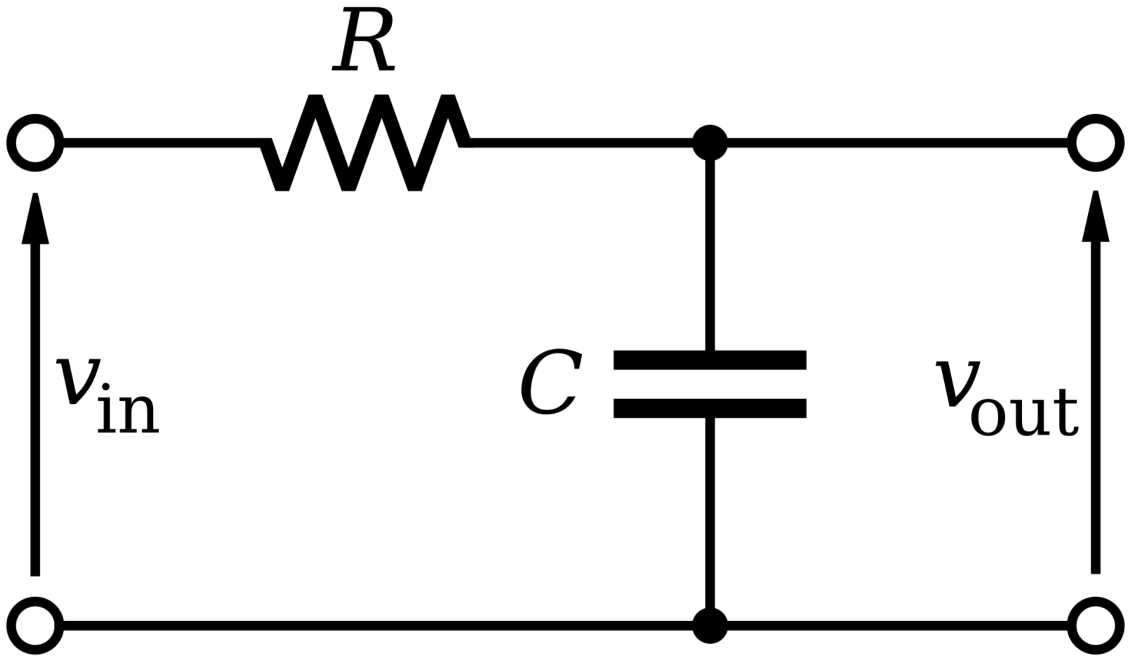
Zuzanna Godek 318373
MARM sprawozdanie z laboratorium 6

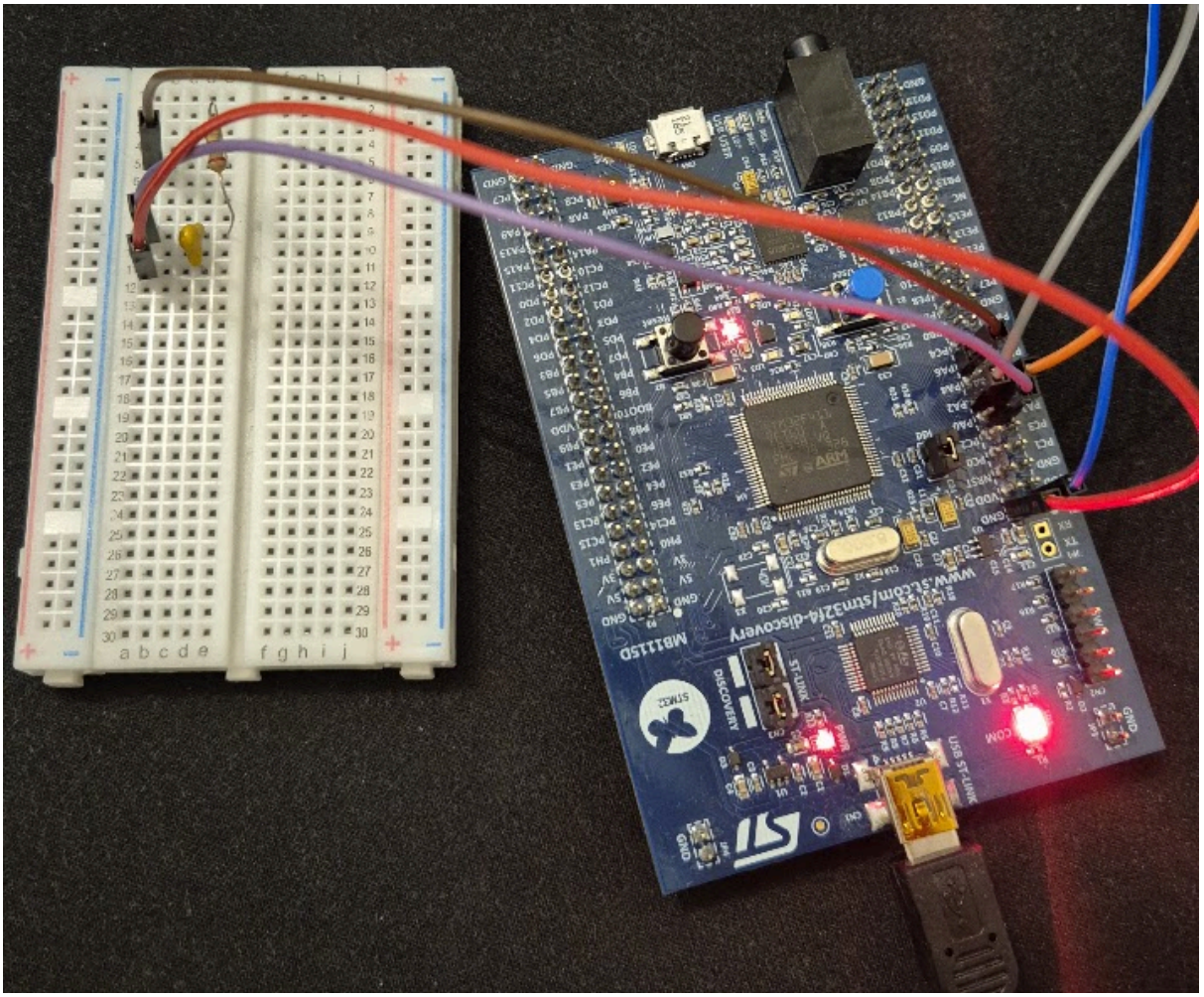
Celem laboratorium było zapoznanie się z przetwarzaniem sygnałów analogowych za pomocą mikrokontrolerów rodziny STM32, oraz obsługą zintegrowanych przetworników analogowych-cyfrowych oraz cyfrowo-analogowych.

Zadanie 1.1.
W tym zadaniu należało napisać program, który zmierzyłby wartość skuteczną napięcia dla różnych przebiegów o częstotliwości 1 kHz.

Z racji, że laboratorium było wykonywane w domu i nie posiadam generatora napięcia zmiennego, jedyną opcją był generator PWM płytki STM32. Aby otrzymać płynniejszą zmianę napięcia skorzystano z filtra dolnoprzepustowego RC składającego się z kondensatora 330 nF i opornika 220 Ω, bo takie wartości dają bezpieczną częstotliwość odcięcia:
f_cutoff = 1/(RC*2pi) = 1/(2*pi*330*10^(-9)*220) = 1/(6,28*7,26 * 10^(-5)) ≈ 2193 Hz >> 1 kHz
Połączono je zgodnie ze schematem:



W rzeczywistości wyglądało to następująco:



V_out wychodził na przetwornik A/C (pin PA1).

Generator sygnału zmiennego przy użyciu PWM i filtra RC:

Na początku należało zainicjalizować stałe i zmienne globalne:

```
static const bool GENERATE_SINE = true; // czy generować sinusa czy sygnał trójkątny

const float PWM_PERIOD = 1000;
const float PWM_CENTER = PWM_PERIOD / 2.0f;
static const int WAVE_SAMPLES = 100;
static const int WAVE_FREQ_DELAY = 10;   // 10us delay = 1kHz przy 100 próbkach
static uint16_t wave_table[WAVE_SAMPLES];
```

PWM_PERIOD oznacza okres generowanego PWM, natomiast WAVE_FREQ_DELAY jest tak ustawiony, żeby osiągnąć częstotliwość sygnału 1kHz (okres trwa 100*10µs = 1 ms).

Przy inicjalizacji programu oraz podczas zmian zadanej wartości napięcia międzyszczytowego generowano tablicę wartości sinusa przeskalowaną na wypełnienie okresu PWM dla zadanego Vpp:

```
   void update_sine_table(float vpp_mv) {
       if (vpp_mv > 3300.0f) vpp_mv = 3300.0f;
       if (vpp_mv < 0.0f) vpp_mv = 0.0f;
       float vpp_ratio = vpp_mv / 3300.0f;

       float pwm_amp = PWM_CENTER * vpp_ratio;

       for (int i = 0; i < WAVE_SAMPLES; i++) {
           // Kąt w radianach
           float angle = (2.0f * 3.14159f * i) / WAVE_SAMPLES;

           // Przeskalowanie do zakresu PWM
           wave_table[i] = (uint32_t)(PWM_CENTER + (sinf(angle) *
pwm_amp));
       }
   }
```

lub generowano tablicę dla sygnału o kształcie trójkątnym:

```
   void update_triangle_table(float vpp_mv) {
       if (vpp_mv > 3300.0f) vpp_mv = 3300.0f;
       if (vpp_mv < 0.0f) vpp_mv = 0.0f;
       float vpp_ratio = vpp_mv / 3300.0f;

       float pwm_amp = PWM_CENTER * vpp_ratio;

       for (int i = 0; i < WAVE_SAMPLES; i++) {
           float sample_val = 0.0f;

           // Pierwsza połowa (wznoszenie od -1 do 1)
           if (i < WAVE_SAMPLES / 2) {
               float progress = (float)i / (WAVE_SAMPLES / 2.0f);
               sample_val = -1.0f + (progress * 2.0f);
           }
           // Druga połowa (opadanie od 1 do -1)
           else {
               float progress = (float)(i - WAVE_SAMPLES / 2) /
(WAVE_SAMPLES / 2.0f);
               sample_val = 1.0f - (progress * 2.0f);
           }
```

```
            // Przeskalowanie do zakresu PWM
            wave_table[i] = (uint32_t)(PWM_CENTER + (sample_val *
pwm_amp));
        }
    }
```

Poniżej jest konfiguracja PWM, skorzystano w niej z TIM3 i wyprowadzono sygnał na pin PA6.

```
    void setup_pwm_generator() {
        LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
        LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM3);

        LL_GPIO_InitTypeDef gpio_init = {};
        gpio_init.Pin = LL_GPIO_PIN_6;
        gpio_init.Mode = LL_GPIO_MODE_ALTERNATE;
        gpio_init.Speed = LL_GPIO_SPEED_FREQ_HIGH;
        gpio_init.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
        gpio_init.Pull = LL_GPIO_PULL_NO;
        gpio_init.Alternate = LL_GPIO_AF_2;
        LL_GPIO_Init(GPIOA, &gpio_init);

        LL_TIM_SetPrescaler(TIM3, 0);
        LL_TIM_SetAutoReload(TIM3, PWM_PERIOD);
        LL_TIM_OC_SetMode(TIM3, LL_TIM_CHANNEL_CH1, LL_TIM_OCMODE_PWM1);
        LL_TIM_OC_SetCompareCH1(TIM3, 0);
        LL_TIM_OC_EnablePreload(TIM3, LL_TIM_CHANNEL_CH1);
        LL_TIM_CC_EnableChannel(TIM3, LL_TIM_CHANNEL_CH1);
        LL_TIM_EnableCounter(TIM3);
        LL_TIM_EnableAllOutputs(TIM3);
    }
```

Przetwornik A/C:

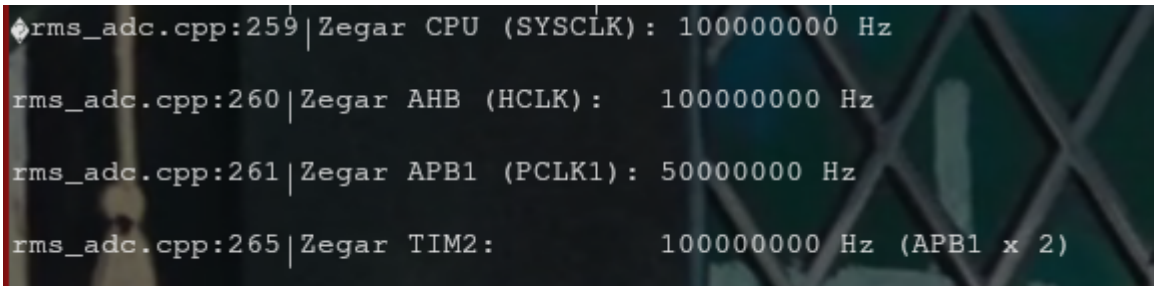
```
static const int ADC_BUF_SIZE = 1000;
volatile uint16_t adc_buffer[ADC_BUF_SIZE];
volatile int adc_buf_index = 0;
static isix::semaphore adc_ready_sem(0, 1);
```

Na początku należało skonfigurować, co uwzględnia konfigurację TIM2 do sprzętowego wyzwalania przetwornika z częstotliwością 48 kHz. Przetwornik był uruchamiany przez zbocze wzrastające wywołane przez TIM2. Jako wejście przetwornika służył pin GPIO PA1. Skonfigurowano także przerwania dla przetwornika, które zapisywały wartość do bufora i jeśli ten był pełny sygnalizowały to wątkowi przez semafor.

