Zuzanna Godek 318373
MARM sprawozdanie z laboratorium 4

Celem laboratorium było zapoznanie się z protokołami transmisji szeregowej I2C oraz SPI w mikrokontrolerach rodziny STM32.

Zadanie 1.1. Obsługa magistrali SPI, żyroskop L3GD20
Celem zadania było zaimplementowanie programu wykorzystującego bibliotekę systemu ISIX do obsługi magistrali SPI do:

- Odczytywania stanu odchylenia płytki (sensora L3GD20) w osiach X,Y,Z
- Wypisywania stanu odchylenia na osiach X,Y, Z na diagnostycznym porcie szeregowym (dbprintf)
- Wizualizacji stanu odchylenia płytki czujnika w osiach X,Y za pomocą diod: LD3 ... LD6

Jako graniczną wartość odchylenia włączającą daną diodę LED można było przyjąć wartość bezwzględną 500 odczytaną z rejestrów OUT_X oraz OUT_Y.

Na początku programu są zdefiniowane stałe dla większej przejrzystości kodu:

```
namespace {
   constexpr auto led_green  = periph::gpio::num::PD12; // LD4
   constexpr auto led_orange = periph::gpio::num::PD13; // LD3
   constexpr auto led_red    = periph::gpio::num::PD14; // LD5
   constexpr auto led_blue   = periph::gpio::num::PD15; // LD6

   constexpr int16_t THR = 500; // wartość bezwzgl odchylenia
   constexpr int CS_ADDR = 0;

   struct reg {
       static constexpr uint8_t WHO_AM_I      = 0x0F;
       static constexpr uint8_t CTRL_REG1     = 0x20;
       static constexpr uint8_t CTRL_REG2     = 0x21;
       static constexpr uint8_t CTRL_REG3     = 0x22;
       static constexpr uint8_t CTRL_REG4     = 0x23;
       static constexpr uint8_t CTRL_REG5     = 0x24;
       static constexpr uint8_t OUT_X_L       = 0x28;
       static constexpr uint8_t OUT_X_H       = 0x29;
       static constexpr uint8_t OUT_Y_L       = 0x2A;
       static constexpr uint8_t OUT_Y_H       = 0x2B;
       static constexpr uint8_t OUT_Z_L       = 0x2C;
       static constexpr uint8_t OUT_Z_H       = 0x2D;

   };
```

```
   // Skorzystano z globalnej zmiennej spi, żeby nie musieć przekazywać
zmiennej do funkcji
   periph::drivers::spi_master* spi = nullptr;
```

Następnie widnieją funkcje inicjalizujące oraz obsługujące diody:

```
void leds_init() {
       const auto mode_out = periph::gpio::mode::out {
           periph::gpio::outtype::pushpull,
           periph::gpio::speed::low
       };

       periph::gpio::setup(led_green,  mode_out);
       periph::gpio::setup(led_orange, mode_out);
       periph::gpio::setup(led_red,    mode_out);
       periph::gpio::setup(led_blue,   mode_out);
   }
void leds_update(int16_t x, int16_t y) {

       // Oś X: Green (PD12) / Red (PD14)
       if (x > THR) {
           periph::gpio::set(led_green, true);
           periph::gpio::set(led_red,   false);
       } else if (x < -THR) {
           periph::gpio::set(led_green, false);
           periph::gpio::set(led_red,   true);
       } else {
           periph::gpio::set(led_green, false);
           periph::gpio::set(led_red,   false);
       }

       // Oś Y: Orange (PD13) / Blue (PD15)
       if (y > THR) {
           periph::gpio::set(led_blue,   true);
           periph::gpio::set(led_orange, false);
       } else if (y < -THR) {
           periph::gpio::set(led_blue,   false);
           periph::gpio::set(led_orange, true);
       } else {
           periph::gpio::set(led_blue,   false);
           periph::gpio::set(led_orange, false);
       }
   }
```

Później są funkcję inicjalizujące magistralę SPI oraz żyroskop L3GD20. Ustawiono parametry magistrali SPI z instrukcji, tak samo ustawiono parametry żyroskopu L3GD20 zgodnie z poleceniem:

```
int spi_init_isix() {
    namespace opt = periph::option;
    int ret = 0;
    if (spi == nullptr) {
        spi = new periph::drivers::spi_master("spi1");
    }
    do {
        // 8 bitów danych
        if((ret=spi->set_option(opt::dwidth(8)))<0) break;
        // Pierwszy bit najbardziej znaczący (MSB)
        if((ret=spi->set_option(opt::bitorder(
                                opt::bitorder::msb)))<0) break;
        // Częstotliwość 10Mhz
        if((ret=spi->set_option(opt::speed(10E6)))<0) break;
        // Próbkowanie danych na pierwszym zboczu zegara
        if((ret=spi->set_option(opt::phase(
                                opt::phase::_1_edge)))<0) break;
        // Próbkowanie danych przy zmianie z niskiego na wysoki
        if((ret=spi->set_option(opt::polarity(
                                opt::polarity::low)))<0) break;
        if((ret=spi->open(ISIX_TIME_INFINITE))<0) break;
    } while(0);
    return ret;
}

int l3gd20_init_isix() {
    uint8_t id = 0;

    if (spi_read_reg_isix(reg::WHO_AM_I, id) != 0) {
        return -1;
    }
    dbprintf("L3GD20 ID: 0x%02X", id);

    if (id != 0xD4 && id != 0xD3 && id != 0xD7) {
        return -2;
    }

    int ret = 0;
    // Wartości z instrukcji
    ret |= spi_write_reg_isix(reg::CTRL_REG1, 0x0F);
```

```
        ret |= spi_write_reg_isix(reg::CTRL_REG2, 0x00);
        ret |= spi_write_reg_isix(reg::CTRL_REG3, 0b00001000);
        ret |= spi_write_reg_isix(reg::CTRL_REG4, 0b00110000);
        ret |= spi_write_reg_isix(reg::CTRL_REG5, 0x00);
        return ret;
    }
```

Poniżej są funkcje odpowiedzialne z komunikację z żyroskopem przy użyciu SPI:

```
    int spi_write_reg_isix(uint8_t addr, uint8_t val) {
        uint8_t buf[] = { addr, val };
        periph::blk::tx_transfer tran(buf, sizeof(buf));
        return spi->transaction(CS_ADDR, tran);
    }

    int spi_read_reg_isix(uint8_t addr, uint8_t& val) {
        uint8_t tx[] = { static_cast<uint8_t>(addr | 0x80), 0x00 };
        uint8_t rx[2] = {};

        periph::blk::trx_transfer tran(tx, rx, sizeof(tx));
        int ret = spi->transaction(CS_ADDR, tran);

        if (ret == 0) {
            val = rx[1];
        }
        return ret;
    }
```

Natomiast tutaj jest funkcja odpowiedzialna za odczytywanie pozycji odchylenia z dwóch rejestrów 8 bitowych i łączenia tych wartości na 16 bitów:

```
int l3gd20_read_all_isix(int16_t& x, int16_t& y, int16_t& z) {
        uint8_t xl, xh, yl, yh, zl, zh;
        int ret = 0;

        ret |= spi_read_reg_isix(reg::OUT_X_L, xl);
        ret |= spi_read_reg_isix(reg::OUT_X_H, xh);
        ret |= spi_read_reg_isix(reg::OUT_Y_L, yl);
        ret |= spi_read_reg_isix(reg::OUT_Y_H, yh);
        ret |= spi_read_reg_isix(reg::OUT_Z_L, zl);
        ret |= spi_read_reg_isix(reg::OUT_Z_H, zh);

        if(ret != 0) return ret;

        x = static_cast<int16_t>((xh << 8) | xl);
```

```
        y = static_cast<int16_t>((yh << 8) | yl);
        z = static_cast<int16_t>((zh << 8) | zl);
        return 0;
    }
```

Tak wygląda pętla łącząca zaimplementowane wcześniej funkcje i wypisująca pozycję odczytaną z żyroskopu:

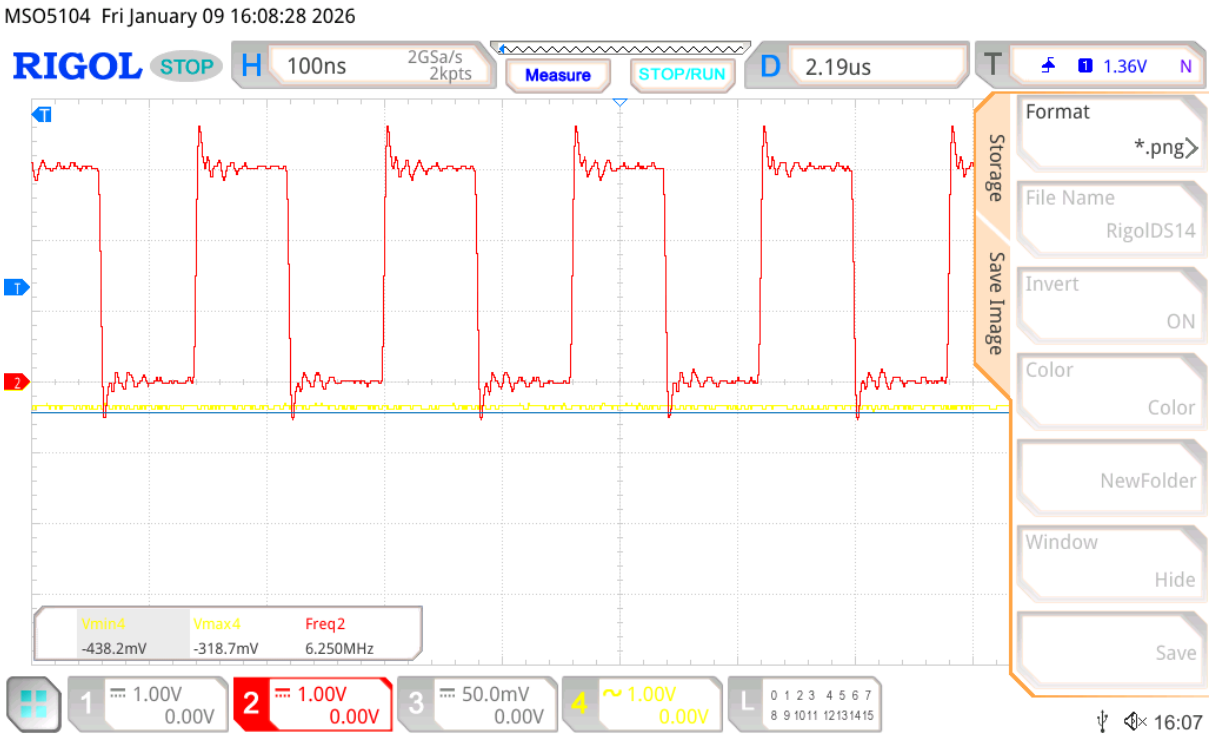
```
void watch_sensor(void*) {
        int ret {};

        if ((ret = spi_init_isix()) < 0) {
            dbprintf("SPI init failed: %d", ret);
            return;
        }

        if ((ret = l3gd20_init_isix()) != 0) {
            dbprintf("Gyro init failed: %d", ret);
            return;
        }

        dbprintf("SPI and Gyro init OK. Loop start.");

        do {
            while(true) {
                int16_t x, y, z;
                if (l3gd20_read_all_isix(x, y, z) == 0) {
                    dbprintf("X:%5d Y:%5d Z:%5d", x, y, z);
                    leds_update(x, y);
                } else {
                    dbprintf("Error reading axes");
                }
                //Cykl programu 250ms zgodnie z poleceniem
                isix::wait_ms(250);
            }

        } while(0);

        dbg_info("Task failed finished with code %i", ret);
    }
```

Zweryfikowano, że program działa poprawnie i diody zapalają się w dobrym momencie.

