Laboratorium 4 MARM Magistrale I2C oraz SPI

Lucjan Bryndza *Politechnika Warszawska*

1 grudnia 2019

Celem laboratorium jest zapoznanie się protokołami transmisji szeregowej **I2C** oraz **SPI** w mikrokontrolerach rodziny STM32, oraz sposobami komunikacji przykładowymi układami peryferyjnymi wyposażonymi w powyższe magistrale szeregowe.

1. Zadania do wykonania na ćwiczeniach

Wszystkie ćwiczenia związane laboratorium realizować będziemy z wykorzystaniem płytki ewaluacyjnej **STM32F411E-DISCOVERY**.Po wykonaniu ćwiczenia należy opracować sprawozdanie, którego sposób przygotowania opisano w rozdziale 3.

Do obsługi magistrali *I2C* oraz *SPI* należy wykorzystać sterownik dostępny w bibliotece **libperiph** systemu ISIX, aby uprościć ćwiczenie.

1.1. Obsługa magistrali SPI, żyroskop L3GD20

Do sprzętowej magistrali **SPI1** w zestawie ewaluacyjnym *STM32F411E-DISCO* fabrycznie dołączono sensor MEMS **L3GD20** pełniący funkcję żyroskopu. Napisz program który pracując w cyklu 250ms:

- Odczyta stan odchylenia płytki (sensora L3GD20) w osiach X,Y,Z
- Wypisze stan odchylenia na osiach X,Y,Z na diagnostycznym porcie szeregowym (*dbprintf*)
- Będzie wizualizował stan odchylenia płytki czujnika w osiach X,Y za pomocą diod: **LD3** ... **LD6**
- Jako graniczną wartość odchylenia włączającą daną diodę LED można przyjąć wartość bezwzględną 500 odczytaną z rejestrów **OUT_X** oraz **OUT_Y**

Konfiguracja sygnałów wyboru układu **CS**, opis rejestrów znajduje się w materiałach pomocniczych 2. oraz w literaturze.

Program powinien być napisany tak aby w prosty sposób można było podmienić procedury obsługi magistrali SPI systemu ISIX na inne.

1.2. Obsługa magistrali I2C, akcelerometr LMS303

Do sprzętowego kontrolera **I2C1** w zestawie ewaluacyjnym *STM32F411E-DISCO* fabrycznie dołączono sensor MEMS pełniący funkcję akcelerometru. Napisz program który w cyklu 500ms:

- Odczyta wartość przyśpieszenia w osiach X,Y,Z
- Jeśli wykryto jakąś zmianę na diagnostycznym porcie szeregowym wypisze informację o aktualnej wartości przyśpieszenia.

Program powinien być napisany tak aby w prosty sposób można było podmienić procedury obsługi magistrali I2C systemu ISIX na inne.

1.3. Obsługa magistrali SPI oraz I2C z wykorzystaniem bibliotek LL

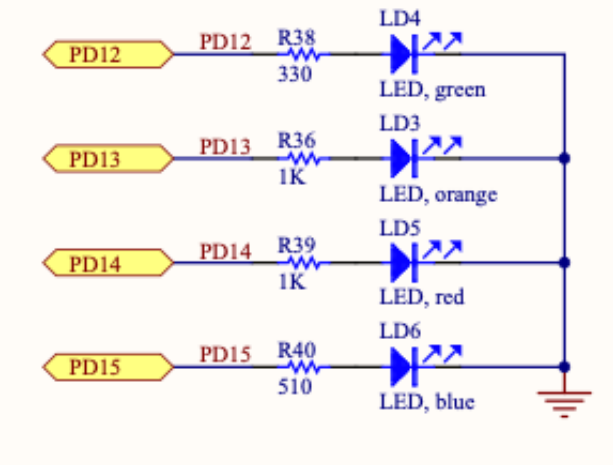
Uwaga! Ten podpunkt jest opcjonalny.

Jeśli po wykonaniu powyższych podpunktów na laboratorium pozostanie jeszcze czas zmodyfikuj powyższe przykłady tak aby zamiast sterowników systemu ISIX do obsługi obu magistral wykorzystać własnoręcznie przygotowane procedury z wykorzystaniem biblioteki *Low Level Driver*.

2. Materiały pomocnicze

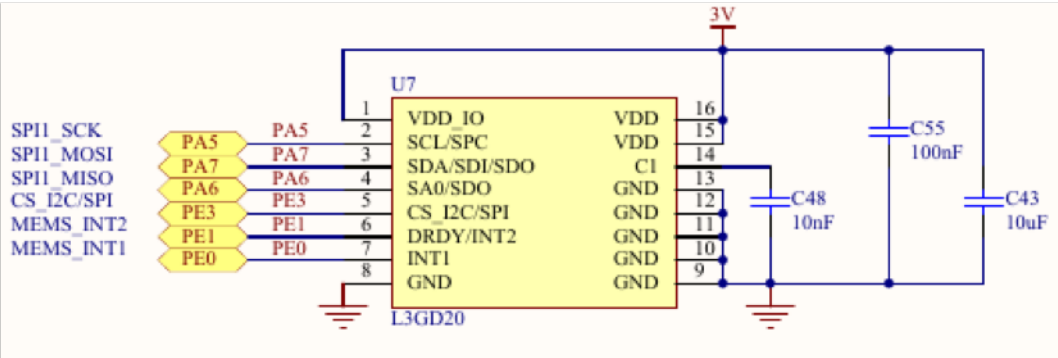
2.1. Diody LED w zestawie STM32F411E-DISCO

Zestaw ewaluacyjny posiada 4 diody LED podłączone do portu **PD** do pinów 12...15.



2.2. Żyroskop L3GD20

Fragmenty schematu zestawu przedstawiające sposób dołączenia układu żyroskopu do mikrokontrolera przedstawiono na poniższym rysunku:



Układ dołączony jest do pinów sprzętowego kontrolera **SPI1** natomiast linia wyboru układu **CS** (*Chip Select*) dołączona została do portu **PE3**.

W trybie **SPI** układ pracuje z następującymi parametrami:

- **8 bitów** danych
- Pierwszy bit jest najbardziej znaczący (**MSB**)
- Częstotliwość maksymalna sygnału zegarowego **10MHz**
- Próbkowanie danych **na pierwszym zboczu** zegara
- Próbkowanie danych przy zmianie stanu **z niskiego na wysoki**

Po włączeniu zasilania należy sprawdzić czy układ jest podłączony do magistrali oraz przeprowadzić jego konfigurację, ponieważ posiada wiele trybów pracy.

Procedurę sprawdzenia czy układ jest dostępny możemy przeprowadzić w następujących krokach:

- Odczytaj zawartość rejestru identyfikacyjnego **ADDR_WHO_AM_I**
- Porównaj czy odczytana wartość to *0xD4*, *0xD3*, *0xD7*
- Jeśli tak oznacza to iż układ jest dołączony do magistrali

Procedura inicjalizacji układu w trybie odpytywania bez wewnętrznej kolejki FIFO i systemu przerwań przedstawia się następująco:

- Wpisz do rejestru **L3GD20_CTRL_REG1** wartość *0x0f*
- Wpisz do rejestru **L3GD20_CTRL_REG2** wartość *0x00*
- Wpisz do rejestru **L3GD20_CTRL_REG3** wartość *0b00001000*
- Wpisz do rejestru **L3GD20_CTRL_REG4** wartość *0b00110000*
- Wpisz do rejestru **L3GD20_CTRL_REG5** wartość *0x00*

Pozycja odchylenia zwracana jest w postaci liczby 16 bitowej ze znakiem, natomiast magistrala SPI wykorzystuje tryb 8 bitowy. Aby odczytać wartość pozycji należy złożyć dwa ośmiobitowe rejestry:

- Odchylenie w osi X: **L3GD20_OUT_X_L**, **L3GD20_OUT_X_H**
- Odchylenie w osi X: **L3GD20_OUT_Y_L**, **L3GD20_OUT_Y_H**
- Odchylenie w osi X: **L3GD20_OUT_Z_L**, **L3GD20_OUT_Z_H**

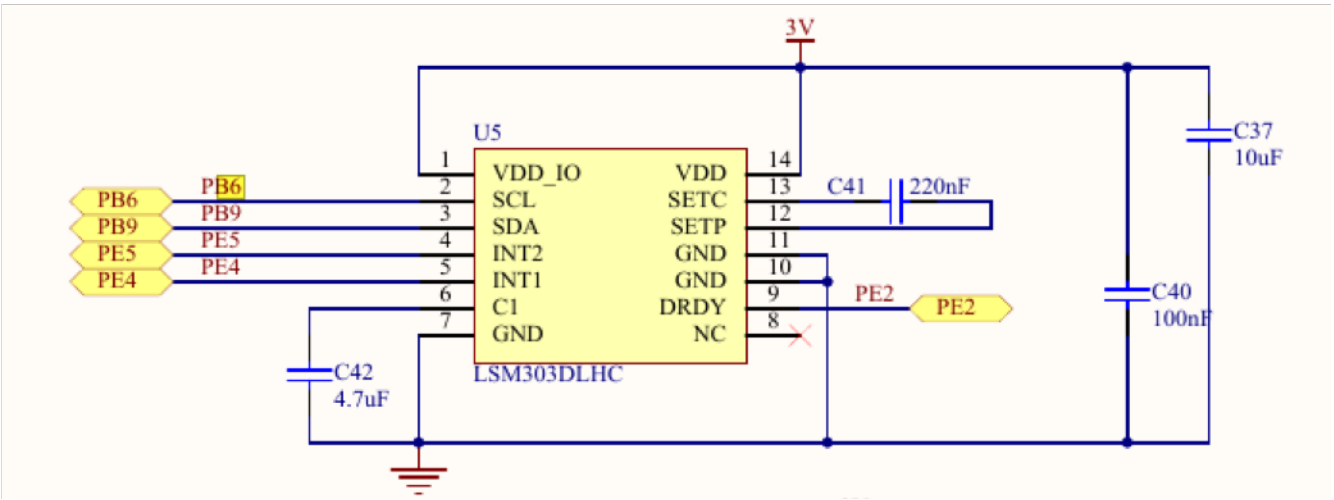
Adresy niektórych rejestrów układu przedstawiono w poniższej tabeli:

Rejestr	Adres (Hex)
ADDR_WHO_AM_I	0x0F
L3GD20_CTRL_REG1	0x20
L3GD20_CTRL_REG2	0x21
L3GD20_CTRL_REG3	0x22
L3GD20_CTRL_REG4	0x23
L3GD20_CTRL_REG5	0x24
L3GD20_OUT_X_L	0x28
L3GD20_OUT_X_H	0x29
L3GD20_OUT_Y_L	0x2A
L3GD20_OUT_Y_H	0x2B
L3GD20_OUT_Z_L	0x2C
L3GD20_OUT_Z_H	0x2D

2.3. Akcelerometr LSM303

Układ LSM303 jest zintegrowanym czujnikiem MEMS zawierającym w jednej obudowie układ kompasu oraz akcelerometru.

Fragmenty schematu zestawu przedstawiające sposób dołączenia układu do mikrokontrolera przedstawiono na poniższym rysunku:



Układ został dołączony do linii portów PB6 (SCL) oraz PB9 (SDA), stanowiącej interfejs wejściowy kontrolera **I2C1**. Linie PE5, oraz PE4 wykorzystywane zostały jako wejścia przerwań zewnętrznych zgłaszanych w wyniku zgłaszania zdarzeń od układu peryferyjnego do mikrokontrolera.

Układ akcelerometru dostępny jest pod adresem I2C: *0x32* natomiast układ kompasu dostępny jest pod adresem I2C: *0x3C*.

Aby skonfigurować akcelerometr i kompas należy ustawić następujące rejestry:

- Do rejestru **LSM303_REGISTER_ACCEL_CTRL_REG1_A** należy wpisać wartość *0x27* (Akcelerometr)
- Do rejestru **LSM303_REGISTER_MAG_MR_REG_M** wpisać wartość *0x00* (Kompas)

Przyśpieszenie zwracane jest w postaci liczby 16 bitowej ze znakiem, natomiast rejestry I2C układu są 8 bitowe. Aby odczytać wartość pozycji należy złożyć dwa ośmiobitowe rejestry:

- Przyśpieszenie dla osi X: **LSM303_REGISTER_ACCEL_OUT_X_L_A**, **LSM303_REGISTER_ACCEL_OUT_X_H_A**
- Przyśpieszenie dla osi Y: **LSM303_REGISTER_ACCEL_OUT_Y_L_A**, **LSM303_REGISTER_ACCEL_OUT_Y_H_A**
- Przyśpieszenie dla osi Z: **LSM303_REGISTER_ACCEL_OUT_Z_L_A**, **LSM303_REGISTER_ACCEL_OUT_Z_H_A**

Sytuacja wygląda analogicznie dla odczytu ziemskiego pola magnetycznego. Aby odczytać wartość pola magnetycznego w poszczególnych osiach należy złożyć dwa 8 bitowe rejestry:

- Pole magnetyczne dla osi X: **LSM303_REGISTER_MAG_OUT_X_L_M**, **LSM303_REGISTER_MAG_OUT_X_H_M**
- Pole magnetyczne dla osi Y: **LSM303_REGISTER_MAG_OUT_Y_L_M**, **LSM303_REGISTER_MAG_OUT_Y_H_M**
- Pole magnetyczne dla osi Z: **LSM303_REGISTER_MAG_OUT_Z_L_M**, **LSM303_REGISTER_MAG_OUT_Z_H_M**

Adresy najważniejszych rejestrów akcelerometru przedstawiono w poniższej tabeli:

Rejestr	Adres (Hex)
LSM303_REGISTER_ACCEL_CTRL_REG1_A	0x20
LSM303_REGISTER_ACCEL_OUT_X_L_A	0x28
LSM303_REGISTER_ACCEL_OUT_X_H_A	0x29
LSM303_REGISTER_ACCEL_OUT_Y_L_A	0x2A
LSM303_REGISTER_ACCEL_OUT_Y_H_A	0x2B
LSM303_REGISTER_ACCEL_OUT_Z_L_A	0x2C
LSM303_REGISTER_ACCEL_OUT_Z_H_A	0x2D

Adresy najważniejszych rejestrów kompasu przedstawiono w poniższej tabeli:

Rejestr	Adres (Hex)
LSM303_REGISTER_MAG_MR_REG_M	0x02
LSM303_REGISTER_MAG_OUT_X_H_M	0x03
LSM303_REGISTER_MAG_OUT_X_L_M	0x04
LSM303_REGISTER_MAG_OUT_Z_H_M	0x05
LSM303_REGISTER_MAG_OUT_Z_L_M	0x06
LSM303_REGISTER_MAG_OUT_Y_H_M	0x07
LSM303_REGISTER_MAG_OUT_Y_L_M	0x08

2.4. Konfiguracja pobranie oraz uruchomienie projektu

Aby pobrać repozytorium możemy się posłużyć komendą, którą należy uruchomić w wierszu polecenia *GIT Bash*:

```
git clone --recursive -b pub/pw/lab4 https://boff.pl/cgit/
  ↪ public/isixsamples/
```

W kolejnym kroku należy zmienić katalog na *isixsamples* oraz pobrać konfigurację dla środowiska *VSCode*:

```
cd isixsamples
curl http://bryndza.boff.pl/downloads/prv/vscodecfg.zip --
  ↪ output vscodecfg.zip
```

Pobrany plik należy rozpakować bezpośrednio do katalogu *isixsamples*.

```
unzip vscodecfg.zip
```

Konfiguracja projektu dla zestawu STM32F411 discovery odbywa się za pomocą polecenia:

```
python waf configure --debug --board=stm32f411e_disco
```

Kompilacje projektu możemy wykonać za pomocą polecenia:

```
python waf
```

Natomiast zaprogramowanie zestawu odbywa się za pomocą polecenia:

```
python waf program
```

Projekt możemy również konfigurować oraz uruchamiać bezpośrednio z *Visual Studio Code*. Najpierw należy w pliku *.vscode/task.json* zmienić argumenty dla polecenia *waf configure*. Następnie za pomocą polecenia **CTRL+P** otwieramy wiersz polecenia i wpisujemy *task waf configure* w kolejny kroku należy zbudować projekt za pomocą polecenia *task waf*, a w kolejnym kroku zaprogramować poleceniem *task waf program*.

Debugowanie programu następuje po wciśnięciu klawisza **F5** wcześniej jednak należy w pliku *.vscode/launch.json* ustawić odpowiednio ścieżkę do uruchamianego programu.

Opis skrótów klawiaturowych dla środowiska **VSCode** możemy znaleźć tutaj: [7].

2.5. Podstawowe API systemu ISIX do ćwiczenia

Podstawową funkcją diagnostyczną służącą do wypisywania danych na domyślny port diagnostyczny (terminal) jest funkcja:

```
#define dbprintf(fmt, ...) fnd::tiny_printf("%s:%d|" fmt "\r\n",
    ↪ _isix_dbglog_extract_basename(__FILE__),__LINE__,##
    ↪ __VA_ARGS__)
```

Funkcja przyjmuje identyczny zestaw argumentów jak standardowa funkcja **printf** pozbawiona jest jedynie formatowania i wyświetlania liczb zmiennoprzecinkowych.

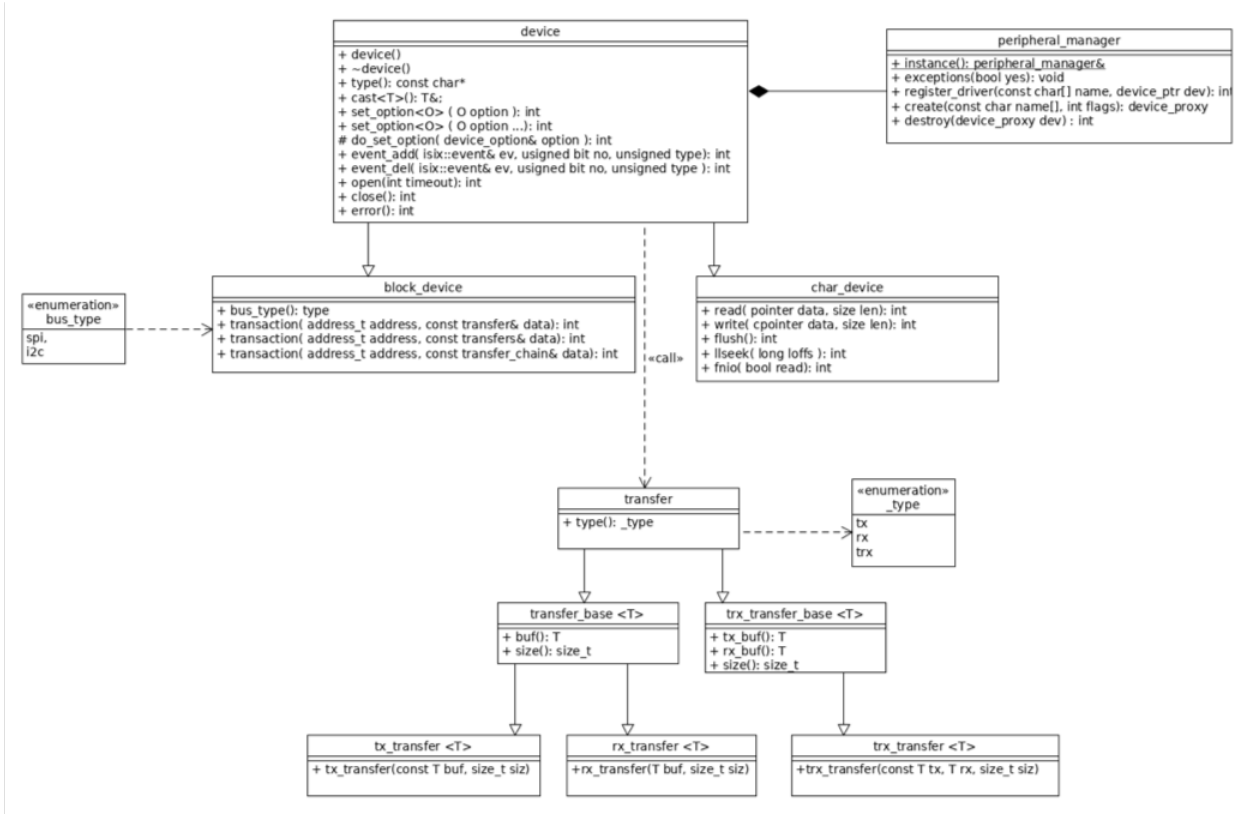
Do utworzenia nowego wątku systemu operacyjnego możemy wykorzystać funkcję:

```
ostask_t isix::task_create(task_func_ptr_t task_func, void *
    ↪ func_param, unsigned long stack_depth, osprio_t priority,
    ↪ unsigned long flags=0);
```

Jako pierwszy argument funkcja przyjmuje funkcję, która będzie wykorzystana do realizacji wątku. Jako argument *func_param* możemy przekazać argument przekazany funkcji realizującej wątek. Parametr *stack_depth* określa wielkość stosu dla danego wątku. Do realizacji zadania powinien wystarczyć stos o wielkości 2kB. Jako parametr *priority* należy przekazać priorytet wątku, przy czym 0 oznacza priorytet najwyższy.

2.6. Interfejs blokowy biblioteki libperiph

Biblioteka **libperiph** została w całości napisana w języku C++17, i od strony interfejsu użytkownika zapewnia jednolite API umożliwiające użytkowanie urządzeń blokowych w postaci niezależnej od sprzętu. Od strony sterowników urządzeń zapewnia również warstwę abstrakcji umożliwiającą konfigurowanie sterowników urządzeń na podstawie danych konfiguracyjnych (*dts.cpp*), oraz warstwę abstrakcji dla układu DMA (*dma.hpp*), czy pamięci cache. Na rysunku przedstawiono hierarchię klas interfejsu blokowego biblioteki **libperiph**:



Klasa **periph_manager** odpowiedzialna jest za rejestrację oraz przechowywanie wszystkich sterowników urządzeń obecnych w systemie. Sterowniki urządzeń możemy rejestrować w systemie za pomocą metody *register_driver()*. Klasą bazową dla wszystkich urządzeń stanowi klasa device, zapewniając interfejs podstawowych metod takich jak otwieranie, zamykanie urządzenia, oraz możliwość ustawiania parametrów sterownika (odpowiednik *ioctl*). Klasa **block_device**, realizuje interfejs dla urządzeń blokowych, gdzie najważniejsza jest metoda transaction umożliwiająca transfer w obu kierunkach zarówno ciągłych jak i nieciągłych bloków danych. Transfer w kierunku hosta, w kierunku urządzenia, oraz dwukierunkowy tworzymy odpowiednio z wykorzystaniem klasy **rx_transfer**, **tx_transfer**, oraz **trx_transfer**, podając bufor oraz jego wielkość w parametrach konstruktora. Transfery również mogą być realizowane z wykorzystaniem nieciągłych bloków pamięci, wówczas należy przekazać je w postaci listy **transfer_chain**. Każda operacja związana z obsługą urządzeń zwraca kod błędu, chyba że ustawiona jest opcja zgłaszania wyjątków za pomocą metody exceptions(bool yes), wtedy w przypadku wystąpienia błędu zostanie zgłoszony wyjątek.

Implementując sterownik urządzenia blokowego (w tym przypadku SPI), należy stworzyć klasę **spi_device**, której klasę bazową stanowić będzie **block_device**, a następnie zaimplementować wszystkie metody czysto wirtualne. Biblioteka **libperiph** zapewnia również niskopoziomowe API ułatwiające pisanie driverów, gdzie jako najważniejsze części możemy wskazać:

- API umożliwiające pobieranie konfiguracji sprzętowej takie jak: konfigura-

cja wybranych portów GPIO związanych z danym układem peryferyjnym, przypisania sygnału zegarowego, czy opcję konfiguracji DMA

- API DMA, umożliwiające wykonywanie operacji bezpośredniego dostępu do pamięci niezależnie od rodzaju kontrolera DMA zintegrowanego z urządzeniem

2.7. Sterownik obsługi magistrali I2C w systemie ISIX

Sterownik magistrali **I2C** został zaimplementowany w postaci klasy **i2c_master** (*stm32_i2c_v1.hpp*) i zawiera implementacje obsługi kontrolera sprzętowego magistrali I2C. Konfiguracja oraz przypisanie pinów sterownika odbywa się za pomocą tablicy konfiguracyjnej **the_machine_config** zawartej w pliku *dts.cpp*. W naszym przykładzie struktura odpowiedzialna za konfigurację magistrali I2C1 przedstawia się następująco:

```
      {
       "i2c1", reinterpret_cast<uintptr_t>(I2C1),
       bus::apb1, LL_GPIO_AF_4,
       unsigned(std::log2(LL_APB1_GRP1_PERIPH_I2C1)),
       i2c1_pins,
              &i2c1_conf
      },
```

Struktura konfiguracyjna rozpoczyna się od podania nazwy urządzenia peryferyjnego w postaci łańcuchowej pod jaką będzie ono widoczne w systemie, oraz fizycznego adresu bazowego układu peryferyjnego. Następnie podajemy identyfikator magistrali do której dołączony jest kontroler, numer funkcji alternatywnej **AF_4** wybierający w multiplekserze kontroler I2C, oraz numer bitu umożliwiający włączenie sygnału zegarowego dla tego układu. Na koniec przekazujemy adres do tablicy zawierającej konfigurację poszczególnych portów GPIO, oraz konfigurację specyficzną dla kontrolera I2C.

Konfigurację poszczególnych linii GPIO przedstawia się następująco:

```
 constexpr pin i2c1_pins[] {
     { pinfunc::scl, gpio::num::PB6 }, //SCK pin
     { pinfunc::sda, gpio::num::PB9 }, //SCL pin
    {}
 };
```

W tablicy znajdują się przypisania oznaczeń portów GPIO do funkcji pełnionych przez linię magistrali I2C. Strukturę odpowiedzialną za konfigurację kontrolera I2C1 przedstawiono poniżej:

```
constexpr device_conf i2c1_conf {
```

```
    {},
    I2C2_EV_IRQn,
    1,7, //! IRQ prio subprio
    0
};
```

Najistotniejszą częścią tej struktury jest numer przerwania zgłaszanego przez kontroler magistrali I2C, oraz priorytet przerwania ustawiany w kontrolerze NVIC.

Aby skorzystać ze sterownika I2C systemu ISIX należy utworzyć obiekt odpowiedniej klasy urządzenia, skonfigurować prędkość pracy, oraz otworzyć port magistrali I2C. Przykład konfiguracji wstępnej magistrali przedstawiono poniżej:

```
      periph::drivers::i2c_master i2c1("i2c1");
      static constexpr auto IFC_TIMEOUT = 1000;
      int ret = i2c1.open(IFC_TIMEOUT);
      if(ret) {
            dbg_err("Unable to open i2c device error: %i", ret);
            report_error();
      }
      ret = i2c1.set_option(periph::option::speed(400'000));
      if(ret) {
            dbg_err("Unable to set i2c option error: %i", ret);
            break;
      }
```

Od tego momentu sterownik magistrali I2C jest gotowy do transferu danych. Aby przesłać dane na magistralę I2C możemy posłużyć się odpowiednim fragmentem kodu:

```
   const unsigned char buf[] = { 0x11, 0x22};
   periph::blk::tx_transfer tran(buf, sizeof buf);
   int status = m_i2c.transaction(i2c_bus_address, tran);
```

Podobnie aby odczytać dane z rejestru jakiegoś układu należy wygenerować transakcję zapisu łączoną z transakcją odczytu (powtórzony start), co możemy zrealizować wykorzystując poniższy fragment kodu:

```
      uint8_t in_dev_addr = 0x00;
      char rx_buffer[1];
   periph::blk::trx_transfer tran( &in_dev_addr,rx_buffer, sizeof
      ↪  in_dev_addr, sizeof rx_buffer);
   int status = m_i2c.transaction(hw_addr, tran);
```

2.8. Sterownik obsługi magistrali SPI w systemie ISIX

Sterownik blokowy zaimplementowany został w klasie **spi_master** i implementuje obsługę interfejsu SPI wbudowanego w mikrokontrolery STM32, dla wszystkich popularnych linii: F1/F2/F4/F7.

Aby móc skorzystać z magistrali SPI w pierwszej kolejności należy utworzyć obiekt klasy SPI master podając w konstruktorze nazwę portu SPI, której konfiguracja znajduje się w pliku *dts.cpp*. W kolejnym kroku za pomocą odpowiednich parametrów option należy skonfigurować port SPI, a następnie otworzyć urządzenie SPI

```
namespace opt = periph::option;
int ret {};
periph::drivers::spi_master spi("spi1");
if((ret=spi.set_option(opt::speed(10E6)))<0) {
        report_error();
}
if((ret=spi.set_option(opt::polarity(opt::polarity::low)))
   ↪ <0) {
        report_error();
}
if((ret=spi.set_option(opt::phase(opt::phase::_1_edge)))<0)
   ↪  {
        report_error();
}
if((ret=spi.set_option(opt::dwidth(8)))<0) {
        report_error();
}
if((ret=spi.set_option(opt::bitorder(opt::bitorder::msb)))
   ↪ <0) {
        report_error();
}
if((ret=spi.open(ISIX_TIME_INFINITE))<0) {
        report_error();
}
```

Ponieważ magistrala SPI jest dwukierunkowa natomiast do komunikacji z układem wykorzystujemy komunikację w jednym kierunku dane przesyłane lub odbieranie są ignorowane. Aby przesłać dane na magistralę I2C należy wywołać metodę **transaction**, której należy przekazać klasę określającą rodzaj transakcji, oraz adres sprzętowy układu na magistrali:

```
/** Make blk specified transfer for single mode
 * @parma[in] addr blk adddress
```

```
     * @param[in] data transfer data in single mode
     * @return error code for blk transfer type
     */
    int transaction(int addr, const blk::transfer& data);
```

W przypadku magistrali SPI jako adres sprzętowy należy przekazać numer porządkowy linii **CS** używanej do sterowania układem, który określono w pliku *dts.cpp*. W naszym przypadku adres porządkowy 0 przypisano do portu **PE3**. Przykład przesłania bufora na magistralę SPI przedstawia poniższy fragment kodu:

```
    const uint8_t tx_buffer[2];
    static constexpr auto csno = 0;
    periph::blk::tx_transfer tran(tx_buffer,sizeof tx_buffer);
    auto status = m_dev.transaction(csno, tran);
```

Natomiast przykład transmisji dwukierunkowej reprezentuje poniższy fragment kodu:

```
    const uint8_t inbuf[2] {};
    uint8_t outbuf[2] {};
    static_assert(sizeof outbuf==sizeof inbuf);
    static constexpr auto csno = 0;
    periph::blk::trx_transfer tran(inbuf,outbuf,sizeof outbuf);
    int ret = m_dev.transaction(csno, tran);
```

3. Instrukcja opracowania sprawozdania

- Umieścić oscylogram transferu danych na magistrali I2C podczas odczytu pozycji z akcelerometru
- Umieścić oscylogram transferu danych na magistrali SPI podczas odczytu odchylenia z żyroskopu
- W sprawozdaniu umieścić kod źródłowy wszystkich programów
- Opisać zasadę działania programów
- Napisać czy preferujesz układy peryferyjne z magistralą SPI czy I2C? Którą magistralę i dlaczego wybrał byś gdybyś miał skonstruować własne urządzenie, a musiał się zdecydować na jedną z nich?

Literatura

[1] *STM32F411E-DISCOVERY* kit user manual

[2] *STM32F411* reference manual

[3] *L3GD20* Datasheet

[4] *L3GD20* Application note

[5] LMS303 Datasheet

[6] *ISIX-RTOS* API examples

[7] *Visual Studio Code* keyboards shortcuts