Ale najpierw należało sprawdzić częstotliwość zegara używając *dbprintf* w funkcji *main*:

```
   dbprintf("Zegar CPU (SYSCLK): %lu Hz\r\n",
rcc_clocks.SYSCLK_Frequency);
   dbprintf("Zegar AHB (HCLK):   %lu Hz\r\n",
rcc_clocks.HCLK_Frequency);
   dbprintf("Zegar APB1 (PCLK1): %lu Hz\r\n",
rcc_clocks.PCLK1_Frequency);

   if (rcc_clocks.PCLK1_Frequency != rcc_clocks.HCLK_Frequency) {
       dbprintf("Zegar TIM2:         %lu Hz (APB1 x 2)\r\n",
rcc_clocks.PCLK1_Frequency * 2);
   } else {
       dbprintf("Zegar TIM2:         %lu Hz (APB1 x 1)\r\n",
rcc_clocks.PCLK1_Frequency);
   }
```

odczytano takie wartości:



Czyli, żeby otrzymać na TIM2 częstotliwość 48 kHz to należy ustawić ARR na 2082, ponieważ:
ARR = f_clk_tim2 / 48 kHz - 1 = 100 MHz / 48 kHz - 1 ≈ 2083 - 1 = 2082 (dlatego też PWM period jest ustawiony na 2082)

```
void setup_adc_reader() {
      LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_ADC1);
      LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
      LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_TIM2);

      LL_GPIO_SetPinMode(GPIOA, LL_GPIO_PIN_1, LL_GPIO_MODE_ANALOG);

      LL_ADC_SetResolution(ADC1, LL_ADC_RESOLUTION_12B);
      LL_ADC_SetDataAlignment(ADC1, LL_ADC_DATA_ALIGN_RIGHT);

      LL_ADC_REG_SetTriggerSource(ADC1,
LL_ADC_REG_TRIG_EXT_TIM2_TRGO);

      ADC1->CR2 &= ~ADC_CR2_EXTEN;  // Wyczyszczenie bitów
      ADC1->CR2 |= ADC_CR2_EXTEN_0; // Ustawienie bitu 0 (Rising Edge)
```

```
        LL_ADC_REG_SetContinuousMode(ADC1, LL_ADC_REG_CONV_SINGLE);

        LL_ADC_REG_SetSequencerRanks(ADC1, LL_ADC_REG_RANK_1,
LL_ADC_CHANNEL_1);
        LL_ADC_SetChannelSamplingTime(ADC1, LL_ADC_CHANNEL_1,
LL_ADC_SAMPLINGTIME_15CYCLES);

        LL_ADC_EnableIT_EOCS(ADC1);

        NVIC_SetPriority(ADC_IRQn, 10);
        NVIC_EnableIRQ(ADC_IRQn);
        LL_ADC_Enable(ADC1);

        LL_TIM_SetPrescaler(TIM2, 0);
        LL_TIM_SetAutoReload(TIM2, 2082); // aby osiągnąć częstotliwość
48 Mhz
        LL_TIM_SetTriggerOutput(TIM2, LL_TIM_TRGO_UPDATE);
        LL_TIM_EnableCounter(TIM2);
    }
```

Obsługa przerwań była zrealizowana poprzez zapisywanie danych do bufora i jeśli bufor się wypełnił to sygnalizowano to poprzez semafor:

```
extern "C" {
    void adc_isr_vector(void) {
        if(LL_ADC_IsActiveFlag_EOCS(ADC1)) {
            LL_ADC_ClearFlag_EOCS(ADC1);

            uint16_t val = LL_ADC_REG_ReadConversionData12(ADC1);

            if(adc_buf_index < ADC_BUF_SIZE) {
                adc_buffer[adc_buf_index++] = val;
            } else {
                adc_buf_index = 0;
                adc_ready_sem.signal_isr();
            }
        }
    }
}
```

Należało także skonfigurować LEDy do wymaganej linijki:

```
   void setup_leds() {
       LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOD);

       LL_GPIO_InitTypeDef gpio_init = {};
       gpio_init.Pin = LL_GPIO_PIN_12 | LL_GPIO_PIN_13 | LL_GPIO_PIN_14
| LL_GPIO_PIN_15;
       gpio_init.Mode = LL_GPIO_MODE_OUTPUT;
       gpio_init.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
       gpio_init.Speed = LL_GPIO_SPEED_FREQ_LOW;
       gpio_init.Pull = LL_GPIO_PULL_NO;
       LL_GPIO_Init(GPIOD, &gpio_init);
   }
```

Zakresy napięć RMS dla zmiany świecącego się LEDa:

```
static const int led_thresholds[3] = {200, 400, 600};
```

W wątku zaimplementowano wyliczanie RMS na podstawie bufora, które oczekiwało na semafor. Po otrzymaniu na nim sygnału wyliczało RMS poprzez sumowanie kwadratów wartości przesuniętych o połowę Vdd zgodnie z poleceniem. Następnie liczono średnią kwadratową i skalowano ją na zakres napięcia. Ta funkcja modyfikowała też wartość Vpp dla różnych pomiarów wymaganych w zadaniu oraz dla różnych zakresów zmierzonego napięcia świeciła odpowiednimi ledami.

```
   void measure_task(void*) {
       setup_adc_reader();

       // Parametry zadania
       int current_vpp = 100;
       int step = 250;
       int max_vpp = 3000;
       int measurements_count = 0;

       // Ustawienie początkowe fali
       target_vpp_mv = current_vpp;
       if(GENERATE_SINE) update_sine_table(target_vpp_mv);
       else update_triangle_table(target_vpp_mv);

       dbprintf("=== START LABORATORIUM: %s ===", GENERATE_SINE ?
"SINUS" : "TROJKAT");
       isix::wait_ms(1000);

       for(;;) {
           if(adc_ready_sem.wait(ISIX_TIME_INFINITE) == ISIX_EOK) {
```

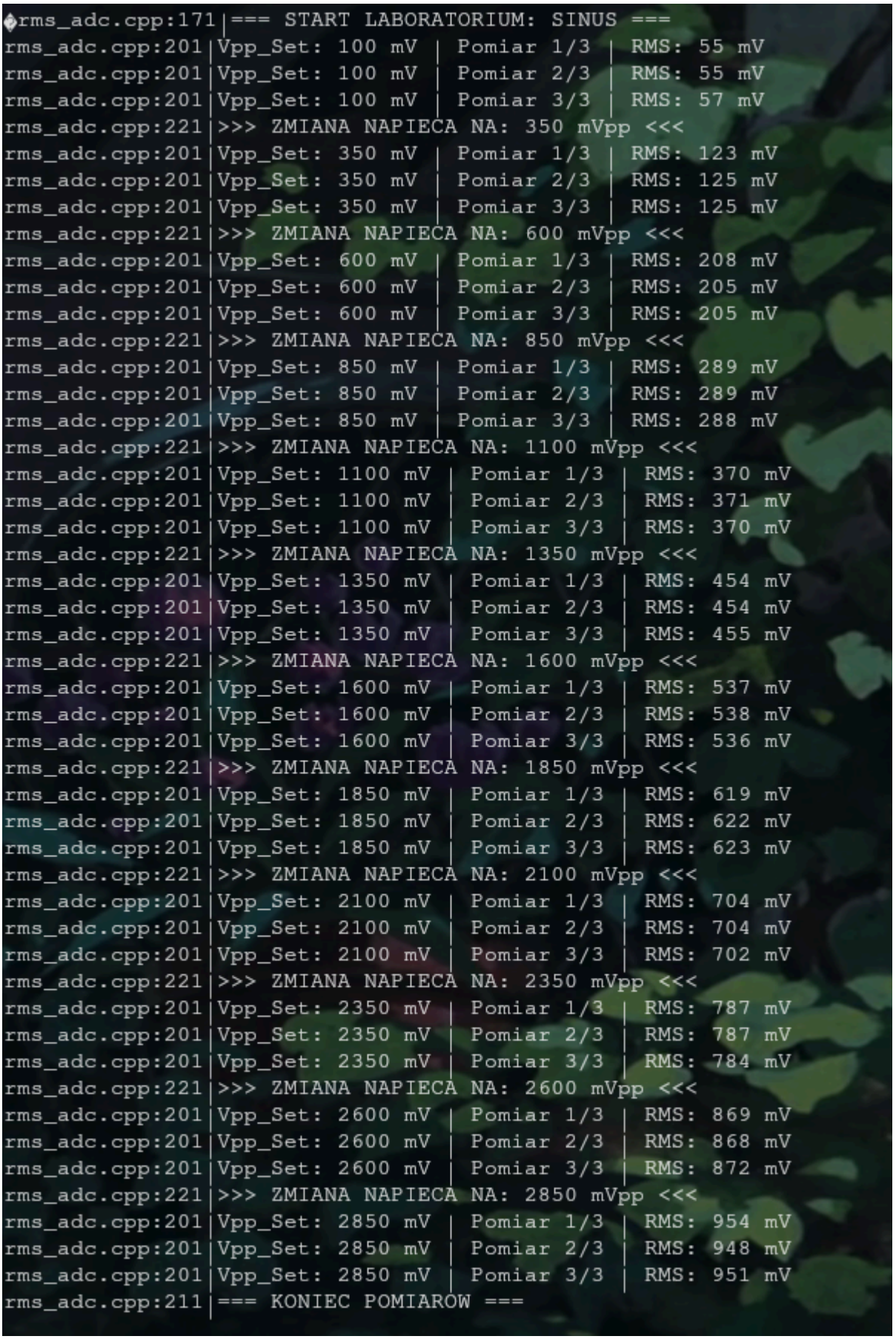
```
                long long sum_squares = 0;
                for(int i=0; i<ADC_BUF_SIZE; i++) {
                    int32_t val_ac = (int32_t)adc_buffer[i] - 2048;
//przesunięcie wartości, żeby oscylowała wokół 0 zamiast 1,65V - jak
zakłada RMS
                    sum_squares += (val_ac * val_ac);
                }
                float rms_raw = std::sqrt(sum_squares /
(float)ADC_BUF_SIZE);
                float rms_mv = (rms_raw * 3300.0f) / 4096.0f;
                int rms = (int)rms_mv;
                // Linijka zmierzonego napięcia na LEDach
                if (rms < led_thresholds[0]) {
                    LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_12);
                    LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_13 |
LL_GPIO_PIN_14 | LL_GPIO_PIN_15);
                }
                else if (rms < led_thresholds[1]) {
                    LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_13);
                    LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_12 |
LL_GPIO_PIN_14 | LL_GPIO_PIN_15);
                }
                else if (rms < led_thresholds[2]) {
                    LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_14);
                    LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_12 |
LL_GPIO_PIN_13 | LL_GPIO_PIN_15);
                }
                else {
                    LL_GPIO_SetOutputPin(GPIOD, LL_GPIO_PIN_15);
                    LL_GPIO_ResetOutputPin(GPIOD, LL_GPIO_PIN_12 |
LL_GPIO_PIN_13 | LL_GPIO_PIN_14);
                }
                dbprintf("Vpp_Set: %d mV | Pomiar %d/3 | RMS: %d mV",
                         target_vpp_mv, measurements_count + 1, rms);

                measurements_count++;

                if (measurements_count >= 3) {
                    measurements_count = 0;
                    current_vpp += step;

                    if (current_vpp > max_vpp) {
```

```
                    dbprintf("=== KONIEC POMIAROW ===");
                    isix::wait_ms(10000);
                    current_vpp = 100;
                }

                target_vpp_mv = current_vpp;

                if(GENERATE_SINE) update_sine_table(target_vpp_mv);
                else update_triangle_table(target_vpp_mv);

                dbprintf(">>> ZMIANA NAPIECA NA: %d mVpp <<<",
target_vpp_mv);
                isix::wait_ms(2000);
            } else {
                isix::wait_ms(500);
            }
        }
    }
}
```

Funkcja main wyglądała analogicznie do poprzednich laboratoriów.

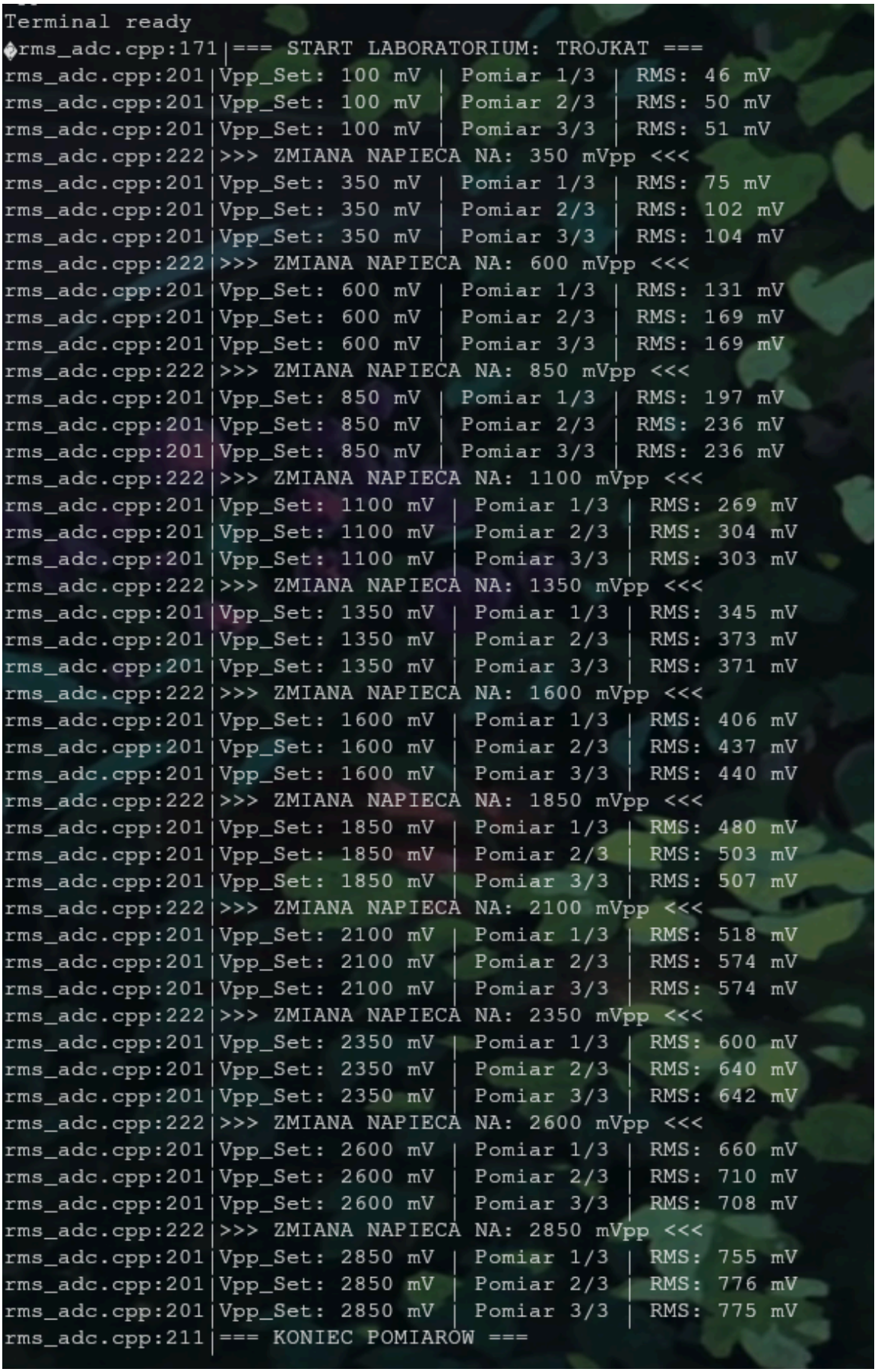
Program się skompilował i diody się świeciły. Zbadano także logi wyświetlające pomiary do 3 pomiarów wartości skutecznej napięcia dla przebiegu trójkątnego oraz sinusoidalnego o częstotliwości 1kHz w zakresie od 100mVpp do 3Vpp z krokiem 250mVpp.

Logi dla przebiegu sinusoidalnego:



Następnie dokonano pomiaru multimetrem **ANENG AN870 Pro**:
Vpp_Set: 100 mV TrueRMS: 30,1 mV
Vpp_Set: 350 mV TrueRMS: 100,5 mV
Vpp_Set: 600 mV TrueRMS: 180,2 mV
Vpp_Set: 850 mV TrueRMS: 254,7 mV
Vpp_Set: 1100 mV TrueRMS: 329,2 mV
Vpp_Set: 1350 mV TrueRMS: 404,1 mV
Vpp_Set: 1600 mV TrueRMS: 478,2 mV
Vpp_Set: 1850 mV TrueRMS: 552,6 mV
Vpp_Set: 2100 mV TrueRMS: 626,8 mV
Vpp_Set: 2350 mV TrueRMS: 700,9 mV
Vpp_Set: 2600 mV TrueRMS: 775,1 mV
Vpp_Set: 2850 mV TrueRMS: 848,8 mV

Dla sygnału trójkątnego analogicznie dokonano pomiarów:



Wartości zmierzone multimetrem podczas tych samych testów:
Vpp_Set: 100 mV TrueRMS: 24,2 mV
Vpp_Set: 350 mV TrueRMS: 85,4 mV
Vpp_Set: 600 mV TrueRMS: 146,4 mV
Vpp_Set: 850 mV TrueRMS: 207,2 mV
Vpp_Set: 1100 mV TrueRMS: 268 mV
Vpp_Set: 1350 mV TrueRMS: 328,5 mV
Vpp_Set: 1600 mV TrueRMS: 389 mV
Vpp_Set: 1850 mV TrueRMS: 449,5 mV
Vpp_Set: 2100 mV TrueRMS: 509,8 mV
Vpp_Set: 2350 mV TrueRMS: 570,2 mV
Vpp_Set: 2600 mV TrueRMS: 630,3 mV
Vpp_Set: 2850 mV TrueRMS: 690,4 mV

Wartości teoretyczne (ze wzorów):

dla sygnału sinus: $V\_RMS = Vpp/(2*\sqrt{2}) \approx Vpp * 0{,}35355$

dla sygnału trójkątnego: $V\_RMS = Vpp/(2*\sqrt{3}) \approx Vpp * 0{,}28867$

Wyniki przedstawiono w tabeli:

Napięcie zadane Vpp	**Sinus: Teoria**	**Sinus: STM32 (uśrednione)**	**Sinus: Multimetr**	**Trójkąt: Teoria**	**Trójkąt: STM32 (uśrednione)**	**Trójkąt: Multimetr**
[mV]	**RMS [mV]**	**RMS [mV]**	**RMS [mV]**	**RMS [mV]**	**RMS [mV]**	**RMS [mV]**
100	35,4	56	30,1	28,9	49	24,2
350	123,7	124	100,5	101,0	94	85,4
600	212,1	206	180,2	173,2	156	146,4
850	300,5	289	254,7	245,4	223	207,2

1100	388,9	370	329,2	317,5	292	268,0
1350	477,3	454	404,1	389,7	363	328,5
1600	565,7	537	478,2	461,9	427	389,0
1850	654,1	621	552,6	534,0	496	449,5
2100	742,5	703	626,8	606,2	555	509,8
2350	830,8	786	700,9	678,4	627	570,2
2600	919,2	870	775,1	750,5	692	630,3
2850	1007,6	951	848,8	822,7	769	690,4

Następnie policzono błędy:
Błąd bezwzględny = |RMS_STM - RMS_MULTIMETR|
Błąd względny = |RMS_STM - RMS_MULTIMETR| / RMS_MULTIMETR * 100%

Napięcie zadane Vpp [mV]	**Sinus** **Błąd bezwzgl.**	**Sinus** **Błąd wzgl. [%]**
100	25,9	86,05

350	23,5	23,38
600	25,8	14,32
850	34,3	13,47
1100	40,8	12,39
1350	49,9	12,35
1600	58,8	12,3
1850	68,4	11,01
2100	76,2	11,84
2350	85,1	12,14
2600	94,9	12,24
2850	102,2	12,04

Niepewność wzorcowania (wyznaczona dla sygnału sinusoidalnego):
Dla wyników z STM32 wybrano pomiary dla jednej z wartości Vpp i na ich podstawie wyliczono niepewność:
Dla Vpp = 2850mV zmierzono 954, 948, 951 mV, które dają średnią 951 mV

odchylenie standardowe = $\sqrt{((951 - 954)^2 + (951 - 948)^2 + (951 - 951)^2)/2} = \sqrt{18/2} = 3$ mV

niepewność standardowa = 3 mV / $\sqrt{3} \approx 1{,}73$ mV

Dla miernika sprawdzono dokładność w dokumentacji i wynosiła +/-0,3% + 3.
Czyli np. dla pomiaru 848,8 mV (dla Vpp=2850 mV):
błąd graniczny = (0,003 * 848,8) + (3 * 0,1) = 2,85 mV

niepewność standardowa = 2,85 mV / $\sqrt{3} \approx 1{,}65$ mV

Jaki układ analogowy należy zastosować na wejściu przetwornika A/C gdybyśmy chcieli opracować woltomierz True RMS, a których zabrakło podczas realizacji ćwiczenia?
Należałoby mieć układu sumującego ze składową stałą (DC offset), ponieważ prawdziwe sygnały przemienne AC przyjmują wartości ujemne, których nasza płytka STM32 nie obsługuje.

Wnioski:
Z powodu braku dokładnego generatora sygnału AC osiągnięte wyniki są mało miarodajne. Widać, że wartości zmierzone na STM32 były bliższe wartościom teoretycznym, ale te nie uwzględniały tłumienia wnoszonego przez rzeczywisty filtr RC. Więc multimetr podawał wartości dokładniejsze, najprawdopodobniej dlatego że eliminował szum z PWM. Jeśli chodzi o STM32F411 Discovery to zabrakło przetwornika C/A, który pozwoliłby wygenerować sygnał AC.