Systemy mikroprocesorowe

Instrukcja laboratoryjna

Spotkanie 3

Interfejsy komunikacyjne

Autorzy: dr inż. Paweł Dąbal,
dr inż. Krzysztof Sieczkowski
mgr inż. Sylwia Zawadzka

Ostatnia aktualizacja: 24.11.2022 r.

Wersja: 2.0.0

Warszawa 2022

Spis treści

1. Wprowadzenie

Instrukcja ta zawiera zadania związane z tworzeniem programów obsługujących interfejsy komunikacyjne dostępne w mikrokontrolerze. W trakcie zajęć student będzie korzystał z środowiska programistycznego *STM32CubeIDE*, rozbuduje dostarczony projekt aplikacji dla mikrokontrolera STM32L496ZGT6 umieszczonego na płycie uruchomieniowej *KAmeLeon* o funkcjonalność umożliwiającą obsłużenie interfejsów komunikacyjnych. Ponadto będzie potrafił obsługiwać narzędzia wspomagające kontrolę wersji oprogramowania w celu dokumentowania własnych postępów.

1.1. Cel ćwiczenia

Ćwiczenie laboratoryjne ma na celu:

- nabycie umiejętności konfiguracji stanowiska pracy w oparciu o oprogramowanie *STM32CubeIDE 1.7.0*;
- poznanie płyty uruchomieniowej *KAmeLeon* z mikrokontrolerem *STM32L496ZGT6*;
- przypomnienie i usystematyzowanie wiedzy i umiejętności z zakresu posługiwania się językiem C;
- przypomnienie i usystematyzowanie wiedzy i umiejętności z zakresu obsługi interfejsów UART i I2C;
- praktyczne korzystanie z systemu kontroli wersji *Git* i serwisu *GitHub*.

1.2. Wymagania wstępne

Przed przystąpieniem do wykonywania ćwiczenia laboratoryjnego należy :

- zapoznać się z schematem płyty uruchomieniowej *KAmeLeon* oraz dokumentacją mikrokontrolera *STM32L496ZGT6* (*Product Specifications*, *Reference Manuals*);
- zapoznać się z składnią języka C – pojęcie zmiennej, stałej, funkcji, prototypu funkcji, parametru funkcji, wyrażenia warunkowego, pętli, wskaźnik, struktury i tablicy;
- zapoznanie się z sposobem działania interfejsu UART oraz I2C;
- utworzyć konto na platformie *GitHub*;
- zapoznać się z dokumentacją środowiska *STM32CubeIDE* i biblioteki *STM32CubeL4*.

1.3. Opis stanowiska laboratoryjnego

Stanowisko do przeprowadzenia zajęć składa się z komputera PC z zainstalowanym oprogramowaniem koniecznym do realizacji zajęć: *STM32CubeIDE*, dedykowaną do układu biblioteką *STM32CubeL4* i systemem *Git* oraz płyty *KAmeleon* podłączonej do komputera za pomocą przewodu USB do złącza programatora *SWD PRG/DBG/vCOM* płyty uruchomieniowej. Połączenie to również odpowiada za zasilanie płyty.

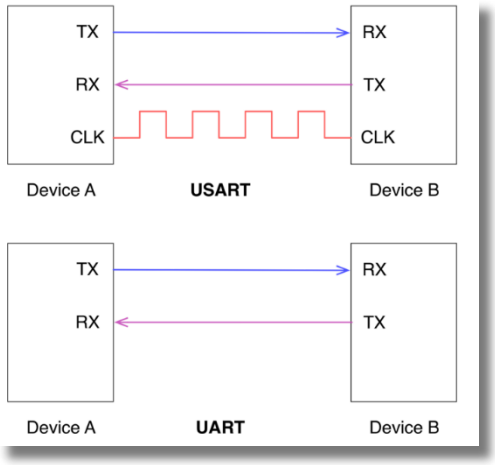
1.4. Wprowadzenie teoretyczne

Współczesne mikrokontrolery wyposażone są w szereg różnych interfejsów. Podzielić możemy je na dwie zasadnicze grupy: lokalne łączące układy scalone w obrębie urządzenia (UART, SPI, I2C, SAI) oraz rozległe łączące różne systemy cyfrowe (UART, CAN, USB).

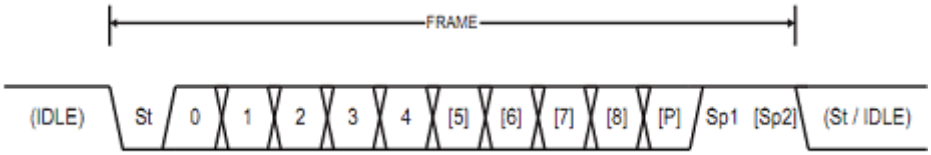
1.4.1. Interfejs szeregowy UART

Interfejs szeregowy UART stanowi podstawowy interfejs komunikacyjny występujący we współczesnych mikrokontrolerach. Standard ten wywodzi się ze standardu RS232, który został opracowany w 1969 r. przez

Electronic Industries Association. UART pozwala na dwukierunkową, asynchroniczną transmisję danych pomiędzy dwoma urządzeniami pracującymi w tym standardzie. Asynchroniczna transmisja w odróżnieniu od transmisji synchronicznej nie posiada dodatkowego sygnału synchronizującego (***CLK***). Zatem do dwukierunkowej transmisji danych w standardzie UART wymagane są tylko dwie linie transmisyjne, tj. ***RX*** oraz ***TX***. Linia ***RX*** stanowi wejście danych napływających z zewnętrznego urządzenia, natomiast linia ***TX*** stanowi wyjście wyprowadzanych danych wyjściowych. Urządzenia komunikujące się w standardzie UART często określane są mianem *Host* oraz *Device*, gdzie *Host* w przypadku transmisji synchronicznej traktowany jest jako urządzenie nadrzędne. W transmisji synchronicznej Host zapewnia sygnał zegarowy. Ponieważ w transmisji UART nie występuje sygnał zegarowy to urządzenia te nie mają określonych priorytetów działania i mogą niezależnie od siebie, w dowolnym czasie transmitować dane. Podstawowa ramka transmisyjna, stosowana w komunikacji UART przedstawiona została na poniższym rysunku.