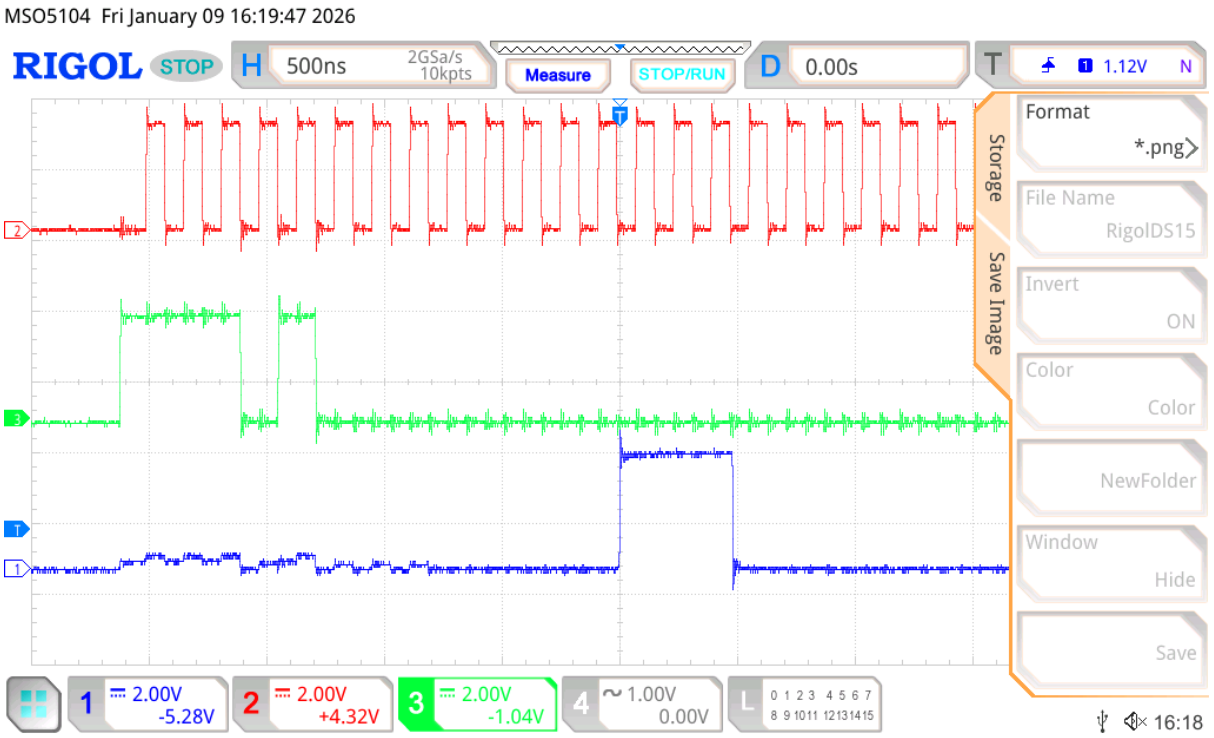
Odczytano także wartości wysyłane protokołem USART przy użyciu programu picocom:

Program łatwo przepisać na inną bibliotekę, ponieważ logika została wydzielona do osobnych funkcji, więc w głównej pętli programu można po prostu podmienić nazwy na funkcje realizujące to samo zadanie tylko zaimplementowane np. w bibliotece LL.

Otrzymano oscylogramy sygnału magistrali SPI od kolegi (Adama Rybojada).
Pierwszy oscylogram zawierał tylko sygnał zegara (czerwony) i sygnał CS (tutaj tylko w stanie niskim, czyli w trakcie transmisji):



Na kolejnym oscylogramie widnieje sygnał zegarowy na czerwono, linia MOSI na zielono i MISO na niebiesko:



Na zdjęciu widać, że stan wysoki na linii MOSI pojawia się przed pierwszym narastającym zboczem zegara. Jest to spowodowane tym, że master musi wystawić dane wcześniej, żeby slave z tym narastającym zboczem mógł już je odczytywać.

Zadanie 1.2. Obsługa magistrali I2C, akcelerometr LMS303

W tym zadaniu należało napisać program odczytujący po magistrali I2C1 przyspieszenie w osiach X, Y i Z w cyklu 500 ms.

Analogicznie na początku zdefiniowano stałe i zmienne globalne:

```
namespace {
   static constexpr int LSM303_ACC_ADDR = 0x32;

   static constexpr uint8_t CTRL_REG1_A = 0x20;
   static constexpr uint8_t OUT_X_L_A   = 0x28;
   static constexpr uint8_t AUTO_INCREMENT = 0x80;

   periph::drivers::i2c_master* i2c = nullptr;
```

Następnie potrzebne są funkcje inicjalizujące magistralę I2C i akcelerometr LSM303:

```
   int i2c_init_isix() {
       i2c = new periph::drivers::i2c_master("i2c1");
       static constexpr auto IFC_TIMEOUT = 1000;
       int ret = i2c->open(IFC_TIMEOUT);
       if(ret) {
           dbg_err("Unable to open i2c device error: %i", ret);
           return ret;
       }

       ret = i2c->set_option(periph::option::speed(400'000));
       if(ret) {
           dbg_err("Unable to set i2c option error: %i", ret);
           return ret;
       }
       return 0;
   }
   int lsm303_init() {
       return i2c_write_reg_isix(LSM303_ACC_ADDR, CTRL_REG1_A, 0x27);
   }
```

Pisanie i odczytywanie danych na I2C działało następująco:

```
   int i2c_write_reg_isix(uint8_t dev_addr, uint8_t reg_addr, uint8_t
value) {
       uint8_t buf[] = { reg_addr, value };
       periph::blk::tx_transfer tran(buf, sizeof(buf));
       return i2c->transaction(dev_addr, tran);
   }
```

```
   int i2c_read_regs_isix(uint8_t dev_addr, uint8_t reg_addr, uint8_t*
rx_buffer, size_t len) {
       uint8_t tx_buf[] = { reg_addr };
       periph::blk::trx_transfer tran(tx_buf, rx_buffer,
sizeof(tx_buf), len);
       return i2c->transaction(dev_addr, tran);
   }
```

Funkcja odczytująca przyspieszenie w osi X, Y i Z była następująca:

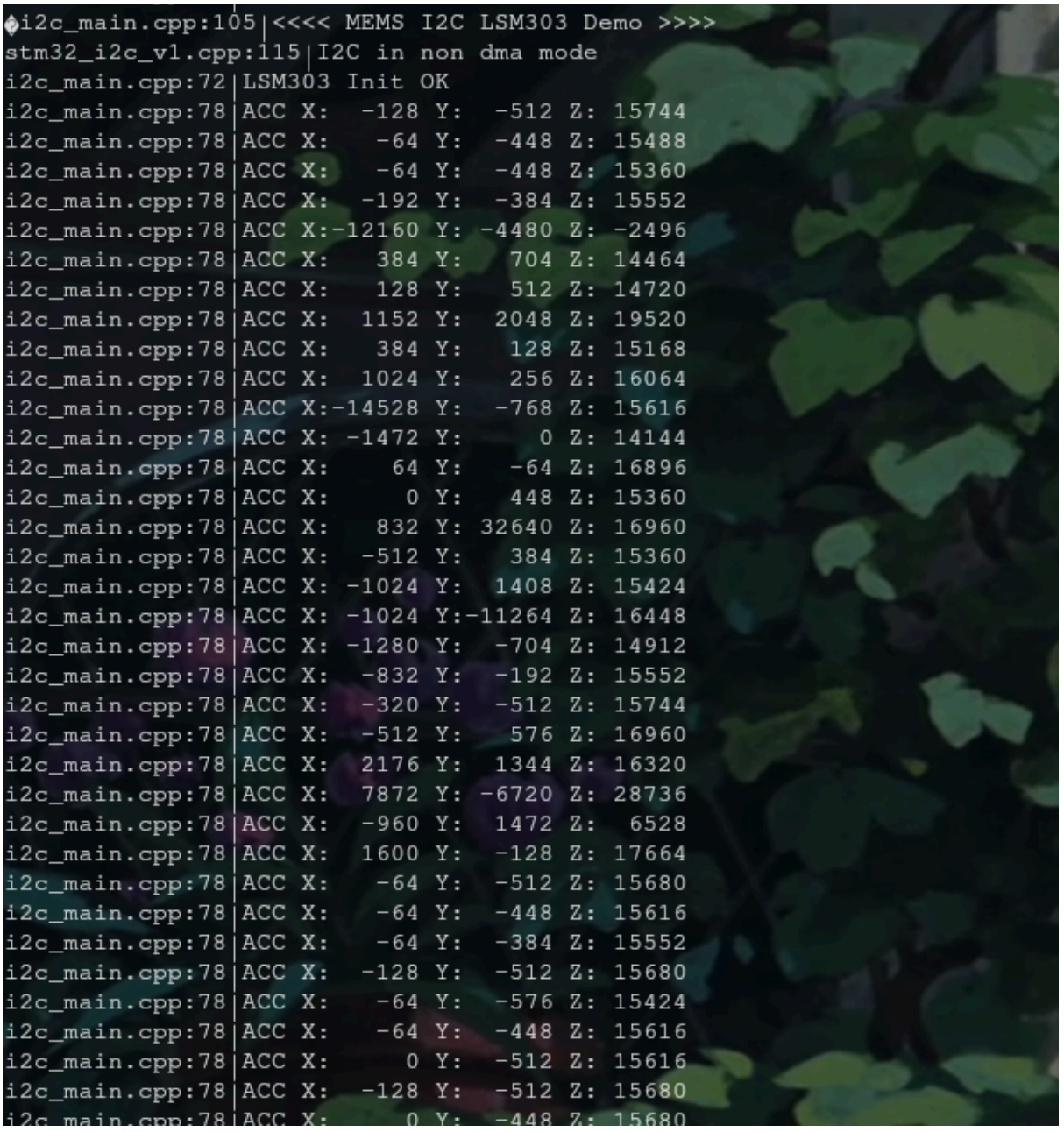
```
   int lsm303_read_accel(int16_t& x, int16_t& y, int16_t& z) {
       uint8_t buf[6] = {};
       int ret = i2c_read_regs_isix(LSM303_ACC_ADDR, OUT_X_L_A |
AUTO_INCREMENT, buf, 6);
       if (ret != 0) return ret;

       x = static_cast<int16_t>((buf[1] << 8) | buf[0]);
       y = static_cast<int16_t>((buf[3] << 8) | buf[2]);
       z = static_cast<int16_t>((buf[5] << 8) | buf[4]);
       return 0;
   }
```

Główna pętla programu była analogiczna do tej z poprzedniego programu:

```
   void watch_sensor(void*) {
       int ret = 0;

       if ((ret = i2c_init_isix()) != 0) {
           dbg_err("I2C init failed: %i", ret);
           return;
       }

       if ((ret = lsm303_init()) != 0) {
           dbg_err("LSM303 init failed: %i", ret);
       } else {
           dbprintf("LSM303 Init OK");
       }

       while (true) {
           int16_t x, y, z;
           if (lsm303_read_accel(x, y, z) == 0) {
               dbprintf("ACC X:%6d Y:%6d Z:%6d", x, y, z);
           }
```

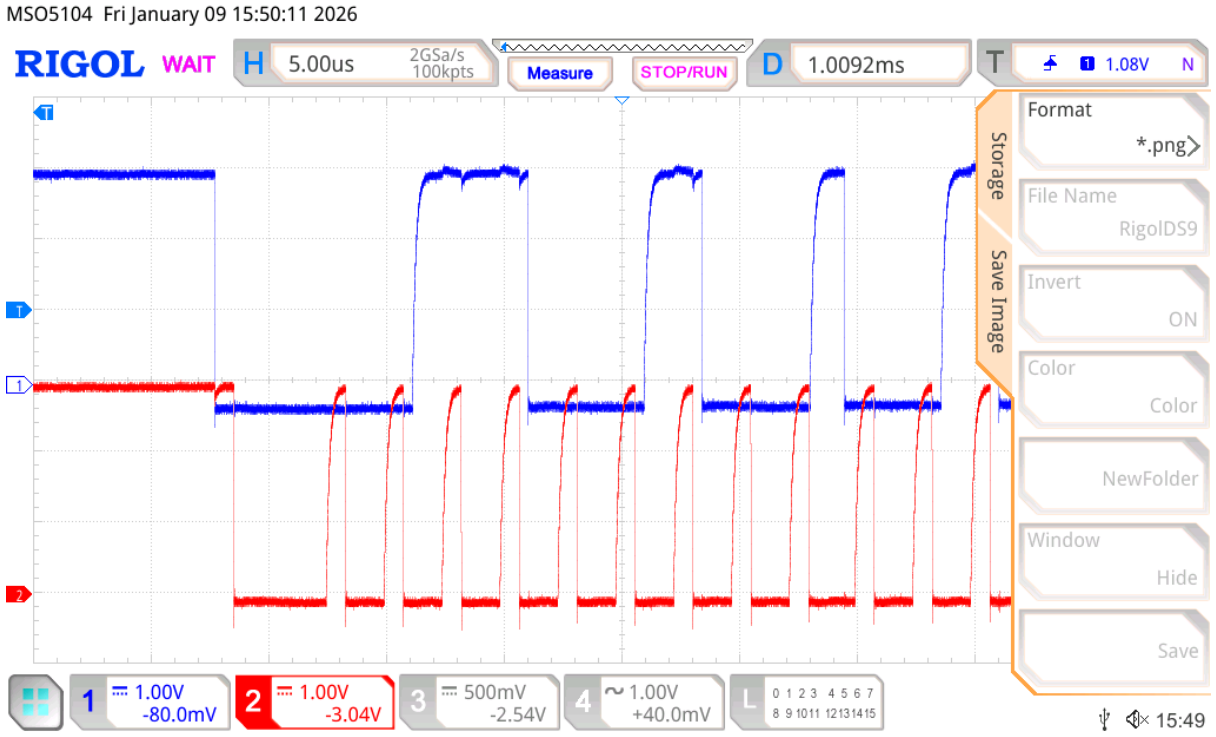
```
            //Cykl programu 500 ms zgodnie z poleceniem
            isix::wait_ms(500);
        }
    }
```

Zweryfikowano działanie programu na konsoli do której odczytane wartości przyspieszenia w osi X, Y i Z były wysyłane po protokole USART:



Po uruchomieniu poruszano płytkę w różne strony (najpierw w osi Y, a potem X) a na końcu położono ją na stole. W odczytach widać dużą niepewność pomiarową (w spoczynku wartości się wahają od -100 do 500 pomimo braku ruchu). Wartość przyspieszenia w osi Z wynosi około 16000 co odpowiada przyspieszeniu grawitacyjnemu. To potwierdza obserwację, że testy przeprowadzone były na Ziemi, a nie w przestrzeni kosmicznej.

Tym razem także otrzymano oscylogram od kolegi (linia czerwona to zegar, niebieska to SDA):



Widać, że SDA jest ustawione na stan niski chwilę przed uruchomieniem zegara, analogicznie jak przy magistrali SPI.

Zadanie 1.3. Obsługa magistrali SPI oraz I2C z wykorzystaniem bibliotek LL

SPI:

Na początku zdefiniowano kilka stałych dla przejrzystości kodu:

```
#define SPI_SCK_PIN   LL_GPIO_PIN_5
#define SPI_MISO_PIN  LL_GPIO_PIN_6
#define SPI_MOSI_PIN  LL_GPIO_PIN_7
```

Potem podmieniono następujące funkcje inicjalizujące SPI i żyroskop:

```
int spi_init_ll() {
    //Włączenie zegarów
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOA);
    LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOE);
    LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_SPI1);

    //Inicjalizacja struktur
    LL_SPI_InitTypeDef SPI_InitStruct;
    LL_GPIO_InitTypeDef GPIO_InitStruct;
```

```
	LL_SPI_StructInit(&SPI_InitStruct);
	LL_GPIO_StructInit(&GPIO_InitStruct);

	//Konfiguracja pinu CS (PE3) jako Output Push-Pull
	LL_GPIO_SetOutputPin(GPIOE, LL_GPIO_PIN_3); // CS domyślnie w
stanie wysokim

	GPIO_InitStruct.Pin = LL_GPIO_PIN_3;
	GPIO_InitStruct.Mode = LL_GPIO_MODE_OUTPUT;
	GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_HIGH;
	GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
	GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
	LL_GPIO_Init(GPIOE, &GPIO_InitStruct);

	//Konfiguracja pinów SPI (PA5, PA6, PA7) jako Alternate Function
	GPIO_InitStruct.Pin = SPI_SCK_PIN | SPI_MISO_PIN | SPI_MOSI_PIN;
	GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
	GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
	GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
	GPIO_InitStruct.Pull = LL_GPIO_PULL_NO;
	GPIO_InitStruct.Alternate = LL_GPIO_AF_5;
	LL_GPIO_Init(GPIOA, &GPIO_InitStruct);

	//Konfiguracja parametrów SPI zgodnie z instrukcją
	SPI_InitStruct.TransferDirection = LL_SPI_FULL_DUPLEX;
	SPI_InitStruct.Mode = LL_SPI_MODE_MASTER;
	SPI_InitStruct.DataWidth = LL_SPI_DATAWIDTH_8BIT;
	SPI_InitStruct.ClockPolarity = LL_SPI_POLARITY_LOW;
	SPI_InitStruct.ClockPhase = LL_SPI_PHASE_1EDGE;
	SPI_InitStruct.NSS = LL_SPI_NSS_SOFT; //NSS wyłączony, 'ręczne'
sterowanie CS
	SPI_InitStruct.BaudRate = LL_SPI_BAUDRATEPRESCALER_DIV16;
//preskaler tak żeby zegar był poniżej 10 Mhz (100/16 = 6,25 Mhz)
	SPI_InitStruct.BitOrder = LL_SPI_MSB_FIRST;
	SPI_InitStruct.CRCCalculation = LL_SPI_CRCCALCULATION_DISABLE;

	if (LL_SPI_Init(SPI1, &SPI_InitStruct) != SUCCESS) {
	    return -1;
	}

	LL_SPI_Enable(SPI1);

	return 0;
```

```
}
```

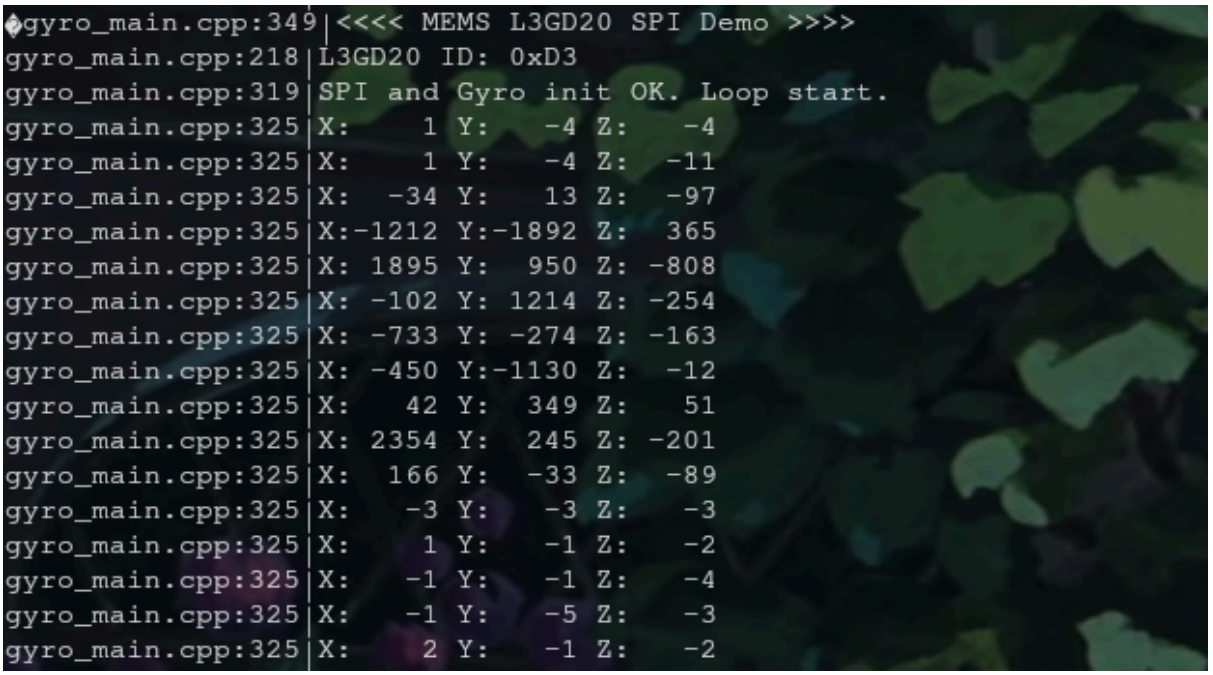
```
int l3gd20_init_ll() {
    uint8_t id = 0;

    spi_read_reg_ll(reg::WHO_AM_I, id);
    dbprintf("L3GD20 ID: 0x%02X", id);

    if (id != 0xD4 && id != 0xD3 && id != 0xD7) {
        return -2;
    }
    //Wartości z instrukcji
    spi_write_reg_ll(reg::CTRL_REG1, 0x0F);
    spi_write_reg_ll(reg::CTRL_REG2, 0x00);
    spi_write_reg_ll(reg::CTRL_REG3, 0b00001000);
    spi_write_reg_ll(reg::CTRL_REG4, 0b00110000);
    spi_write_reg_ll(reg::CTRL_REG5, 0x00);
    return 0;
}
```

Natomiast pisanie i odczytywanie z magistrali SPI wyglądało następująco:

```
uint8_t spi_ll_transfer_byte(uint8_t data) {
    while (!LL_SPI_IsActiveFlag_TXE(SPI1)) {}

    LL_SPI_TransmitData8(SPI1, data);

    while (!LL_SPI_IsActiveFlag_RXNE(SPI1)) {}

    return LL_SPI_ReceiveData8(SPI1);
}

int spi_write_reg_ll(uint8_t addr, uint8_t val) {
    //CS ustawiony na 0 -> inicjalizacja rozmowy z żyroskopem
    LL_GPIO_ResetOutputPin(GPIOE, LL_GPIO_PIN_3);

    spi_ll_transfer_byte(addr);
    spi_ll_transfer_byte(val);

    LL_GPIO_SetOutputPin(GPIOE, LL_GPIO_PIN_3);
    //CS ustawiony na 1 -> koniec rozmowy z żyroskopem
    return 0;
}
```

```
int spi_read_reg_ll(uint8_t addr, uint8_t& val) {
    LL_GPIO_ResetOutputPin(GPIOE, LL_GPIO_PIN_3);

    spi_ll_transfer_byte(addr | 0x80);
    val = spi_ll_transfer_byte(0x00);

    LL_GPIO_SetOutputPin(GPIOE, LL_GPIO_PIN_3);
    return 0;
}
```

Reszta kodu pozostała bez zmian, poza zmianą nazw wywoływanych funkcji.
Uruchomiono program i zweryfikowano, że diody świecą się poprawnie a w picocom odczytano wartości X, Y i Z:



I2C:
Na początku trzeba było dodać więcej stałych:

```
#define I2C_INSTANCE    I2C1
#define I2C_GPIO_PORT   GPIOB
#define I2C_SCL_PIN     LL_GPIO_PIN_6
#define I2C_SDA_PIN     LL_GPIO_PIN_9
#define I2C_AF          LL_GPIO_AF_4
```

Zmodyfikowano inicjalizację I2C oraz akcelerometru:

```
int i2c_init_ll() {
      LL_AHB1_GRP1_EnableClock(LL_AHB1_GRP1_PERIPH_GPIOB);
      LL_APB1_GRP1_EnableClock(LL_APB1_GRP1_PERIPH_I2C1);
```

```
        LL_I2C_InitTypeDef I2C_InitStruct;
        LL_GPIO_InitTypeDef GPIO_InitStruct;

        // Konfiguracja GPIO (Open-Drain i Pull-Up)
        LL_GPIO_StructInit(&GPIO_InitStruct);
        GPIO_InitStruct.Pin = I2C_SCL_PIN | I2C_SDA_PIN;
        GPIO_InitStruct.Mode = LL_GPIO_MODE_ALTERNATE;
        GPIO_InitStruct.Speed = LL_GPIO_SPEED_FREQ_VERY_HIGH;
        GPIO_InitStruct.OutputType = LL_GPIO_OUTPUT_OPENDRAIN; //Open
drain jest kluczowy, ponieważ komunikacja jest w obie strony poprzez
SDA
        GPIO_InitStruct.Pull = LL_GPIO_PULL_UP; // Wewnętrzne pull-upy
        GPIO_InitStruct.Alternate = I2C_AF;
        LL_GPIO_Init(I2C_GPIO_PORT, &GPIO_InitStruct);

        // Konfiguracja I2C
        LL_I2C_Disable(I2C_INSTANCE);
        LL_I2C_StructInit(&I2C_InitStruct);
        I2C_InitStruct.PeripheralMode = LL_I2C_MODE_I2C;
        I2C_InitStruct.ClockSpeed = 400000;
        I2C_InitStruct.DutyCycle = LL_I2C_DUTYCYCLE_2;
        I2C_InitStruct.OwnAddress1 = 0;
        I2C_InitStruct.TypeAcknowledge = LL_I2C_ACK;
        I2C_InitStruct.OwnAddrSize = LL_I2C_OWNADDRESS1_7BIT;

        if (LL_I2C_Init(I2C_INSTANCE, &I2C_InitStruct) != SUCCESS) {
            return -1;
        }

        LL_I2C_Enable(I2C_INSTANCE);
        return 0;
    }
```

```
    int lsm303_init_ll() {
        return i2c_write_reg_ll(LSM303_ACC_ADDR, CTRL_REG1_A, 0x27);
    }
```

Najbardziej skomplikowane okazały się funkcję wysyłania i odbierania danych, ponieważ należało zrealizować całą, długą sekwencję komunikacji:

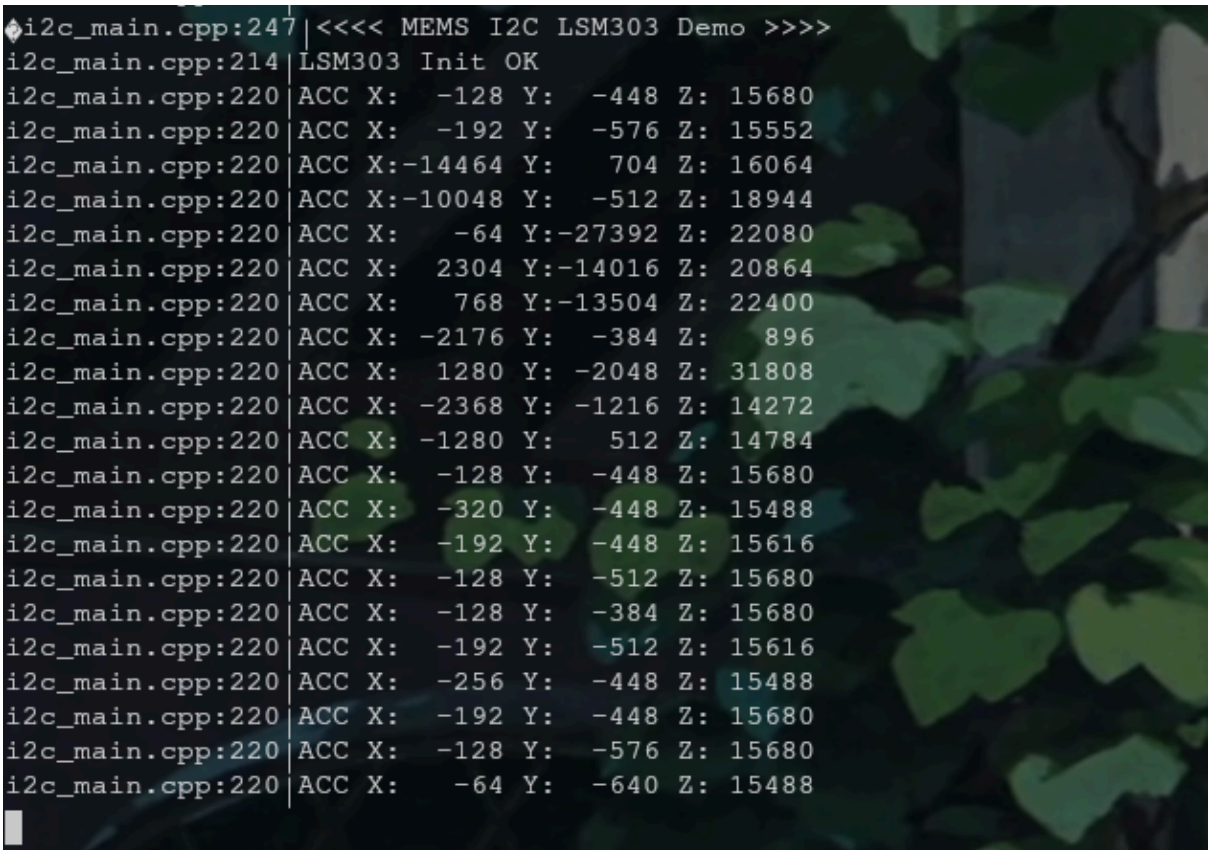
```
int i2c_write_reg_ll(uint8_t dev_addr, uint8_t reg_addr, uint8_t value)
{
        // 1. START
```

```
        LL_I2C_GenerateStartCondition(I2C_INSTANCE);
        while (!LL_I2C_IsActiveFlag_SB(I2C_INSTANCE));

        // 2. Adres Urządzenia + zapis
        LL_I2C_TransmitData8(I2C_INSTANCE, dev_addr);
        while (!LL_I2C_IsActiveFlag_ADDR(I2C_INSTANCE));
        LL_I2C_ClearFlag_ADDR(I2C_INSTANCE);

        // 3. Adres Rejestru
        while (!LL_I2C_IsActiveFlag_TXE(I2C_INSTANCE));
        LL_I2C_TransmitData8(I2C_INSTANCE, reg_addr);

        // 4. Wartość
        while (!LL_I2C_IsActiveFlag_TXE(I2C_INSTANCE));
        LL_I2C_TransmitData8(I2C_INSTANCE, value);

        while (!LL_I2C_IsActiveFlag_BTF(I2C_INSTANCE));

        // 6. STOP
        LL_I2C_GenerateStopCondition(I2C_INSTANCE);

        return 0;
    }

    int i2c_read_regs_ll(uint8_t dev_addr, uint8_t reg_addr, uint8_t*
rx_buffer, size_t len) {
        // 1. START
        LL_I2C_GenerateStartCondition(I2C_INSTANCE);
        while (!LL_I2C_IsActiveFlag_SB(I2C_INSTANCE));

        // 2. Adres Urządzenia + zapis
        LL_I2C_TransmitData8(I2C_INSTANCE, dev_addr);
        while (!LL_I2C_IsActiveFlag_ADDR(I2C_INSTANCE));
        LL_I2C_ClearFlag_ADDR(I2C_INSTANCE);

        // 3. Adres Rejestru
        while (!LL_I2C_IsActiveFlag_TXE(I2C_INSTANCE));
        LL_I2C_TransmitData8(I2C_INSTANCE, reg_addr);

        while (!LL_I2C_IsActiveFlag_TXE(I2C_INSTANCE));

        // 4. REPEATED START
        LL_I2C_GenerateStartCondition(I2C_INSTANCE);
```

```
        while (!LL_I2C_IsActiveFlag_SB(I2C_INSTANCE));

        // 5. Adres Urządzenia + odczyt
        LL_I2C_TransmitData8(I2C_INSTANCE, dev_addr | 0x01);
        while (!LL_I2C_IsActiveFlag_ADDR(I2C_INSTANCE));

        if (len == 1) {
            // Jeśli tylko 1 bajt:
            LL_I2C_AcknowledgeNextData(I2C_INSTANCE, LL_I2C_NACK);
            LL_I2C_ClearFlag_ADDR(I2C_INSTANCE);
            LL_I2C_GenerateStopCondition(I2C_INSTANCE); // STOP od razu
po ADDR
        } else {
            // Jeśli więcej niż 1 bajt:
            LL_I2C_AcknowledgeNextData(I2C_INSTANCE, LL_I2C_ACK);
            LL_I2C_ClearFlag_ADDR(I2C_INSTANCE);
        }

        // 6. Pętla odczytu danych
        for (size_t i = 0; i < len; i++) {
            if (i == len - 1) {
                //Jeśli to już koniec danych to trzeba wysłać NACK i
STOP
                if (len > 1) {
                    LL_I2C_AcknowledgeNextData(I2C_INSTANCE,
LL_I2C_NACK);
                    LL_I2C_GenerateStopCondition(I2C_INSTANCE);
                }
            }

            while (!LL_I2C_IsActiveFlag_RXNE(I2C_INSTANCE));

            // Odczyt danych
            rx_buffer[i] = LL_I2C_ReceiveData8(I2C_INSTANCE);
        }

        return 0;
    }
```

Główna pętla programu bez zmian.

Przetestowano kod i działał analogicznie co wersja ISIX:



W wersji LL kodu widać sedno w różnicy między SPI i I2C. I2C ma krótszą inicjalizację niż SPI (bo jest mniej szyn), natomiast posiada bardziej skomplikowany protokół komunikacji, ponieważ jest tylko jedna, współdzielona szyna danych SDA. Ja osobiście bardziej preferuję I2C, ponieważ wymaga mniej pracy przy podłączaniu i konfigurowaniu własnych urządzeń. Natomiast finalny wybór zależy też od prędkości, której wykorzystywane urządzenia wymagają. Dla szybszej transmisji należy oczywiście wybrać SPI.