Zuzanna Godek 318373
Sprawozdanie z laboratorium 1 MARM

Celem laboratorium było zapoznanie się z zegarami, obsługą pinów GPIO oraz ze sposobem zgłaszania przerwań zewnętrznych przy użyciu kontrolera EXTI.

Do konfiguracji, budowania i uruchamiania kodu wykorzystano komendy:

```
python3 waf configure --debug --board=stm32f411e_disco
--crystal-hz=8000000
python3 waf clean build program
```

Zadanie 1.1.

Po uruchomieniu kodu dostarczonego przez prowadzącego dioda włączała się na dłuższy czas niż ustawione jest w programie, czyli kilka sekund (zmierzono stoperem około 6 s) zamiast 0,5 s zapisanych w programie. Było to spowodowane złą konfiguracją zegara HCLK. Z pliku isix_config.h można odczytać jaką konfigurację zegara zakłada oprogramowanie:

```
// --------  Hardware specific config ------------
#define  CONFIG_XTAL_HZ   8000000LU
#define  CONFIG_HCLK_HZ  100000000LU
#define     CONFIG_PCLK1_HZ (CONFIG_HCLK_HZ/2)
#define  CONFIG_PCLK2_HZ (CONFIG_HCLK_HZ/1)
```

CONFIG_HCLK_HZ jest ustawione na 100000000LU czyli zakłada częstotliwość zegara 100 Mhz. Rzeczywista częstotliwość była ustawiona na 8 Mhz, co widać w tym kawałku kodu:

```
//! Enable HSE generator
LL_RCC_HSE_Enable();
…
LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_HSE);
```

Źródło zegara jest ustawione na HSE (czyli zewnętrzny kwarc, którego częstotliwość wynosi 8 MHz), a PLL nie jest używany. W związku z tym kiedy w kodzie jest włączone uśpienie na 500ms:

```
void test_thread(void*) {
    for(int i=0;;++i) {
        isix::wait_ms(500);
        if(i%2==0) {
            dbprintf("Loop %i",i>>1);
        }
        periph::gpio::set(led_0, i%2);
    }
}
```

To ponieważ w configu jest ustawiona częstotliwość na 100 MHz odliczane jest:
100 000 000 tyknięć/s * 0,5 s = 50 000 000 tyknięć zegara.
Skoro w rzeczywistości częstotliwość HCLK jest ustawiona na 8 Mhz to:
(50 000 000 t) / (8 000 000 t/s) = 6,25 s
Czyli rzeczywisty czas w jakim led się przełącza to około 6,25 s, co zgadza się z obserwacją.

Żeby to naprawić należało, skonfigurować mnożnik PLL i ustawić go jako źródło SysClk w pliku clocks_and_pll.cpp:

```
LL_FLASH_SetLatency( LL_FLASH_LATENCY_3 );
…
//! Set MCU Prescallers
LL_RCC_SetAHBPrescaler( LL_RCC_SYSCLK_DIV_1 );
LL_RCC_SetAPB2Prescaler( LL_RCC_APB2_DIV_1 );
LL_RCC_SetAPB1Prescaler( LL_RCC_APB1_DIV_2 );
//! Enable HSE generator
LL_RCC_HSE_Enable();
…
LL_RCC_PLL_ConfigDomain_SYS(LL_RCC_PLLSOURCE_HSE, 8, 400, 4);
LL_RCC_PLL_Enable();
…
LL_RCC_SetSysClkSource(LL_RCC_SYS_CLKSOURCE_PLL);
```

W powyższych fragmentach kodu widać konfigurację PLL, w które PLL_M = 8, PLL_N = 400, a PLL_P na 4.
PLL_M to dzielnik wejścia, czyli 8 MHz / 8 = 1 MHz
PLL_N to mnożnik, czyli 1 MHz * 400 = 400 MHz
PLL_P to dzielnik na wyjściu, czyli 400 Mhz / 4 = 100 Mhz

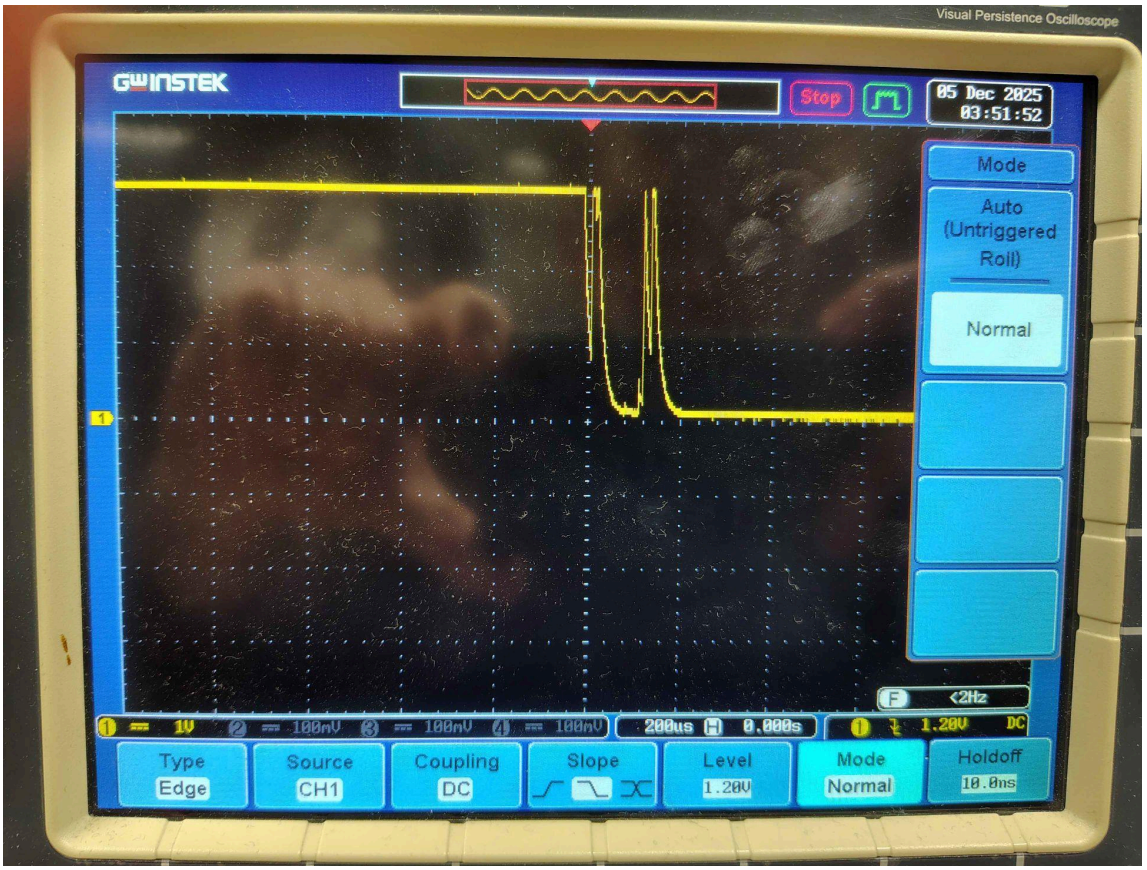
Ustawiono także odpowiednio preskalery magistral APB1 na 2 i APB2 na 1. Skonfigurowano także opóźnienie pamięci Flash na większe, ponieważ czas dostępu do pamięci Flash jest stały i mniejszy, więc i przy wysokich częstotliwościach zegara (100 MHz) pamięć nie nadąża z dostarczaniem danych w jednym cyklu zegara. Na końcu ustawiono źródło SysClk na PLL. Po tych zmianach osiągnięto zadaną w poleceniu częstotliwość HCLK 100 Mhz, ABP1 50 MHz i APB2 100 MHz, a także dioda mrugała poprawnie co 500 ms.

Zadanie 1.2.
• Umieścić oscylogram powstały po wciśnięciu klawisza USER. Wyciągnij wnioski jakie płyną z tego oscylogramu?
W pierwszym podpunkcie należało zbadać wyjście przycisku PA0 oscyloskopem. Przycisk był naciskany i puszczany niechlujnie, żeby móc zaobserwować drgania zestyków. Jest to fizyczne obijanie się zestyków przycisku po naciśnięciu go i puszczeniu.
Konfiguracja oscyloskopu do pomiaru była następująca:

Trigger – falling edge
Mode – normal
Time/div – 200 µs
Przycisk ustawiono na tryb floating:

```
setup( num::PA0, mode::in{ pulltype::floating } );
```

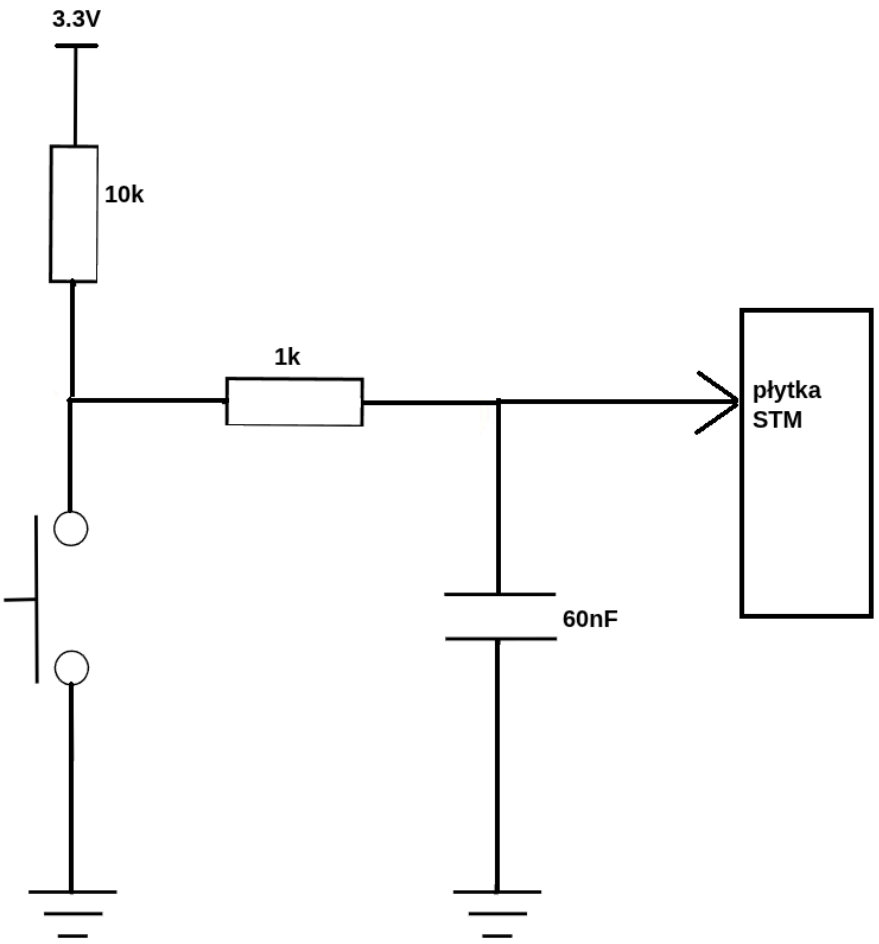
Drgania nie zawsze występowały, ale w końcu udało się je uchwycić na zdjęciu, na którym widać je po puszczeniu przycisku (dlatego, że *trigger* to był *falling edge* na oscyloskopie):



W tym wypadku drgania trwały około 200 µs.

• Uzasadnić dlaczego konieczna jest eliminacja drgań zestyków w przypadku używania przełączników mechanicznych?
Eliminacja drgań zestyków jest konieczna w przypadku przełączników mechanicznych, ponieważ może powodować zliczanie wielu kliknięć przycisku, kiedy w rzeczywistości wciskany jest tylko raz. Takie zjawisko zaobserwowano na początku laboratorium, kiedy licznik przeskakiwał kilka (zazwyczaj 2) wartości po jednym przyciśnięciu.

• Czy drgania zestyków można wyeliminować w sposób sprzętowy zaproponuj odpowiedni schemat, wraz ze stosownymi obliczeniami. Proszę również uzasadnić i opisać wybrany schemat.
Drgania zestyków można wyeliminować układem RC, w którym jest kondensator powodujący opóźnienie przesyłu sygnału na wyjście poprzez swoje ładowanie i rozładowywanie. Schemat takiego układu wygląda następująco:



Najpierw trzeba wyliczyć stałą czasową kondensatora τ. Oznacza ona czas w jakim około 37% procent kondensatora się wyładuje lub 67% się naładuje. Zakłada się, że trzeba ją wymnożyć razy 5, żeby otrzymać czas po którym kondensator się w pełni wyładuje lub naładuje.

$\tau_{disch} = R_1 C$

ponieważ R_1= 1kΩ

potrzebujemy żeby 5τ > 200µs, załóżmy że chcemy otrzymać 5τ = 300µs -> τ = 60µs

$C = \frac{\tau_{disch}}{R_1} = \frac{60\mu}{1k} = 60nF$

Natomiast czas w jakim się naładuje kondensator wyliczono:

$\tau_{ch} = (R_1 + R_2)C = 11k\Omega \cdot 60nF = 660\mu s$

Czyli $5\tau_{ch} = 3,3ms$

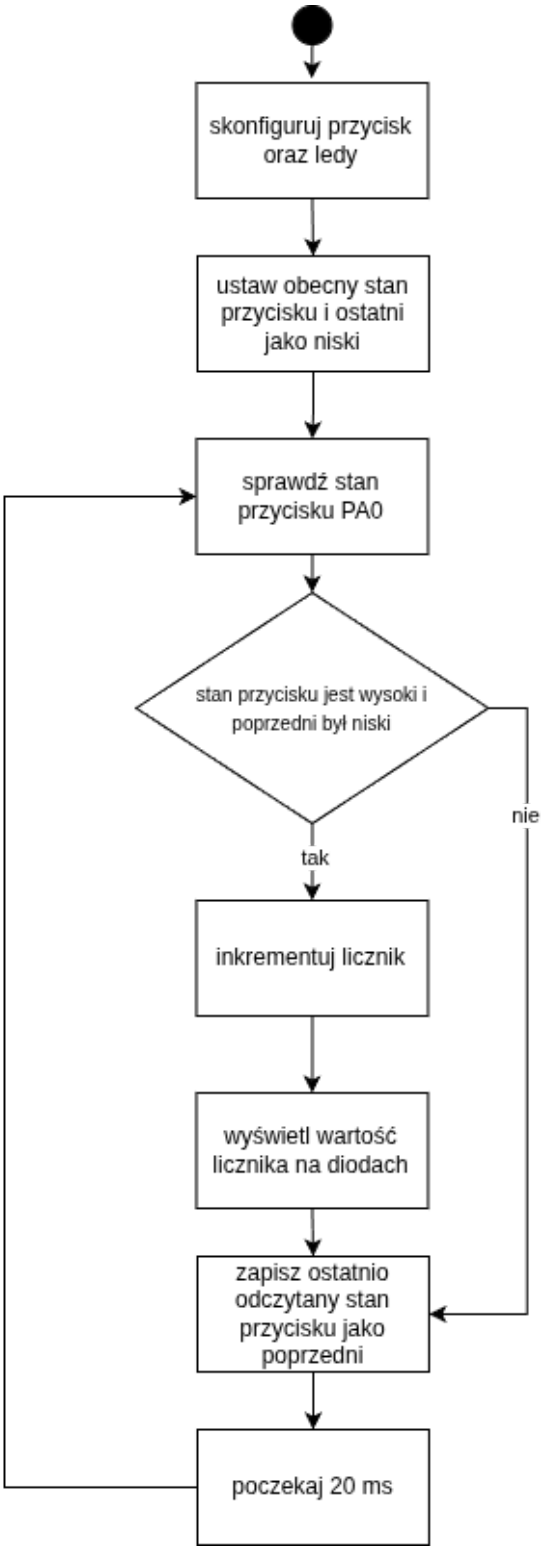
Wartość ta nie jest na tyle duża, żeby opóźnienie przycisku było widoczne dla ludzkiego oka, także wyliczona pojemność kondensatora jest poprawna.

Uwaga: Schemat realizuje przycisk *active low*, czyli stan niski pinu jest interpretowany jako przycisk naciśnięty, w kodzie jest odwrotnie, ale schemat ma być tylko teoretycznym narzędziem.

• Narysuj diagramy aktywności UML zastosowanego algorytmu programowej eliminacji drgań zestyków w trybie obsługi z odpytywaniem, oraz w trybie z przerwaniem zewnętrznym EXTI.

W następnym podpunkcie zadaniem było napisanie programu zliczającego naciśnięcia przycisku PA0 i wyświetlanie wartości tego licznika na czterech diodach led, z wykorzystaniem biblioteki ISIX.

Logika programu była następująca:



Najpierw należało skonfigurować diody i przycisk:

```
namespace {
	constexpr auto led_4 = periph::gpio::num::PD12;
	constexpr auto led_3 = periph::gpio::num::PD13;
	constexpr auto led_5 = periph::gpio::num::PD14;
	constexpr auto led_6 = periph::gpio::num::PD15;
	constexpr auto key_0 = periph::gpio::num::PA0;
	volatile uint32_t counter = 0;
}

auto input_output_config() -> void {
	using namespace periph::gpio;
	setup( led_4, mode::out{ outtype::pushpull, speed::low } );
	setup( led_3, mode::out{ outtype::pushpull, speed::low } );
	setup( led_5, mode::out{ outtype::pushpull, speed::low } );
	setup( led_6, mode::out{ outtype::pushpull, speed::low } );

	LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);

	setup( num::PA0, mode::in{ pulltype::down } );
}
```

Samą logikę zliczania i wyświetlania liczby przyciśnięć zrealizowano przy użyciu pętli odpytującej przycisk i zapisującej ostatni stan przycisku.

```
auto counter_task() -> void {
	bool last_button_state = false;
	bool curr_button_state = false;
	while (true) {
		curr_button_state = periph::gpio::get(key_0);
		if (curr_button_state && !last_button_state) {
			counter++;
			periph::gpio::set( led_4, (counter >> 0) & 1 );
			periph::gpio::set( led_3, (counter >> 1) & 1 );
			periph::gpio::set( led_5, (counter >> 2) & 1 );
			periph::gpio::set( led_6, (counter >> 3) & 1 );
		}
		last_button_state = curr_button_state;
		isix::wait_ms(20);
	}
}
```

Mitygację drgania zestyków osiągnięto poprzez usypianie programu w tym pętli na 20ms. Dawało to wystarczający bufor na przeczekanie drgań oraz nie spowalniało w widoczny

sposób odczytu przycisku. W trakcie laboratorium nie uwzględniono jeszcze odczytu drgań z oscyloskopu, ponieważ zadania wykonywano nie po kolei.

W kolejnym podpunkcie program miał zostać zmodyfikowany tak, aby wykorzystywał bibliotekę LL do obsługi portów GPIO. Wykorzystany algorytm był taki sam jak w poprzednim programie, w związku z czym ostatni diagram UML nadal obowiązywał. Zmodyfikowano więc konfigurację:

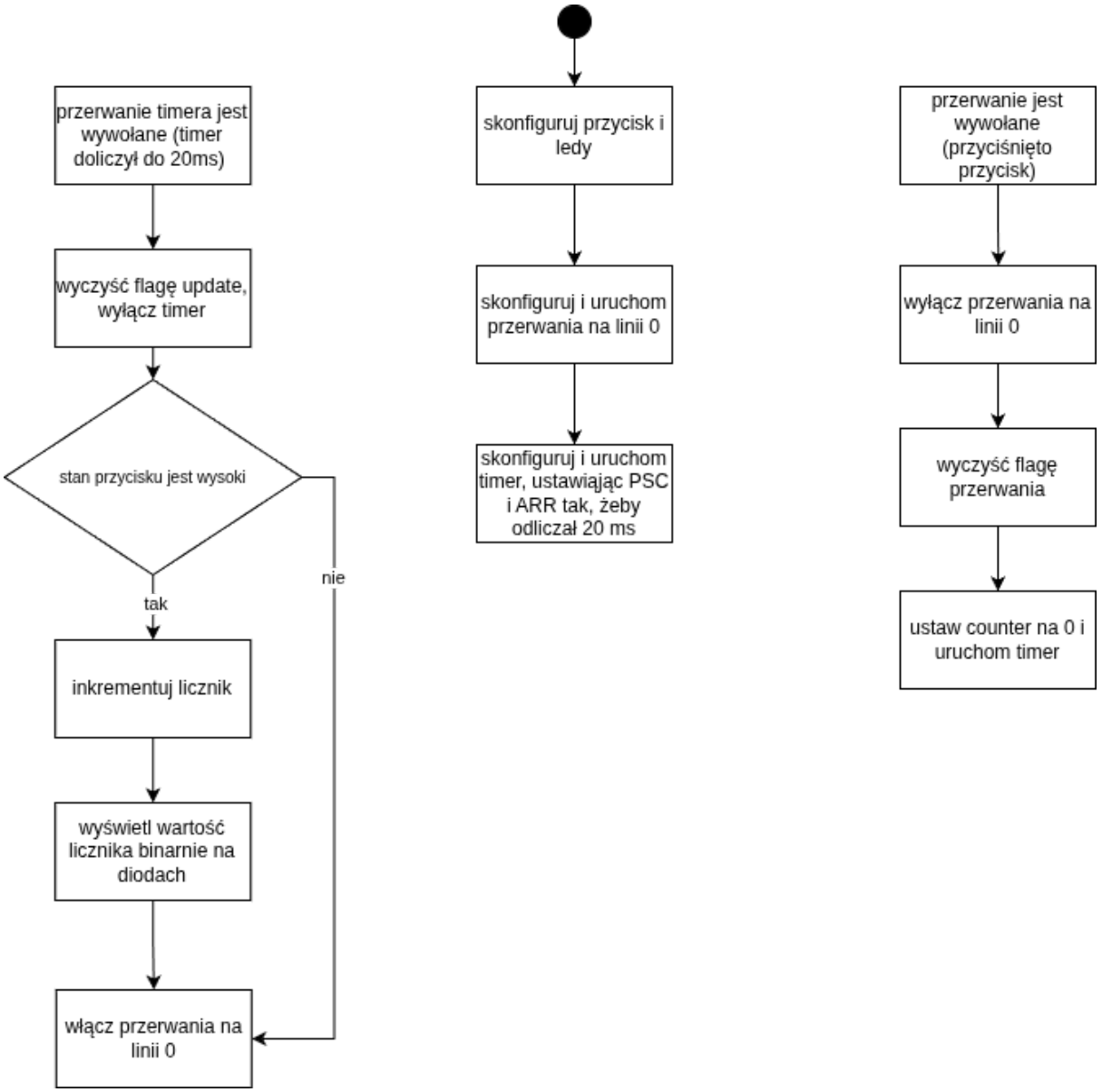
```
auto input_output_config() -> void {
      LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD);

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

      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 (GPIOD, LL_GPIO_PIN_14,
LL_GPIO_SPEED_FREQ_LOW);
      LL_GPIO_SetPinSpeed (GPIOD, LL_GPIO_PIN_15,
LL_GPIO_SPEED_FREQ_LOW);

      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);
}
```

Następnie także zmodyfikowano pętlę. Główna różnica w porównaniu do biblioteki ISIX to, że w bibliotece LL nie można ustawiać wartości danej wartości na wyjściu pinu tylko można ustawiać pin na stan wysoki (`LL_GPIO_SetOutputPin`) lub niski (`LL_GPIO_ResetOutputPin`). Można także przełączać wartość pinu na przeciwną (`LL_GPIO_TogglePin`), ale nie ma to tutaj zastosowania.

```
auto counter_task() -> void {
	bool last_button_state = false;
	bool curr_button_state = false;
	while (true) {
		curr_button_state = LL_GPIO_IsInputPinSet(GPIOA,
	LL_GPIO_PIN_0);
		if (curr_button_state && !last_button_state) {
			counter++;
			if (counter & 1)
				LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_12);
			else
				LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_12);
			if ((counter >> 1) & 1 )
				LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_13);
			else
				LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_13);
			if ((counter >> 2) & 1 )
				LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_14);
			else
				LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_14);
			if ((counter >> 3) & 1 )
				LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_15);
			else
				LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_15);
		}
		last_button_state = curr_button_state;
		isix::wait_ms(20);
	}
}
```

W ostatnim podpunkcie tego zadania należało wykorzystać przerwania zamiast odpytywania przycisku w pętli. Do eliminacji drgań zestyków zdecydowano się wykorzystać timer, ze względu na to że nie zaleca się wykorzystania funkcji *sleep* w przerwaniu. W związku z czym jeśli przerwanie zostało wywołane przez naciśnięcie przycisku (*trigger* ustawiony na *rising*), to wówczas wyłączane były przerwania tego przycisku (na linii 0) oraz uruchamiano timer. Kiedy czas upłynął, timer zgłaszał swoje przerwanie, w którym sprawdzano stan przycisku oraz jeśli wykryto, że nadal jest wciśnięty inkrementowano counter i wyświetlano jego wartość na ledach. Na końcu timer uruchamiał z powrotem przerwania dla przycisku.

Schemat blokowy algorytmu wyglądał następująco:



Na początku poza ledami i przyciskiem należało skonfigurować także przerwania i timer:

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

      LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_12,
LL_GPIO_MODE_OUTPUT);
      LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_13,
LL_GPIO_MODE_OUTPUT);
      LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_14,
LL_GPIO_MODE_OUTPUT);
      LL_GPIO_SetPinMode(GPIOD, LL_GPIO_PIN_15,
LL_GPIO_MODE_OUTPUT);
```

```
	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 (GPIOD, LL_GPIO_PIN_14,
LL_GPIO_SPEED_FREQ_LOW);
	LL_GPIO_SetPinSpeed (GPIOD, LL_GPIO_PIN_15,
LL_GPIO_SPEED_FREQ_LOW);

	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);
}
```

Dodatkowo należało zdefiniować konfigurację timera:

```
auto 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;
	}

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

```
	LL_TIM_SetPrescaler(TIM2, prescaler_value);
	LL_TIM_SetAutoReload(TIM2, 200 - 1); // 20 ms
	LL_TIM_GenerateEvent_UPDATE(TIM2);
	LL_TIM_ClearFlag_UPDATE(TIM2);
	LL_TIM_EnableIT_UPDATE(TIM2);
	NVIC_SetPriority(TIM2_IRQn, 0);
	NVIC_EnableIRQ(TIM2_IRQn);
}
```

Na początku odczytywano częstotliwość zegara na magistrali APB1 oraz wartość preskalera APB1, ponieważ jeśli jest ona różna od 1 to wówczas zegar timera ma dwukrotnie większą częstotliwość od taktowania APB1. Następnie ustawiano prescaler tak, aby timer zliczał co 0,1 ms. To zapewniało, że timer trwał 20 ms przy rejestrze ARR ustawionym na 199. Generowany był także Event UPDATE dla TIM2, żeby konfiguracja „weszła w życie" od razu.

Następnie trzeba było zdefiniować działanie przerwania przycisku, tak żeby wyłączał przerwania na linii 0, następnie czyścił flagę, a na końcu ustawiał i włączał timer TIM2.

```
void exti0_isr_vector() {
	if(LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_0)) {
		LL_EXTI_DisableIT_0_31(LL_EXTI_LINE_0);
		LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_0);
		LL_TIM_SetCounter(TIM2, 0);
		LL_TIM_EnableCounter(TIM2);
	}
}
```

Przerwanie timera natomiast czyściło flagę timera UPDATE, wyłączało timer, oraz sprawdzało czy przycisk był przyciśnięty i jeśli tak to inkrementowało licznik oraz wyświetlało jego wartość binarną na diodach. Na końcu jeszcze raz czyściło flagę przerwań oraz włączało z powrotem przerwania na linii 0.

```
void tim2_isr_vector() {
	if (LL_TIM_IsActiveFlag_UPDATE(TIM2)) {
		LL_TIM_ClearFlag_UPDATE(TIM2);
		LL_TIM_DisableCounter(TIM2);
		if (LL_GPIO_IsInputPinSet(GPIOA, LL_GPIO_PIN_0)) {
			counter++;
			if (counter & 1)
				LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_12);
			else
				LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_12);
			if ((counter >> 1) & 1 )
				LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_13);
```

```
                else
                        LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_13);
                if ((counter >> 2) & 1 )
                        LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_14);
                else
                        LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_14);
                if ((counter >> 3) & 1 )
                        LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_15);
                else
                        LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_15);
        }
        LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_0);
        LL_EXTI_EnableIT_0_31(LL_EXTI_LINE_0);
    }
}
```

Zadanie 1.3.

Najpierw należało napisać program, który z maksymalną częstotliwością przełączał stan wyjścia PB0.

Najpierw napisano program, który przełączał wyjście diody przy użyciu toggle:

```
auto toggle_PB0() -> void {
    for(;;) {
        LL_GPIO_TogglePin(GPIOB, LL_GPIO_PIN_0);
    }
}
```

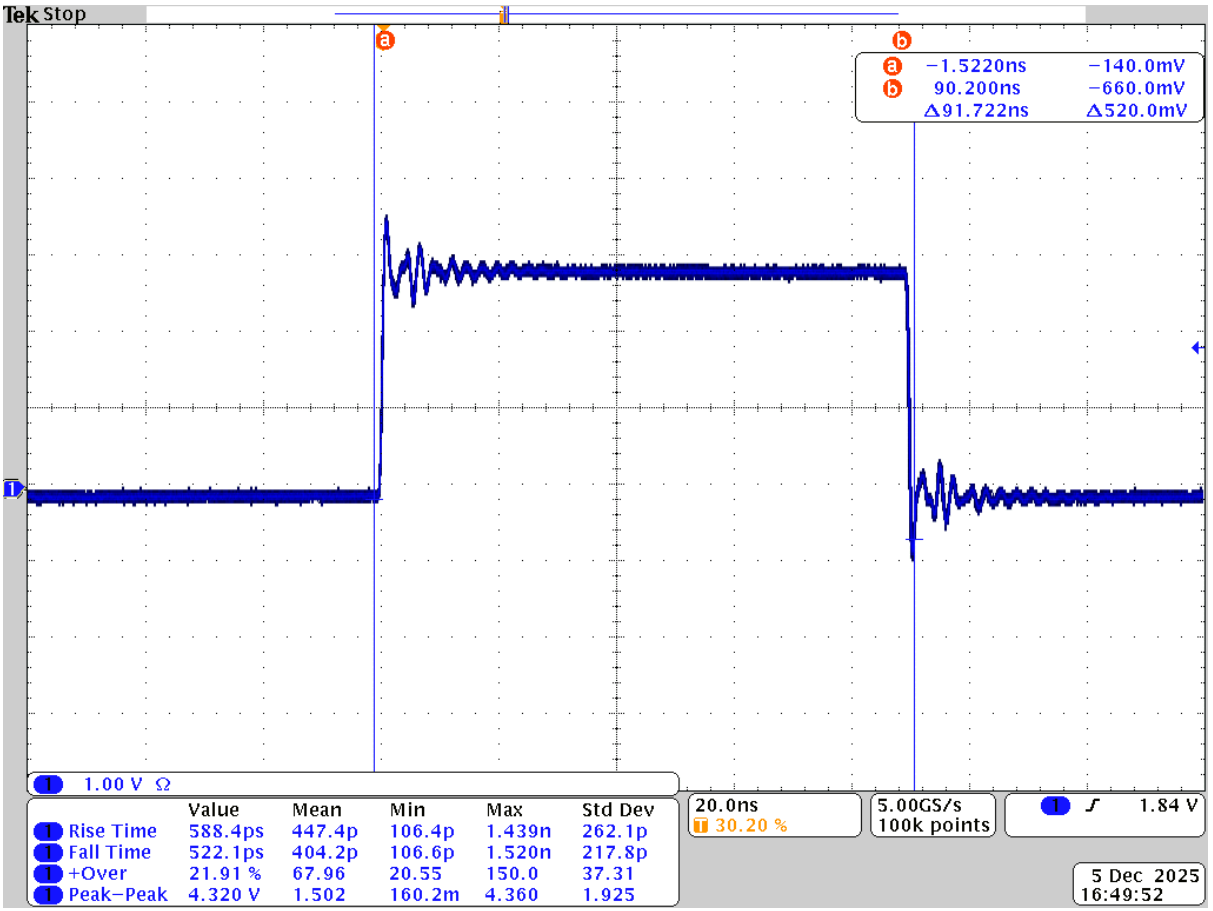
Dla każdego pomiaru należało skonfigurować odpowiednią prędkość pinu PB0, sprawdzane były wszystkie dostępne wartości (*low, medium, high* i *very high*):

```
auto pb0_config() -> void {
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);

    LL_GPIO_SetPinOutputType(GPIOB, LL_GPIO_PIN_0,
LL_GPIO_OUTPUT_PUSHPULL);

    LL_GPIO_SetPinSpeed(GPIOB, LL_GPIO_PIN_0,
LL_GPIO_SPEED_FREQ_LOW);
    // LL_GPIO_SetPinSpeed(GPIOB, LL_GPIO_PIN_0,
LL_GPIO_SPEED_FREQ_MEDIUM);
    // LL_GPIO_SetPinSpeed(GPIOB, LL_GPIO_PIN_0,
LL_GPIO_SPEED_FREQ_HIGH);
    // LL_GPIO_SetPinSpeed(GPIOB, LL_GPIO_PIN_0,
LL_GPIO_SPEED_FREQ_VERY_HIGH);
```

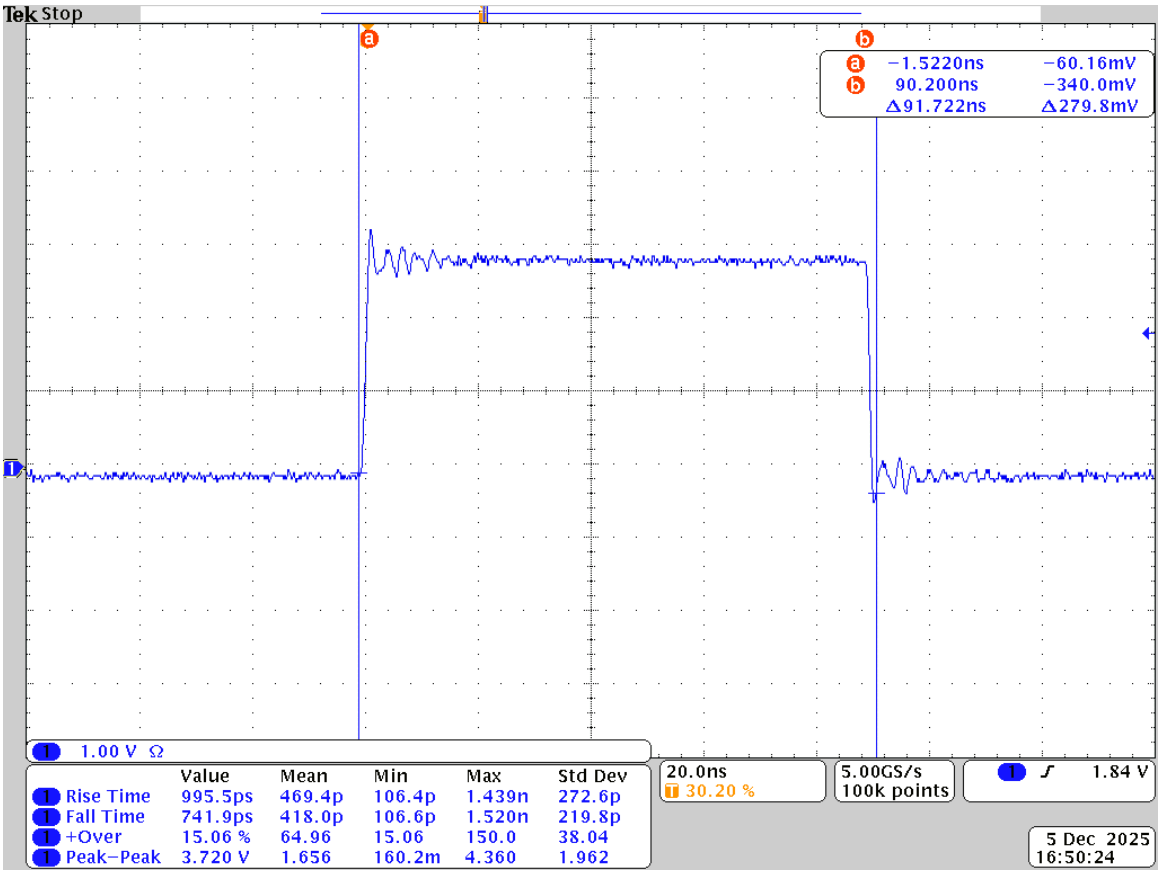
```
        LL_GPIO_ResetOutputPin(GPIOB, LL_GPIO_PIN_0);
}
```

• Umieścić oscylogramy prezentujące sygnał wyjściowy w zależności od ustawienia trybu szybkości pracy portu. Wyciągnąć wnioski płynące z oscylogramów.

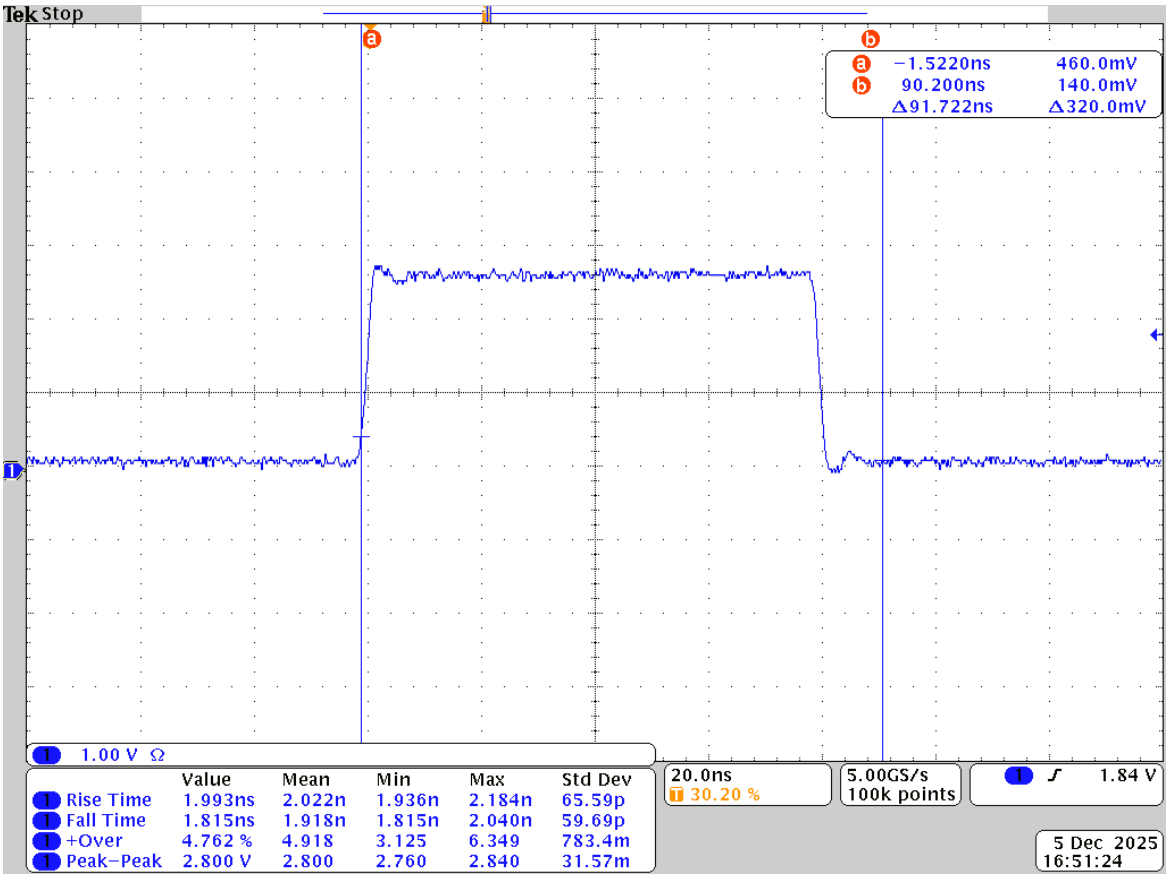
Dla prędkości narastania zbocza *very high*:



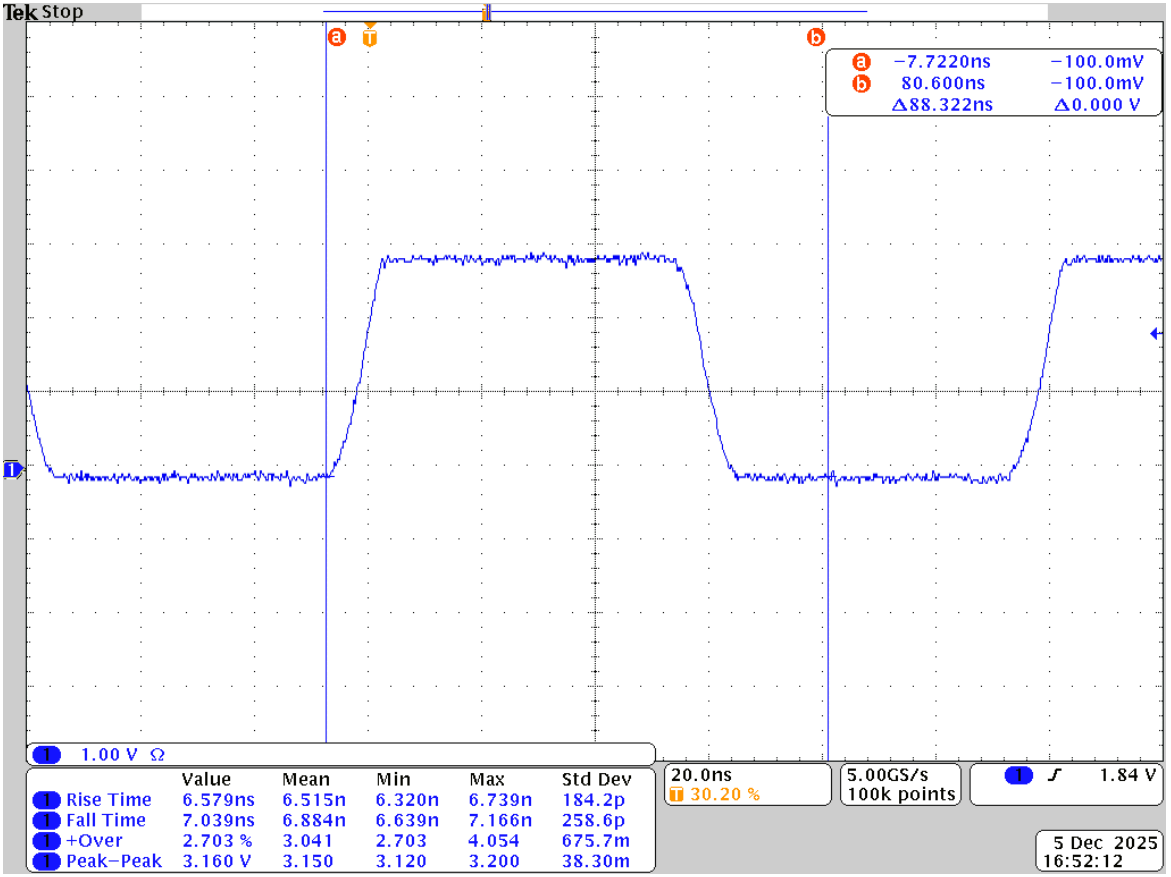
Dla prędkości *high*:



Dla prędkości *medium*:



Dla prędkości *low*:



Ze zdjęć z oscyloskopu można zaobserwować zakłócenia pojawiające się w sygnale dla bardzo dużej i dużej prędkości narastania zbocza, co jest zgodne z dokumentacją płytki. Nazywa się to zjawiskiem dzwonienia. Dlatego trzeba ustawić odpowiednią szybkość narastania zbocza dla danego modułu GPIO, uwzględniając jego tolerancję napięciową. Gdy prędkość narastania była niska, to zbocze jest dużo łagodniejsze i wykres ma kształt prawie jak trapez, co widać na ostatnim zdjęciu.

Tak naprawdę pętla *for* lub *while* przełączająca stan pinu PB0 nie da najszybszej możliwej częstotliwości, ponieważ czas wykonania programu jest zbyt wolny. Aby osiągnąć maksymalną częstotliwość przełączania pinu należałoby skonfigurować timer TIM3, którego kanał 3 jest podłączony do pinu PB0. Można TIM3 ustawić na tryb PWM, gdzie preskaler był ustawiony na 0, ARR na 1 i CCR też na 1. Dzięki temu uzyskanoby częstotliwość przełączania 50 Mhz.
f_out = f_clock / ((PSC+1)*(ARR+1)) = 100 Mhz / (1*2) = 50 Mhz
A duty cycle:
CCR / (ARR + 1) * 100% = 1 / 2 * 100% = 50% -> na wyjściu byłoby połowę czasu wartość wysoka i drugą połowę niska

Wówczas dla prędkości narastania zbocza *low* wykres na oscyloskopie wyglądałby jak „piła” lub „płetwa rekina”, ponieważ pin GPIO nie zdążyłby osiągnąć stanu wysokiego lub niskiego zanim już musiałby się przełączyć na przeciwny.