Brak transmisji UART na linii określane jest przez występowanie na niej stanu wysokiego. Urządzenie chcące wysłać ramkę pierwotnie wysyła bit startu (ustawia linię w stan niski), następnie transmitowane są dane (od 5 do 9 bitów), w zależności od konfiguracji interfejsu, wysyłany jest bit parzystości. Zakończenie transmisji realizowane jest poprzez wysłanie bitu stopu (ustawienie linii w stan wysoki). Wszystkie bity danych przesyłane są od najmniej znaczącego (*LSB*) do najbardziej znaczącego (*MSB*) bitu danych. Format ramki, gdzie występuje bit startu, 8 bitów danych, brak bitu parzystości oraz bit stopu o rozmiarze 1 bita danych, określany jest w skrócie: *8N1* (8 bitów danych, *N* – brak bitu parzystości, bit stopu o rozmiarze 1 bitu). Format ramki *8N1* jest najczęściej stosowanym formatem danych, który występuje w transmisji UART. Rozmiar każdego bitu danych jest ściśle związany z szybkością transmisji (ang. *baud rate*). Szybkość transmisji pozwala na określenie czasu trwania poszczególnego bitu jest to tzw. *Bod,* które mogą być przetransmitowane w czasie 1 sekundy. Powszechnie stosowanymi szybkościami transmisji UART są: 9600, 19200, 38400, 57600, 115200. W wielu systemach cyfrowych możliwa jest niestandardowa konfiguracja prędkości do np. 1 Mb, 2 Mb. Ważne, aby szybkość transmisji wyrażaną w *Bod*-ach, nie mylić z szybkością transmisji wyrażaną w bajtach, gdyż np. w konfiguracji 8N1 do przesłania 1 B danych wymagane jest przesłanie 10 *Bod*. Obecnie transmisja danych przy pomocy standardu UART najczęściej stosowana jest do komunikacji systemu z użytkownikiem. Zatem, transmitowanymi danymi są najczęściej litery zakodowane w kodzie ASCII. Nie ma natomiast przeciwwskazań do stosowania interfejsu UART do transmisji danych binarnych. Na poniższym rysunku przedstawiono zrzut ekranu analizatora stanów logicznych, gdzie występuje transmisja danych UART od systemu mikroprocesorowego do terminala zainstalowanego na komputerze PC.



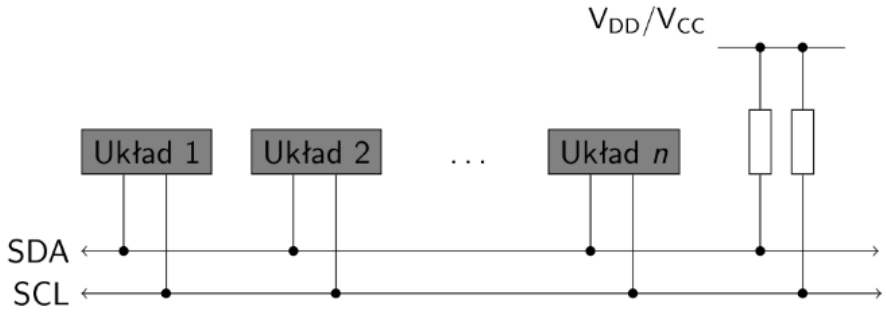
Przedstawiony wykres obrazuje transmisję słowa „*Witaj*!”, które wysyłane zostało z szybkością transmisji: 9600. Natomiast ramka danych transmitowana jest w formacie 8N1. Na przedstawionym wykresie za ramkami transmitującymi znak po znaku słowo „*Witaj*!”, występują również ramki z „\r” oraz „\n”. Znaki „\r” i „\n” są to

tzw. *escape character*, czyli dodatkowe znaki sterujące kursorem podczas transmisji danych w formie tekstowej. Do transmisji danych tekstowych w kodzie ASCII nie ma obowiązku stosowania znaków *escape charakter*, jednak stosowanie ich pozwala przedstawić transmitowany tekst w odpowiedniej formie. Występujące znaki „\r" oraz „\n" pozwalają przenieść kursor do nowej linii (tak jak odbywa się to po naciśnięciu klawisza *Enter*).

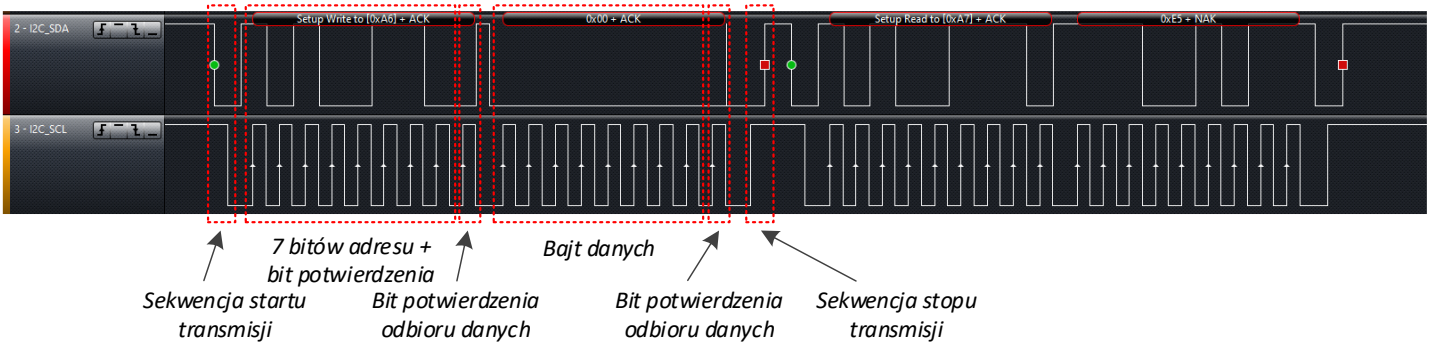
Interfejs szeregowy (USART – ang. *Universal Synchronous/Asynchronous Receiver Transmitter*) jest dostępny w mikrokontrolerze STM32L496ZGT6 w sześciu modułach o zróżnicowanych funkcjonalnościach. Najważniejsze z nich to wsparcie dla sprzętowego wsparcia kontroli przepływu danych, obsługa transmisji z bezpośrednim dostępem do pamięci (DMA), komunikacją między procesorową. Jeden z modułów LPUART dodatkowo może pracować w niższych poziomach zasilania. Niewątpliwą zaletą tego interfejsu jest możliwość transmisji w trybie dwukierunkowym (ang. *full duplex*), czyli sposobność do przesyłania informacji w obu kierunkach jednocześnie.

1.4.2. Interfejs I2C

Interfejs I2C (ang. *Inter-Integrated Circuit*) stanowi kolejny powszechnie stosowany szeregowy interfejs komunikacyjny. Pierwsza wersja interfejsu została opracowana w 1982 przez firmę Philips. W komunikacji I^2C rozróżnia się dwa typy urządzeń: *master* (ten który najczęściej rozpoczyna transmisję) oraz *slave* (urządzenie najczęściej odpowiadające na polecenie *master*). W sieci I^2C możliwa jest konfiguracja (wielu) urządzeń *master* oraz również (wielu) urządzeń *slave*. Najczęściej jednak występuje jedno urządzenie master (mikrokontroler) oraz wiele urządzeń *slave* (zewnętrzne urządzenia peryferyjne, takie jak np. czujniki, pamięci, zegar RTC). Do komunikacji w standardzie I^2C wymagane są dwie linie komunikacyjne: *SCL* oraz *SDA*. Linia *SCL* stanowi linie zapewniającą sygnał zegarowy. Z kolei przy pomocy linii *SDA* transmitowane są dane. Transmisja może odbywać się w obydwie strony, ale nie jednocześnie, dlatego każda ramka danych w pełnej transmisji I^2C ma swoje szczególne znaczenie. Od strony sprzętowej, obydwie linie wymagają tzw. podciągania do linii zasilającej przy pomocy dwóch rezystorów o wartości ściśle zależnej do przyjętej szybkości transmisji. Wartość ta najczęściej wynosi 4,7 kΩ. Standard I^2C w pierwotnej wersji oferuje szybkość transmisji danych (przełączania sygnału zegarowego) na poziomie 100kHz. Występują również kolejne wersje standardu I^2C, które m.in. oferują zwiększoną szybkość transmisji sięgającą nawet pojedynczych MHz. W niniejszym laboratorium zastosowana zostanie podstawowa szybkość transmisji (100kHz). Istotnym wymogiem transmisji I^2C jest stałość poziomu logicznego na linii *SDA* podczas trwania stanu wysokiego na linii *SCL*. Zatem aktualny stan występujący na linii *SDA* musi być stabilny najlepiej już nieco przed zboczem narastającym sygnału *SCL* i utrzymywać się nieco po wystąpieniu zbocza opadającego sygnału zegarowego.



Rozróżnianie wielu urządzeń *slave* występujących na jednej magistrali możliwe jest przy pomocy indywidualnych adresów urządzeń. Każde urządzenie *slave* posiada własny 7 lub 10-bitowy adres. Stan niezajętej magistrali widoczny jest poprzez dłuższe utrzymywanie się stanu wysokiego na liniach *SCL* i *SDA*. Rozpoczęcie transmisji realizowane jest poprzez zmianę stanu linii *SDA* z wysokiego na niski, podczas trwania wysokiego stanu na linii *SCL*. Zatrzymanie transmisji odbywa się poprzez zmianę stanu na linii *SDA* ze stanu niskiego na wysoki podczas występowania wysokiego stanu na linii *SCL*. Prawidłowa transmisja danych występuje po wysłaniu znaku startu. Stan magistrali *SDA* nie zmienia się w momencie występowania stanu wysokiego na *SCL*. Zmiana stanu na magistrali *SDA* może nastąpić podczas stanu niskiego na linii *SCL*. Na poniższym rysunku przedstawiono przykładowy przebieg transmisji I^2C uzyskany z analizatora stanów logicznych.

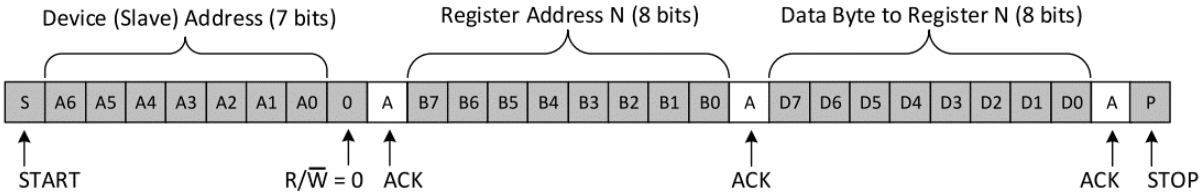


Każdą transmisje rozpoczyna znak startu i kończy znak stopu. Informacje wysyłane linią *SDA* mają rozmiar 8 bitów (1 bajt). Dane wysyłane są od najbardziej znaczącego (*MSB*) do najmniej znaczącego (*LSB*) bitu. Każdy wysyłany bajt musi być potwierdzony przez urządzenie odbiorcze przy pomocy bitu potwierdzenia. Bit potwierdzenia występuje zawsze po 8 bitach danych. Jeżeli bit potwierdzenia jest w stanie wysokim to urządzenie odbiorcze nie potwierdziło faktu odebrania danych lub urządzenie odbiorcze nie będzie dalej odbierało danych. Jeżeli natomiast bit ten jest w stanie niskim to urządzenie odbiorcze poprawnie odebrało dane i domyślnie jest gotowe na odbiór kolejnych ramek I^2C.

W kompletnej transmisji I^2C wyróżniamy dwa typy ramek: ramkę zawierającą adres urządzenia wraz ze wskazaniem zamiaru transmisji oraz ramkę danych która zawiera 1 bajt informacji. Każdy typ ramki zawiera bit potwierdzenia transmisji. Transmisja danych za pośrednictwem interfejsu I^2C ukierunkowana jest albo na odczyt danych z urządzenia *slave* przez *master* albo zapis danych w urządzeniu *slave* przez *master*. W zależności od zamiaru transmisji całkowity przebieg transmisji zawierający ramki danych nieznacznie różni się pomiędzy sobą. Odczyt danych przez *master* realizowany jest najczęściej w poniższy sposób. Po sekwencji startu wysyłany jest 7-bitowy adres urządzenia *slave*. Za tym adresem w polu R/W ustawiany jest bit informujący o bieżącym zamiarze. Ponieważ bit ten jest 0 to urządzenie s*lave* „wie" że będzie u niego występował zapis danych. Zapis danych w tym przypadku polega na ustawieniu aktywnego rejestru poprzez wskazanie go przy pomocy adresu. Ustawienie aktywnego rejestru powoduje, że najbliższy odczyt lub zapis będzie dotyczył właśnie tego rejestru. Po prawidłowym odebraniu adresu urządzenia i bitu R/W urządzenie *slave* zeruje bit *A* informując tym samym, że dane zostały odebrane. Następnie urządzenie Master wysyła drugi bajt danych, który teraz zawiera adres rejestru, do którego będzie chciał mieć dostęp. Po poprawnym odbiorze danych przez *slave* bit *A* jest również zerowany. W tym momencie (połowa transmisji) następuje ponowne wysłanie sekwencji startu. Trzeci bajt danych również zawiera ten sam 7-bitowy adres urządzenia *slave*, ale bit R/W jest ustawiony. Ustawiony bit R/W informuje, że urządzenie Master spodziewa się odbioru danych od *slave*. Po trzeciej poprawnie odebranej

ramce danych *slave* potwierdza (bit *A*), że odebrał dane. W czwartej ramce urządzenie *slave* wysyła zawartość rejestru o adresie wskazanym w ramce drugiej. Jeżeli *master*, po odebraniu danych (1 bajtu, który stanowi zawartość rejestru) zamierza zakończyć transmisje to przed sekwencją stopu ustawia bit NA w stan wysoki co świadczy, że nie jest gotowy na odbiór kolejnych danych. W przypadku zapisu danych do rejestru, sytuacja jest nieco prostsza.

Tak samo jak poprzednio, po sekwencji startu, wysyłany jest 7-bitowy adres urządzenia *slave* oraz bit sygnalizujący zamiar transmisji. Bit R/W również jest zerowany co świadczy, że *slave* otrzyma dane, które będzie musiał zapisać. Po odebraniu pierwszej ramki *slave* potwierdza poprawność odczytu poprzez wyzerowanie bitu *A*. W drugiej ramce wysyłany jest 8-bitowy adres rejestru wewnętrznego *slave*. Po odebraniu slave również potwierdza poprawność (poprzez wyzerowanie bitu *A*). W trzeciej ramce wysyłana jest wartość, którą należy



zapisać rejestr o poprzednio wskazanym adresie. Po odebraniu trzeciej ramki *slave* potwierdza odbiór danych. W większości urządzeń obsługujących transmisję I2C po wysłaniu kolejnych ramek, nastąpi zapisanie kolejnych wewnętrznych rejestrów urządzenia *slave*. Jednak dokładne zachowanie urządzenia peryferyjnego jest sprecyzowane w jego nocie katalogowej.

I2C jest dostępny w mikrokontrolerze STM32L496ZGT6 w czterech modułach. Wspiera szereg trybów pracy w tym *SMBus* i *PMBus*. Przed pierwszym użyciem interfejsu należy go skonfigurować tzn. włączyć zegar dla modułu, wyprowadzenia mikrokontrolera. Obsługa *I2C* z użyciem biblioteki HAL odbywa się z użyciem funkcji grupy *HAL_I2C*. Można wyróżnić kilka trybów pracy: odpytywania z blokowaniem (ang. *polling in blocking*), odpytywania w trybie pamięci (ang. *polling in blocking memory*), przerwania (ang. *interrupt*), transmisji z obsługą bezpośredniego dostępu do pamięci.

1.5. Sposób realizacji ćwiczenia laboratoryjnego

Każde zajęcia składać będą się z następujących elementów:

- zadań obowiązkowych - do wykonania krok po kroku na podstawie instrukcji – do uzyskania od 0 do 12 pkt.;
- zadań uzupełniających – do samodzielnego zbudowania i napisania programu – do uzyskania od 0 do 14 pkt;
- pytań sprawdzających rozumienie działania programu – zadawane przez prowadzącego przyjmującego wykonanie zadania – odpowiedź wpływa na punktację z zadań obowiązkowych i uzupełniających, tzn. czy przyznać maksymalną możliwą liczbę punktów za zadanie czy tylko część przy ewidentnym braku zrozumienia problemu.

Po zrealizowaniu każdego z zadań należy poprosić prowadzącego o sprawdzenie i przyznanie punktów. Na koniec zajęć wystawiana jest ocena na podstawie sumy uzyskanych punktów: **2,0** <0; 10), **3,0** <10; 13), **3,5** <13; 16), **4,0** <16; 19), **4,5** <19; 23), **5,0** <23; 26>. W każdych zajęciach należy uczestniczyć. Przy każdym zadaniu określona jest liczba punktów jakie można uzyskać. W przypadku zadań uzupełniających premiowana jest **jakość** i **czas realizacji** zaprezentowanego rozwiązania. Ocena końcowa z laboratorium to średnia arytmetyczna ocen z każdego spotkania.

W trakcie wykonywania ćwiczenia należy zwrócić szczególną uwagę na wyróżnione w treści instrukcji fragmenty tekstu wyróżnione kolorem:

- ***zielonym*** – etykiety wartości, nazwy pola, nazwy funkcji;
- ***niebieskim*** – wartości jakie należy użyć we skazanym miejscu;
- ***purpurowy*** – odwołania do kodu programu, nazw własnych.

2. Zadania podstawowe do realizacji (12 pkt.)

W tej części instrukcji zamieszczone są treści, z którymi obowiązkowo należy się zapoznać i praktycznie przećwiczyć. Ważne jest, aby zapamiętać wykonywane przedstawione czynności, aby móc na kolejnych zajęciach wykonywać je na kolejnych zajęciach bez potrzeby sięgania do niniejszej instrukcji.

Uwaga: Załączone wycinki z ekranu są poglądowe i pomagają jedynie w wskazaniu lokalizacji elementów interfejsu. Należy używać wartości podanych w tekście.

2.1. Zagadnienia wstępne

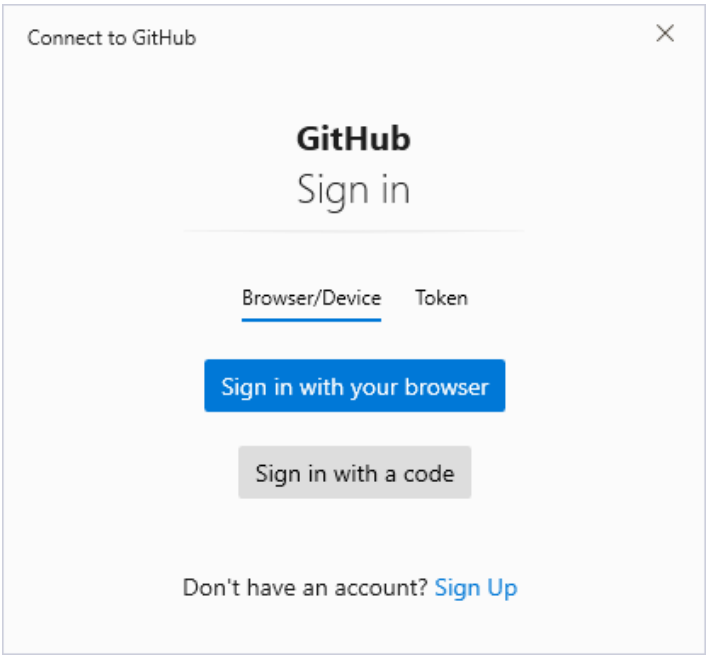
Przed przystąpieniem do pracy należy skonfigurować zainstalowane oprogramowanie i uzyskać dostęp do repozytorium, tzn. system kontroli wersji *Git*, instalacji biblioteki do obsługi mikrokontrolerów rodziny STM32L4 w środowisku *STM32CubeIDE* oraz sklonować repozytorium dla zajęć.

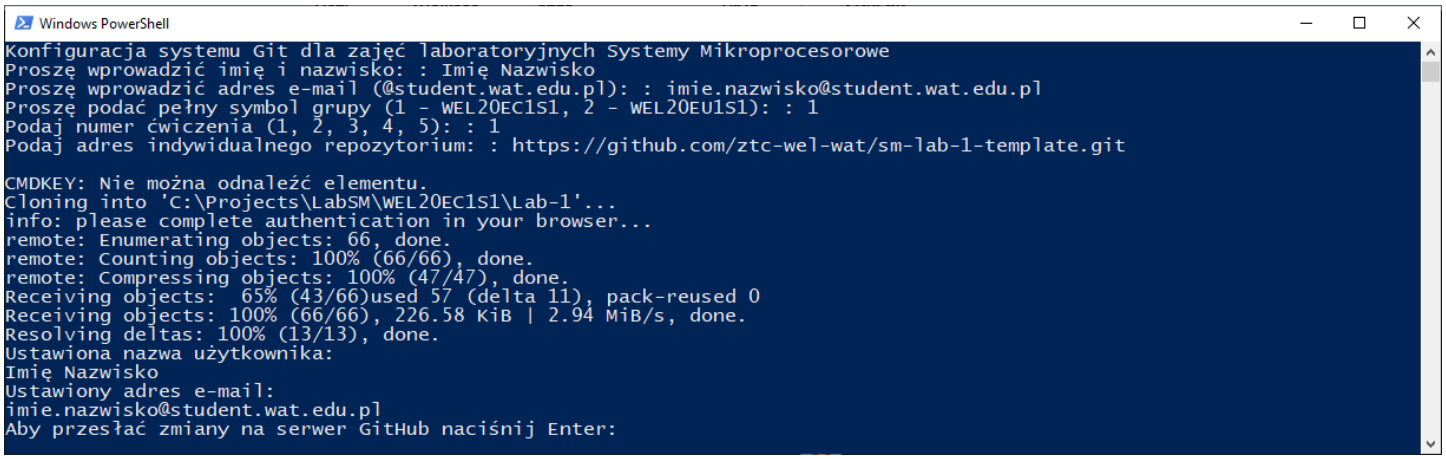
2.1.1. Dołączenie do wirtualnej klasy i utworzenie repozytorium bazowego dla zadania

Na zajęciach należy dołączyć do wirtualnej grupy w ramach *Classrom GitHub* *za pomocą udostępnionego odnośnika prowadzącego do zadania*. Odnośnik prowadzi do kreatora w którym tworzone jest zadanie – indywidualne repozytorium studenta. Z listy należy wybrać swój adres e-mail i potwierdzić przyjęcie zadania (ang. *assignment*). Automatycznie zostanie utworzone prywatne repozytorium indywidualnie dla każdego studenta na podstawie przygotowanego repozytorium-szablonu. Na pierwszych zajęciach jest to wersja minimalna, a na kolejnych będzie zawierała już wstępnie skonfigurowane projekty. W sytuacji jeżeli student nie może odszukać się na liście (np. ktoś inny podłączył się pod daną osobę) proszę zgłosić to prowadzącemu zajęcia. W celu skorygowania nieprawidłowości. Zaakceptowanie zadania wiąże się również z dołączeniem do organizacji – wirtualnego konta organizacji w ramach którego tworzone są indywidualne repozytoria do zadań, z którego prowadzący zajęcia mają dostęp do wszystkich repozytoriów tworzonych w ramach zajęć.

2.1.2. Pobranie repozytorium i konfiguracja lokalna

W celu pobrania repozytorium należy odszukać na pulpicie skrót o nazwie ***LabGitConfig***, który uruchomi skrypt wiersza poleceń, w którym należy podać: 1) *swoje imię i nazwisko*, 2) *adres e-mail*, 3) *symbol grupy*, 4) *numer ćwiczenia*, 5) *adres indywidualnego repozytorium* uzyskany w wcześniejszym punkcie. Po wykonaniu punktu 5) pojawi się okno logowania do serwisu *GitHub* podobne do zamieszczonego obok. W celu połączenia należy wybrać przycisk ***Sign in with your browser*** co spowoduje otworzenie nowej karty przeglądarki w którym należy udzielić dostępu do konta. Po uzyskaniu zgody nastąpi sklonowanie projektu z serwera do lokalizacji wynikającej z symbolu grupy. Na ten moment należy zminimalizować okno wiersza poleceń. Na zakończenie zajęć pozwoli ono na wypchnięcie zmian (ang. *push*) do repozytorium zdalnego. Przykładowy przebieg wykonania skryptu zamieszczony został poniżej.

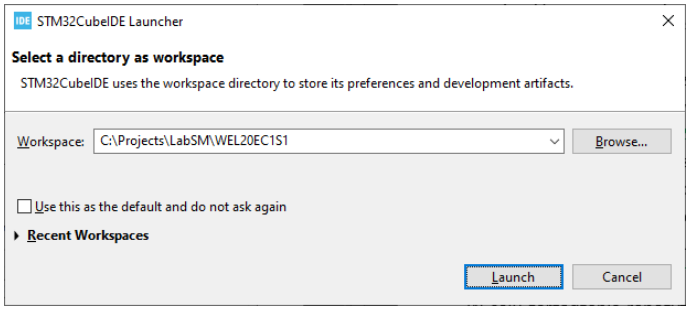


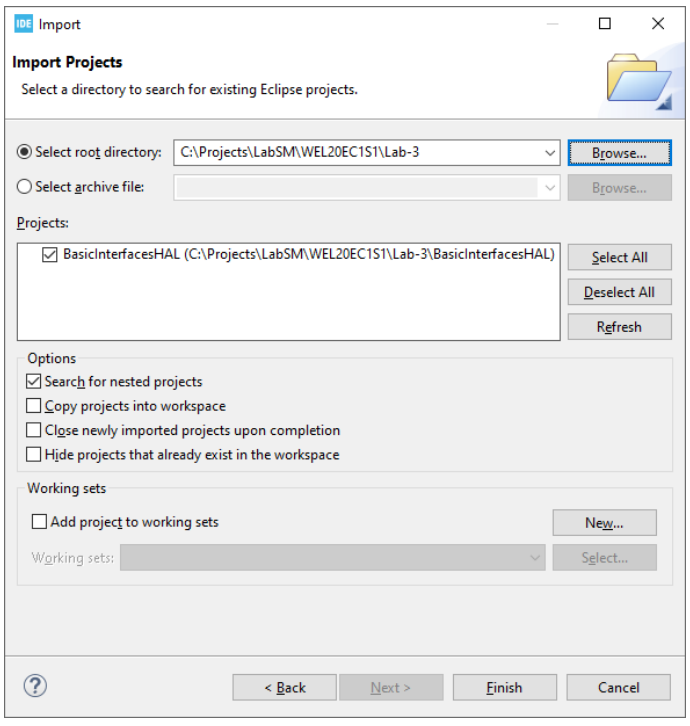


2.1.3. Konfiguracja środowiska STM32CubeIDE

Korzystając ze skrótu na pulpicie lub z menu *Start* należ uruchomić środowisko projektowe *STM32CubeIDE*. Po jego uruchomieniu pojawi się pytanie o wskazanie katalogu, który będzie pełnił rolę przestrzeni roboczej. Należy wskazać na katalog *swojej grupy studenckiej* gdzie umieszczone zostało sklonowane repozytorium (w przykładzie: *C:\Projects\LabSM\WEL20EC1S1*). Wybór katalogu zatwierdzamy przyciskiem ***Launch***. Można używać wielu różnych przestrzeni roboczych w przypadku pracy z różnymi projektami. Po załadowaniu środowiska należy zamknąć zakładkę *Information Center*.

W celu zarządzania repozytorium dla zadania należy w środowisku *STM32CubeIDE* należy otworzyć widok (ang. *perspective*) zarządzania repozytorium. W tym celu z menu wybieramy: ***Window → Perspective → Open Perspective → Other…***, gdzie z listy należy wybrać ***Git***. Środowisko przełączy się do widoku *Git* i po lewej stronie pojawi się zakładka *Git Repositories*. Następnie korzystając z odnośnika ***Add an existing local Git repository*** (domyślnie po lewej stronie programu) otworzy się okno dodawania repozytorium (*Add Git Repositories*). Za pomocą przycisku ***Browse…*** należy wskazać położenie sklonowanego repozytorium (np. *C:\Projects\LabSM\WEL20EC1S1\Lab-3*), a następnie zaznaczyć pozycję dla ćwiczenia pierwszego.





W kolejnym kroku należy zaimportować projekt do przestrzeni roboczej poprzez wybranie z menu ***File → Import***, a następnie w oknie, które się pojawi wybrać ***General → Existing Projects into Workspace***. Zatwierdzić przyciskiem ***Next***. Za pomocą przycisku ***Browse…*** wybrać należy katalog repozytorium (np. *C:\Projects\LabSM\WEL20EC1S1\Lab-3*). Zaktualizuje się lista dostępnych projektów i na niej należy wybrać ***BasicInterfacesHAL*** i zatwierdzić przyciskiem ***Finish***.

W trakcie zajęć ograniczymy się do prostej liniowej struktury migawek wykonywanych po zakończeniu każdego z zadań instrukcji opatrzonych stosownym komentarzem. W dalszej części instrukcji będą podane treści komentarzy jakimi należy opatrzyć realizowane migawki. Powstanie zatem coś w rodzaju sprawozdania z zajęć, które będzie *przechowywane* w serwisie *GitHub*.

2.1.4. Opis projektu bazowego

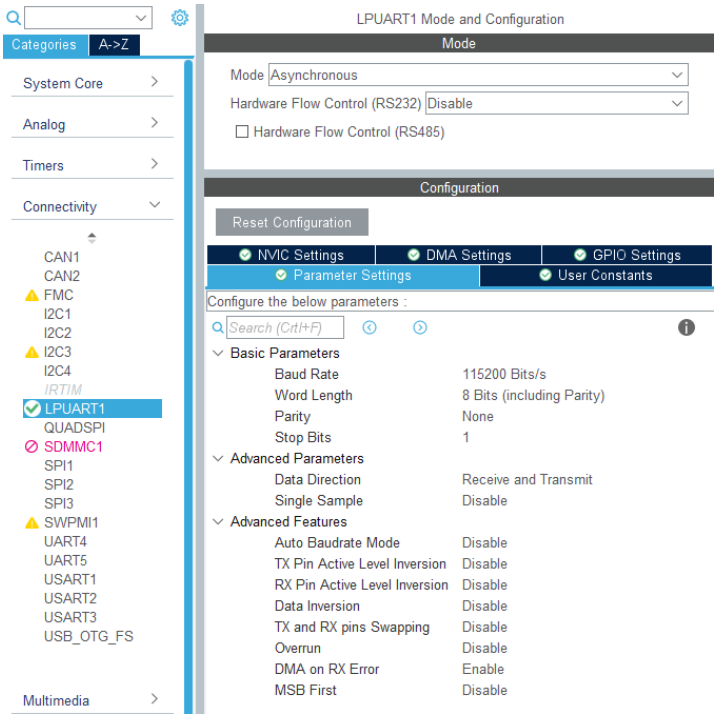
W sklonowanym repozytorium znajduje się projekt bazowy korzystający z biblioteki HAL o nazwie ***BasicInterfaceHAL***, który w poprzednim kroku został zaimportowany do środowiska *STM32CubeIDE*. Posiada on skonfigurowane wyprowadzenia mikrokontrolera do których dołączone są diody ***LED*** (*LED0* … *LED7*) oraz pięć styków joysticka joystick (*SW_RIGHT*, *SW_ LEFT*, *SW_ DOWN*, *SW_UP*, *SW_OK*). W pliku ***gpio.c*** zdefiniowane zostały podstawowe funkcje do obsługi diod LED oraz odczytu stanu joysticka. Ponadto skonfigurowana i dodana została obsługa wyświetlacza 7-segmentowego oraz tekstowego LCD w plikach ***display.c*** i ***lcd.c***.

2.2. Obsługa interfejsu szeregowego (6 pkt.)

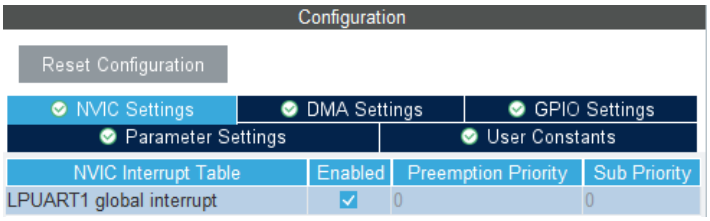
Interfejs szeregowy (USART – ang. *Universal Synchronous/Asynchronous Receiver Transmitter*) jest dostępny w mikrokontrolerze STM32L496ZGT6 w sześciu modułach o zróżnicowanych funkcjonalnościach. Najważniejsze z nich to wsparcie dla sprzętowego wsparcia kontroli przepływu danych, obsługa transmisji z bezpośrednim dostępem do pamięci (DMA), komunikacją między procesorową. Jeden z modułów ***LPUART*** (ang. *Low Power UART*) dodatkowo może pracować w niższych poziomach zasilania. Na płycie *KAmeLeon* komunikacja między mikrokontrolerem a komputerem z użyciem interfejsu szeregowego ***LPUART1*** podłączonego do układu programatora pełniącego role konwertera UART – USB. W tym celu należy skonfigurować wyprowadzenia ***PC0*** i ***PC1***.

2.2.1. Konfiguracja modułu LPUART oraz standardowego wejścia/wyjścia (3 pkt.)

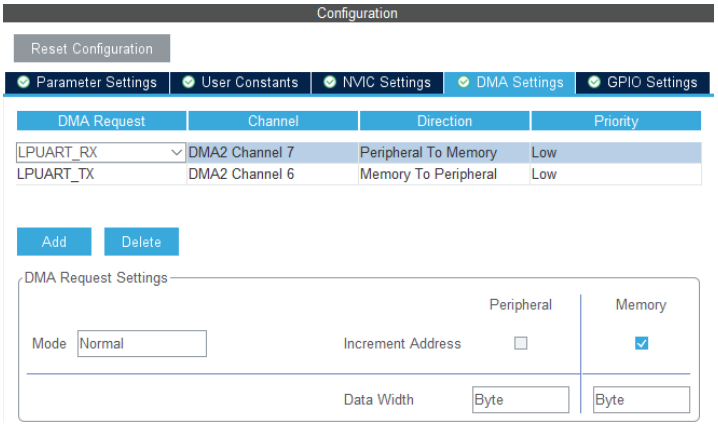
W celu skonfigurowania modułu *LPUART* należy w projekcie otworzyć plik ***BasicInterfacesHAL.ioc*** co spowoduje otworzenie okna konfiguratora w widoku *Pinout & Configuration*. Należy rozwinąć po lewej stronie pole *Conectivity* i wybrać interfejs ***LPUART1***. Następnie na prawo w oknie *LPUART1 Mode and Configuration* w sekcji *Mode* w polu wyboru *Mode* należy wybrać ***Asynchronus*** (standardowy tryb pracy). Poniżej w sekcji *Configuration* zakładce *Parameter Settings* należy skonfigurować w grupie *Basic Parameters*: prędkość transmisji (*Baud Rate*) ***115200***, długość słowa (*Word Length*) ***8 Bits (including Parity)***, parzystość (*Parity*) ***None***, liczba bitów stop (*Stop Bits*) ***1***; w grupie *Advaneced Features* zmienić *Overrun* na ***Disable***. Następnie należy otworzyć zakładkę *NVIC Settings* i zaznaczyć pole ***Enable*** *przy* *LPUART1 global interrupt*. W zakładce *DMA Settings* należy korzystając z przycisku *Add* dodać dwa kanały do transmisji w trybie DMA wybierając odpowiednie pozycji *DMA Request*: dla ***LPUART_RX*** będzie to *DMA2 Channel 7*, a dla ***LPUART_TX*** będzie to *DMA2 Channel 6*. Pozostałe parametry należy pozostawić bez zmian. Zapisać



konfigurację i zezwolić na wygenerowanie kodu startowego aplikacji. Automatycznie wygenerowany kod ponadto konfiguruje wyprowadzenia mikrokontrolera ***PC0*** i ***PC1***, które łączą go z układem programatora/konwertera.



Dodanie obsługi interfejsu szeregowego w funkcjonalności biblioteki standardowej STDIO języka C wymaga dodania implementacji metod ***__io_putchar*** (wejście) oraz ***__io_getchar*** (wyjście). *Uwaga: każda nazwa funkcji zaczyna się dwoma znakami „_”*. W tym celu należy otworzyć plik *Core->Src->syscalls.c*, dodać plik nagłówkowy *usart.h* za pomocą dyrektywy *#include* w sekcji /* Includes */, dodać dyrektywę ***#define STDIO_ECHO_ENABLED 1*** oraz w sekcji /* Functions */ dodać poniższy kod:




```
45 /* Functions */
46 int __io_putchar(int ch){
47     HAL_UART_Transmit(&hlpuart1, (uint8_t*) &ch, 1, HAL_MAX_DELAY);
48     return ch;
49 }
50 int __io_getchar(void) {
51     uint8_t ch =0;
52
53     __HAL_UART_CLEAR_OREFLAG(&hlpuart1);
54     HAL_UART_Receive(&hlpuart1, &ch, 1, HAL_MAX_DELAY);
55 #if STDIO_ECHO_ENABLED
56     HAL_UART_Transmit(&hlpuart1, &ch, 1, HAL_MAX_DELAY);
57 #endif
58     return ch;
59 }
```

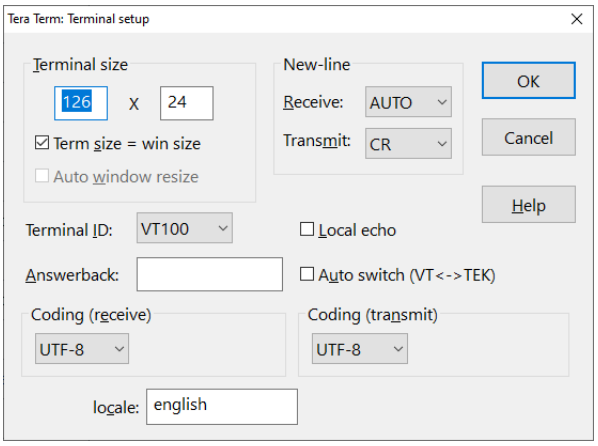
Funkcja ***__io_putchar*** odpowiedzialna jest za wysyłanie znaków. W linii 47 wywoływana jest funkcja ***HAL_UART_Transmit***, która rozpoczyna wysyłanie **1** znaku ***ch***. Z racji podania parametru ***Timeout*** wartości ***HAL_MAX_DELAY*** funkcja ta będzie blokować do momentu skutecznego wysłania danych. W funkcji ***__io_getchar*** w pierwszej kolejności kasujemy przepełnienie bufora odbiorczego (linia 54), a następnie za pomocą funkcji ***HAL_UART_Receive*** podjęta jest próba odebrania ***1*** znaku ***ch***. Z racji podania parametru ***Timeout*** wartości ***HAL_MAX_DELAY*** funkcja ta będzie blokować do momentu skutecznego odebrania danych. Jeżeli stała ***STDIO_ECHO_ENABLED*** ma wartość różną od 0 wtedy zostanie odesłany zwrotnie odebrany znak. Jest to tak zwane *echo*.

W celu przetestowania funkcjonalności należy uzupełnić plik ***main.c*** przez: dodanie odwołania do pliku nagłówkowego ***stdio.h***, ***string.h, math.h*** w sekcji /* USER CODE BEGIN Includes */ oraz modyfikację kodu funkcji ***main*** znajdującej się w pliku *Core->Src->main.c* w sekcji /* USER CODE BEGIN WHILE */ poniższą zawartością. Kluczowym dla prawidłowej pracy jest wywołanie funkcji ***setvbuf*** (linia 100) odpowiedzialnej za ustawienie wielkości bufora odbieranych znaków na 0. Domyślnie ustawiony jest on na 1024 znaki. Za pomocą funkcji ***scanf*** wczytywana jest wpisana liczba całkowita do zmiennej ***value***. Następnie za pomocą funkcji ***SEG_DisplayDec*** jest prezentowana na wyświetlaczu 7-segmentowym.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
uint16_t value;
setvbuf(stdin, NULL, _IONBF, 0);
while (1) {
    printf("\nPodaj liczbe do wyświetlenia: ");
    scanf("%d", (int*) (&value));
    SEG_DisplayDec(value);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Przygotowany program należy skompilować, uruchomić i przetestować we współpracy z programem obsługującym interfejs szeregowy np. ***Tera Term***. Po uruchomieniu aplikacji należy z menu wybrać *File->New connection*, zaznaczyć ***Serial*** i wybrać właściwy port szeregowy. Następnie wybrać w menu *Setup->Terminal* i ustawić *Receive:* ***Auto*** oraz *Transmit:* ***CR***. Na zakończenie konfiguracji wybrać w menu *Setup->Serial Port* i ustawić *Speed:* ***115200***. Działanie programu przedstawić prowadzącemu do ocenienia. Po zatwierdzeniu wykonać migawkę z komentarzem „***Zadanie 2.2.1 – standardowe wejście/wyjście***”.



2.2.2. Transmisja danych w trybie blokowania i przerwania (3 pkt.)

Funkcje biblioteki ***HAL_UART_Transmit*** i ***HAL_UART_Receive*** umożliwiają odpowiednio wysyłanie i odbieranie wskazanej ilości słów w określonym czasie. Jeżeli nie będzie to możliwe funkcja zwróci stosowny kod błędu wykonania. Taka konstrukcja pozwala na pisanie kodu, który będzie blokował możliwość wykonania innych instrukcji przez mikrokontroler. W związku z czym ich użycie preferowane dla transmisji małych porcji danych. Spowodowane jest to przez wartość ostatniego parametru tych funkcji o nazwie ***Timeout***. Kiedy chcemy wysłać lub odebrać większą ilość danych warto użyć odpowiednio funkcji ***HAL_UART_Transmit_IT*** i ***HAL_UART_Receive_IT***. Ich wywołanie inicjuje transmisje danych, która jest później kontynuowana w odpowiedniej bibliotecznej funkcji obsługi przerwań dla danego modułu UART. W ten sposób nie ma blokowania i oczekiwania na transmisje przez stosunkowo wolny interfejs. Dla typowych nastaw interfejsu szeregowego (115200, 8, N, 1) transmisja jednego 8-bitów informacji wymaga 8680,5 ns co przy zegarze taktującym mikrokontroler (4 MHz) o okresie 250 ns oznacza możliwość wykonania maksymalnie 34 instrukcji. Przy maksymalnej możliwej częstotliwości taktowania wynoszącej 80 MHz można w tym czasie wykonać do 680 instrukcji. Programista może dodać funkcję ***HAL_UART_TxCpltCallback*** *i* ***HAL_UART_RxCpltCallback***, które zostaną wywołane na zakończenie transmisji danych. Ponadto można zaimplementować funkcje ***HAL_UART_TxHalfCpltCallback*** oraz ***HAL_UART_RxHalfCpltCallback***, które zostaną wywołane po wysłaniu połowy danych.

Poniższy kod, który należy dodać w pliku ***main.c***, pozwoli zaobserwować wpływ blokowania na wykonywanie programu w porównaniu do pracy z użyciem przerwania. W pierwszej kolejności należy dodać definicję symbolu ***TEST_UART_IT***:

```
/* USER CODE BEGIN PD */
#define TEST_UART_IT 0
/* USER CODE END PD */
```

Należy również dodać deklarację publicznego bufora znaków na odebrane dane o nazwie ***rxBuffer*** o wielkości 16 znaków w sekcji /* USER CODE BEGIN PV */:

```
/* USER CODE BEGIN PV */
uint8_t rxBuffer[16];
/* USER CODE END PV */
```

W pętli głównej ***while***, funkcji ***main***, w sekcji /* USER CODE BEGIN WHILE */ należy umieścić poniższy kod:

```
111   /* USER CODE BEGIN WHILE */
112   while (1) {
113     HAL_Delay(1000);
114     LED_Toggle(0);
115 #if TEST_UART_IT
116     LED_On(7);
117     HAL_UART_Receive_IT(&hlpuart1, rxBuffer, 8);
118 #else
119     LED_On(6);
120     HAL_UART_Receive(&hlpuart1, rxBuffer, 8, HAL_MAX_DELAY);
121     LED_Off(6);
122 #endif
123     /* USER CODE END WHILE */
124     /* USER CODE BEGIN 3 */
125   }
126   /* USER CODE END 3 */
```

Następnie w należy w sekcji /* USER CODE BEGIN 4 */ dodać kod funkcji ***HAL_UART_RxCpltCallback***:

```
174 /* USER CODE BEGIN 4 */
175 void HAL_UART_RxCpltCallback(UART_HandleTypeDef * huart){
176     if(huart->Instance == LPUART1){
177         LED_Off(7);
178         HAL_UART_Transmit_IT(&hlpuart1, rxBuffer, strlen((char *)rxBuffer));
179     }
180 }
```

W przypadku, kiedy stała ***TEST_UART_IT*** ma wartość **0** dioda ***LED0*** będzie zmieniać swój stan po każdorazowym odebraniu 8 znaków. Dioda *LED6* będzie się świecić przez czas potrzebny na odebranie 8 znaków. Zmiana wartości symbolu ***TEST_UART_IT*** na **1** spowoduje, że zamiast diody ***LED6*** zapali się dioda ***LED7***, która zgaśnie po odebraniu 8 znaków. Różnica w działaniu obu konfiguracji polega na tym, że w drugim przypadku dioda ***LED0*** cały czas zmienia swój stan co 1000 ms niezależnie czy odbierane są znaki. Spowodowane jest to przez fakt, że po wywołaniu funkcji w linii 117 odbieranie jest rozpoczęte. Sam odczyt z rejestru odbiorczego wykonywany jest w funkcji obsługi przerwania modułu ***LPUART1***.

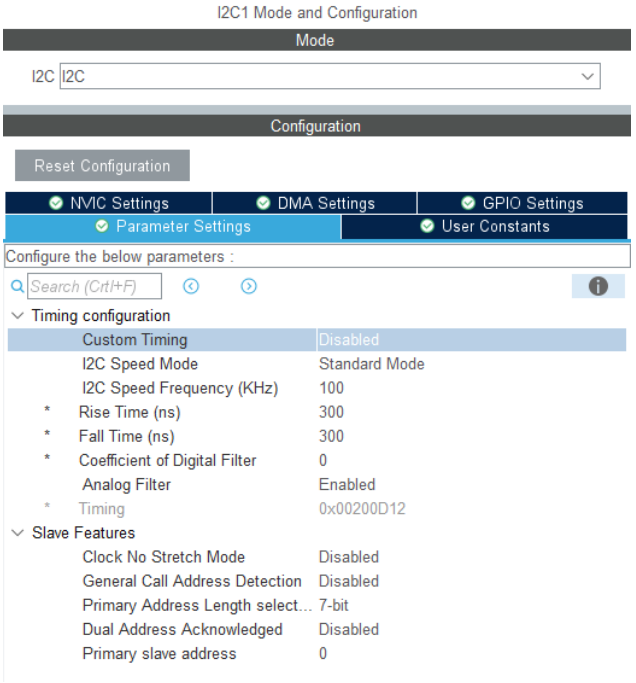
Przygotowany program należy skompilować, uruchomić i przetestować we współpracy z programem obsługującym interfejs szeregowy np. *Tera Term* dla dwóch różnych trybów pracy przez zmianę wartości symbolu ***TEST_UART_IT***. Działanie programu przedstawić prowadzącemu do ocenienia. