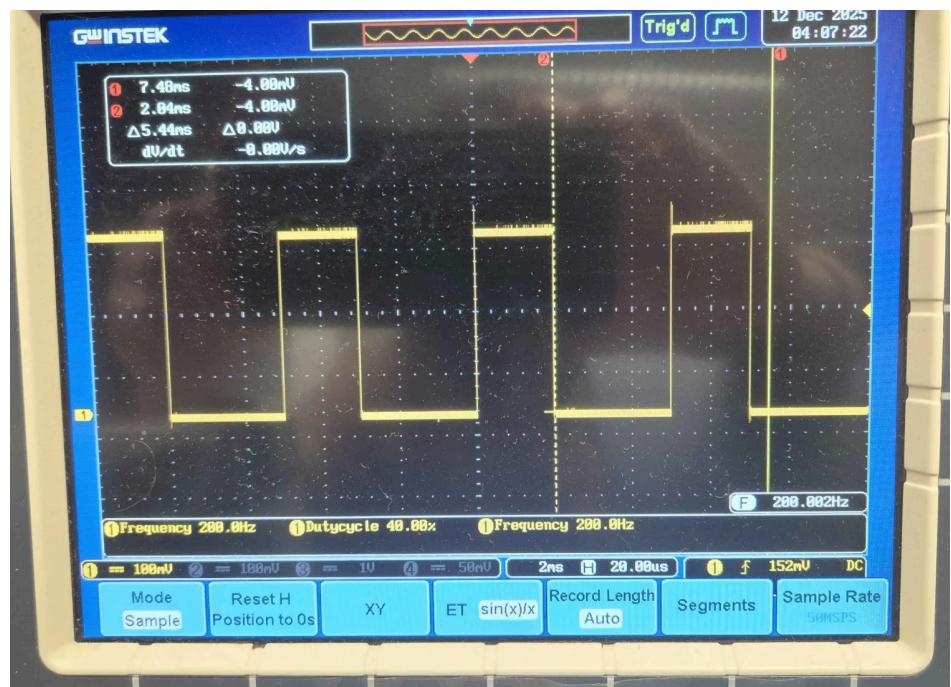


Widać, że częstotliwość oraz wypełnienie jest zgodne z założeniami zadania.

Aby zmienić tryb na PWM2 dla diody LD5 wstawiono następującą linijkę:

```
LL_TIM_OC_SetMode(TIM4, LL_TIM_CHANNEL_CH3, LL_TIM_OCMODE_PWM2);
```

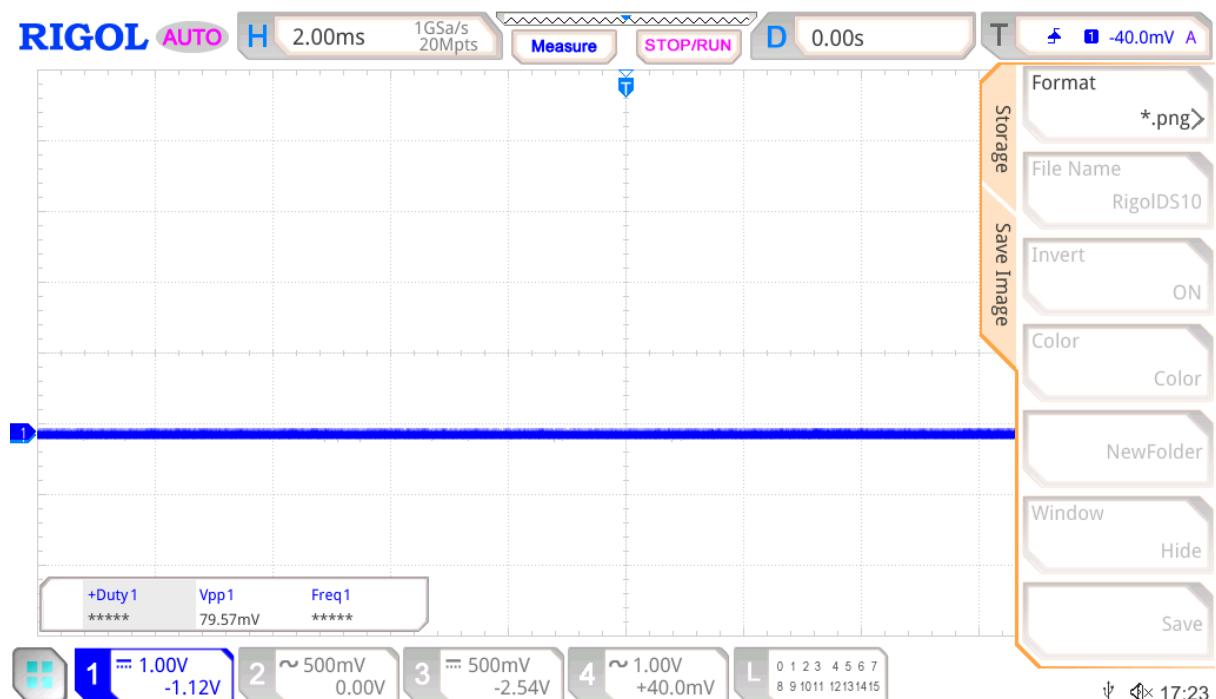
Nie zmieniano wartości CC dla kanału 3 i otrzymano następujący przebieg sygnału:



Wypełnienie wynosiło w tym przypadku 40%, ponieważ tryb PWM2 odwrotnie generuje sygnał wysoki, w tym trybie sygnał jest wysoki dla wartości CNT > CC.

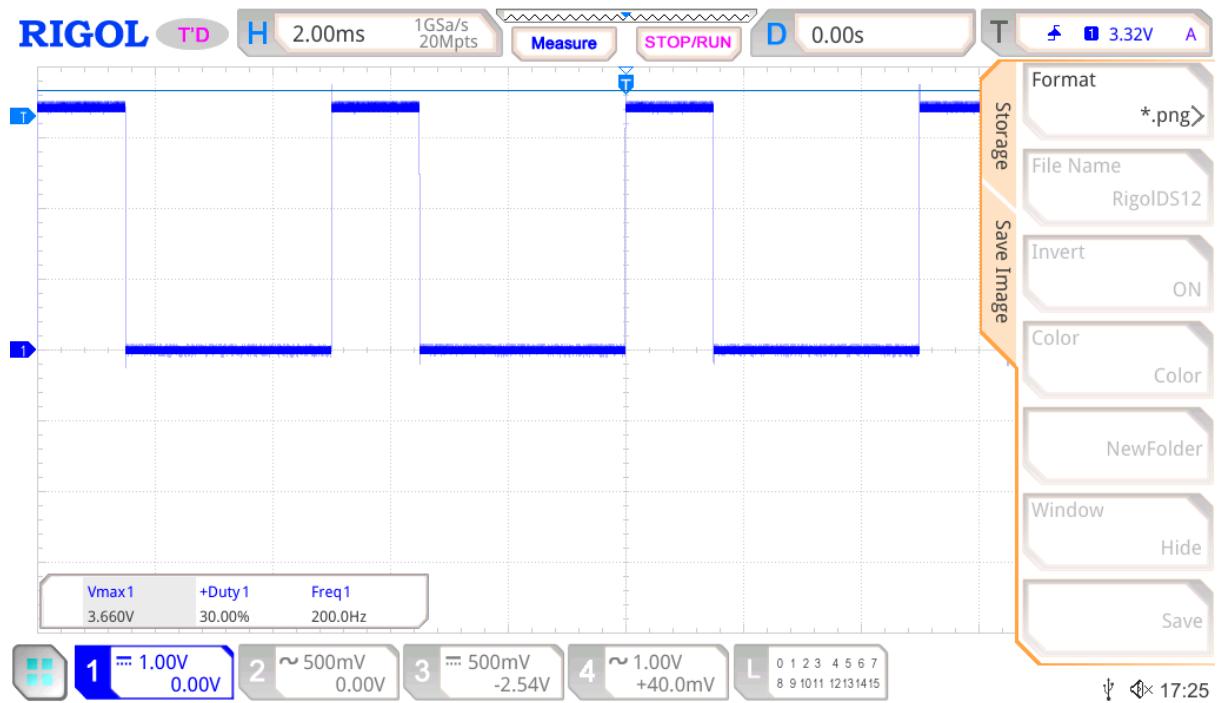
Sprawdzono także przebiegi dla diody LD6 dla wypełnienia 0%, 30% i 100% najpierw w trybie PWM1. 0% wypełnienia było od razu przy uruchomieniu programu, 30% po 3 kliknięciach a 100% po 10.

MSO5104 Mon December 15 17:23:26 2025



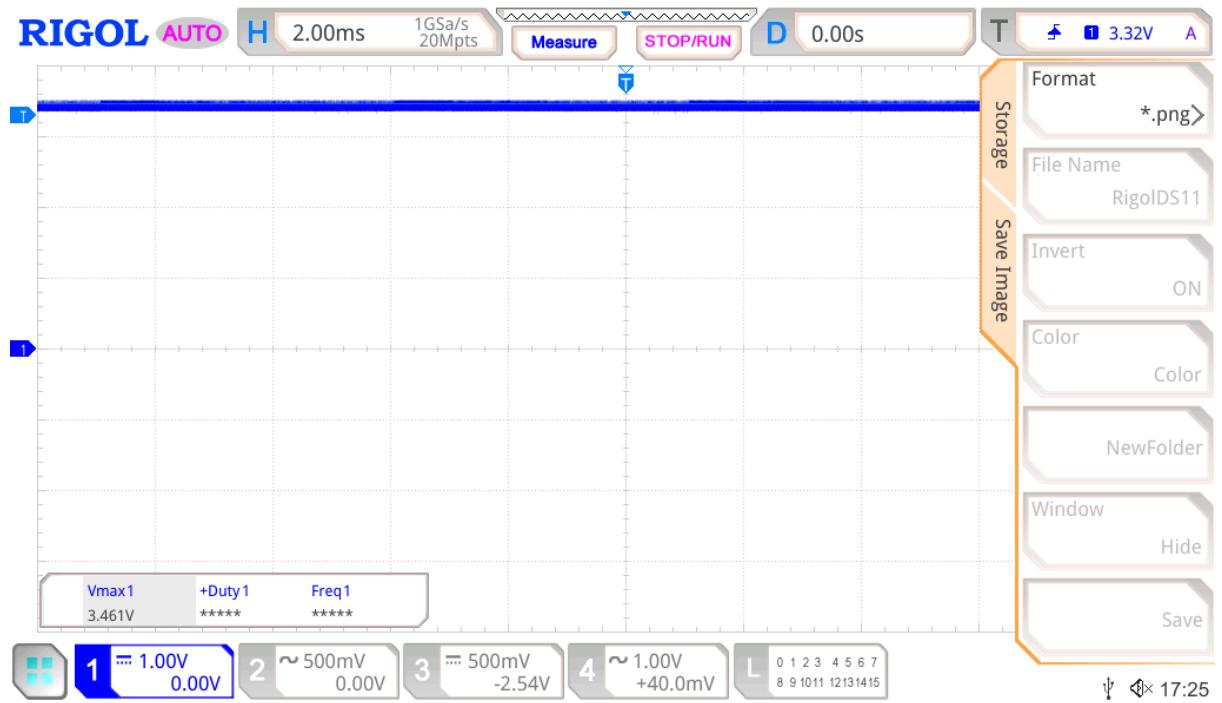
wypełnienie 0% PWM1

MSO5104 Mon December 15 17:25:53 2025



wypełnienie 30% PWM1

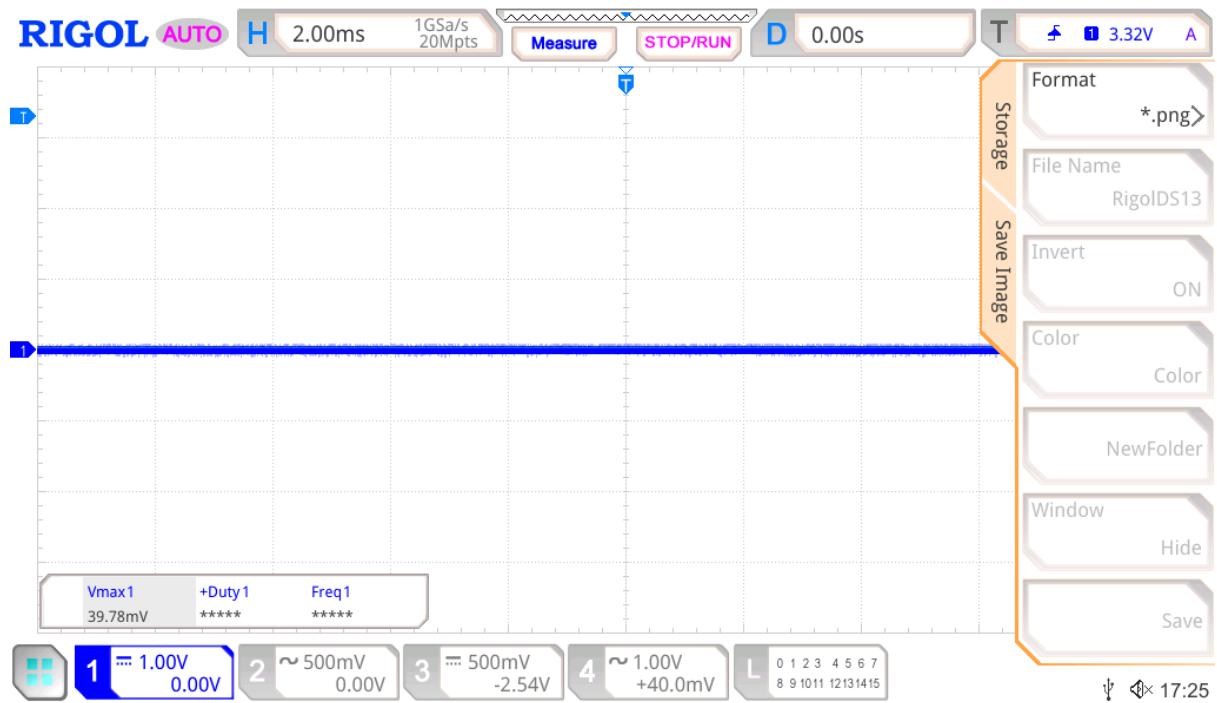
MSO5104 Mon December 15 17:25:27 2025



wypełnienie 100% PWM1

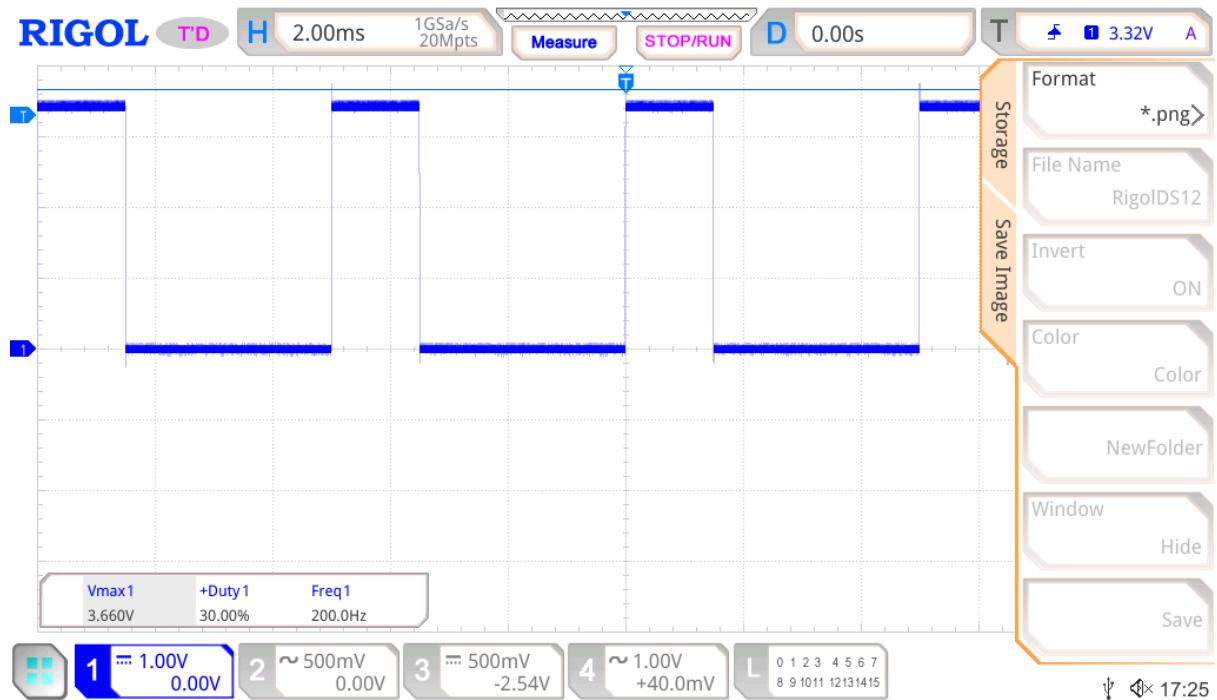
W przypadku wypełnienia 0% dla trybu PWM2 należało przycisnąć przycisk 10 razy, dla 30% 7 razy a dla 100% ani razu.

MSO5104 Mon December 15 17:26:00 2025



wypełnienie 0% PWM2

MSO5104 Mon December 15 17:25:53 2025



wypełnienie 30% PWM2



wypełnienie 100% PWM2

#### Zadanie 1.4.

Ostatnie zadanie polegało na zmierzeniu częstotliwości sygnału prostokątnego z wykorzystaniem układu CC. Do ćwiczenia został wykorzystany timer 3.

Na początku należało skonfigurować timer 3 w trybie CC. Liczył on ze swoim taktowaniem zegara, a kiedy wystąpiło narastające zbocze timera 2, to wartość ta była zapisywana do rejestru capture. W pętli *while* co sekundę sprawdzana była wartość tego rejestru i dzielono częstotliwość timera mierzącego przez liczbę taktów które naliczył podczas jednego okresu timera mierzonego.

Do generacji sygnału prostokątnego wykorzystano timer 2, w którym sygnał generowany miał określoną częstotliwość (10 kHz, 100 kHz lub 1 Mhz). Prescaler był ustawiony na 0, z związku z czym nie dzielił częstotliwości zegara, natomiast wartość ARR była wyliczona tak, żeby sygnał generowany miał zadaną częstotliwość. Wartość rejestru CC wynosiła połowę ARR, żeby osiągnąć wypełnienie 50%. Pin skonfigurowano w trybie *alternate* oraz z prędkością narastania zbocza *very high*, z racji że maksymalna generowana częstotliwość wynosiła 1 Mhz.

```
auto timer_generator_config() -> void {
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM2);
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
    LL_GPIO_SetPinMode(GPIOA, LL_GPIO_PIN_0, LL_GPIO_MODE_ALTERNATE);
    LL_GPIO_SetPinSpeed(GPIOA, LL_GPIO_PIN_0,
LL_GPIO_SPEED_FREQ VERY HIGH);
    LL_GPIO_SetAFPin_0_7(GPIOA, LL_GPIO_PIN_0, LL_GPIO_AF_1);
    LL_RCC_ClocksTypeDef rcc_clocks;
```

```

LL_RCC_GetSystemClocksFreq(&rcc_clocks);

uint32_t timer_clock = rcc_clocks.PCLK1_Frequency;

if (LL_RCC_GetAPB1Prescaler() != LL_RCC_APB1_DIV_1) {
    timer_clock *= 2;
}

const uint32_t target_freq = 10000; //10 kHz
uint32_t prescaler_value = 0;
uint32_t auto_reload = (timer_clock / target_freq) - 1;

LL_TIM_SetPrescaler(TIM2, prescaler_value);
LL_TIM_SetAutoReload(TIM2, auto_reload);

LL_TIM_OC_SetMode(TIM2, LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
LL_TIM_OC_EnablePreload(TIM2, LL_TIM_CHANNEL_CH1);

LL_TIM_OC_SetCompareCH1(TIM2, (auto_reload + 1) / 2);
LL_TIM_CC_EnableChannel(TIM2, LL_TIM_CHANNEL_CH1);

LL_TIM_GenerateEvent_UPDATE(TIM2);
LL_TIM_ClearFlag_UPDATE(TIM2);

LL_TIM_EnableCounter(TIM2);
}

```

Natomiast do pomiaru skorzystano z timera 3, którego pin PA6 ustawiono na tryb *alternate*. Szybkość narastania zbocza była także *very high*. Preskaler wynosił 0, żeby timer zliczał impulsy wewnętrznego zegara z maksymalną rozdzielczością, a ARR ustawiono na maksymalną możliwą wartość. Należało także ustawić *slave mode* timera 3, żeby po każdym zapisaniu wartości do capture register zerowany był rejestr CNT.

```

auto timer_measure_config() -> void {
    LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM3);
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);

    LL_GPIO_SetPinMode(GPIOA, LL_GPIO_PIN_6, LL_GPIO_MODE_ALTERNATE);
    LL_GPIO_SetPinSpeed(GPIOA, LL_GPIO_PIN_6,
LL_GPIO_SPEED_FREQ VERY HIGH);
    LL_GPIO_SetAFPin_0_7(GPIOA, LL_GPIO_PIN_6, LL_GPIO_AF_2);

    LL_GPIO_SetPinPull(GPIOA, LL_GPIO_PIN_6, LL_GPIO_PULL_NO);
}

```

```

LL_RCC_ClocksTypeDef rcc_clocks;
LL_RCC_GetSystemClocksFreq(&rcc_clocks);
measured_timer_clock = rcc_clocks.PCLK1_Frequency;
if (LL_RCC_GetAPB1Prescaler() != LL_RCC_APB1_DIV_1) {
    measured_timer_clock *= 2;
}
LL_TIM_SetPrescaler(TIM3, 0);
LL_TIM_SetAutoReload(TIM3, 0xFFFF);

LL_TIM_IC_SetActiveInput(TIM3, LL_TIM_CHANNEL_CH1,
LL_TIM_ACTIVEINPUT_DIRECTTI);
LL_TIM_IC_SetPrescaler(TIM3, LL_TIM_CHANNEL_CH1, LL_TIM_ICPSC_DIV1);
LL_TIM_IC_SetPolarity(TIM3, LL_TIM_CHANNEL_CH1,
LL_TIM_IC_POLARITY_RISING);
LL_TIM_IC_SetFilter(TIM3, LL_TIM_CHANNEL_CH1,
LL_TIM_IC_FILTER_FDIV1);
LL_TIM_SetTriggerInput(TIM3, LL_TIM_TS_TI1FP1);
LL_TIM_SetSlaveMode(TIM3, LL_TIM_SLAVEMODE_RESET);

LL_TIM_CC_EnableChannel(TIM3, LL_TIM_CHANNEL_CH1);
LL_TIM_EnableCounter(TIM3);
}

```

Pin PA0 (wyjście generatora timera 2) połączono kablem żeńsko-żeńskim z pinem PA6. Co 1 sekundę odczytywano rejestr capture timera 3 w pętli i wypisywano pomiar:

```

void measure_frequency(void*) {
    for(;;) {
        uint32_t captured_value = LL_TIM_IC_GetCaptureCH1(TIM3);
        uint32_t frequency = 0;
        if (captured_value > 0) {
            frequency = measured_timer_clock / captured_value;
        }
        dbprintf("Pomiar: CCRL=%u | Freq=%u Hz", captured_value,
frequency);
        isix::wait_ms(1000);
    }
}

```

Pomiary odczytywano poprzez złącze szeregowe przy użyciu komendy:  
picocom -b 115200 /dev/ttyUSB0

Zmierzono następujące wartości:

Dla 10 kHz:

time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz
time_base.cpp:27	Pomiar: CCR1=9998	Freq=10002 Hz

średnia częstotliwość = 10002 Hz

błąd bezwzględny = 2 Hz

błąd względny =  $2/10000 * 100\% = 0,02\%$

Niepewność pomiaru wyznaczono na podstawie informacji, że w metodzie cyfrowego pomiaru okresu, timer może się pomylić o +/- 1 takt zegara. To jest zakładana graniczna rozdzielncość. Wzór na wyliczenie zmierzonej częstotliwości to  $f = f_{clk} / N$ .

W takim razie dla  $N = 9997$ ,  $f = 100000000 \text{ Hz} / 9997 \approx 10003 \text{ Hz}$

Dla  $N = 9999$ ,  $f \approx 10001 \text{ Hz}$

niepewność pomiaru =  $|10003 \text{ Hz} - 10001 \text{ Hz}| = +/- 2 \text{ Hz}$

Dla 100 kHz:

time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz
time_base.cpp:27	Pomiar: CCR1=998	Freq=100200 Hz

średnia częstotliwość = 100200 Hz

błąd bezwzględny =  $|100000 - 100200| = 200 \text{ Hz}$

błąd względny =  $200 / 100000 * 100\% = 0,2\%$

$N = 997, f = 100000000 \text{ Hz} / 997 \approx 100301 \text{ Hz}$

Dla  $N = 999, f \approx 100100 \text{ Hz}$

niepewność pomiaru =  $|100100 \text{ Hz} - 100301 \text{ Hz}| = +/- 201 \text{ Hz}$

1 Mhz:

time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz
time_base.cpp:27	Pomiar: CCR1=98	Freq=1020408 Hz

średnia częstotliwość =  $1020408 \text{ Hz}$

błąd bezwzględny =  $20408 \text{ Hz}$

błąd względny =  $204080/1000000 * 100\% \approx 2,04\%$

$N = 97, f = 100000000 \text{ Hz} / 97 \approx 1030928 \text{ Hz}$

Dla  $N = 99, f \approx 1010101 \text{ Hz}$

niepewność pomiaru =  $|1010101 \text{ Hz} - 1030928 \text{ Hz}| = +/- 20827 \text{ Hz}$

Jak widać dokładność pomiaru bardzo spadała wraz ze wzrostem mierzonej częstotliwości. Im bardziej częstotliwość mierzona zbliża się do częstotliwości taktowania zegara timera liczącego tym mniej dokładny staje się pomiar. Jest to spowodowane tym, że błąd o jeden takt ma procentowo coraz większy wpływ na dokładność pomiaru ze wzrostem częstotliwości sygnału mierzonego.

Maksymalna częstotliwość to 33 Mhz, ale pomiar wynosił 100 Mhz. Potem mierzona częstotliwość była 0 Hz. Ostatni pomiar, który był zbliżony do rzeczywistej wartości to 4 Mhz (pomiar wynosił 4347826 Hz).

Najmniejszy sensowny pomiar to 2000 Hz. Dla wartości poniżej 1500 Hz zaczęły wychodzić bardzo duże, nieprawdziwe wyniki (np. 88652 Hz).

Aby poprawić dokładność pomiaru można skorzystać z preskalera input capture, np. ustawiony na wartość 8. Wówczas zliczał by zbocze narastające timera mierzonego 8 razy dłużej, co zwiększyłoby rozdzielczość pomiaru i dokładność. Na końcu trzeba tylko pomnożyć zmierzoną częstotliwość razy 8.

Alternatywnie można by ustawić timer mierzący na tryb external clock mode i wtedy mierzony timer traktowałby timer wykonujący pomiar. Wówczas można by np. co 1s sprawdzać ile taktów timer mierzący naliczył i ta wartość będzie częstotliwością timera mierzonego.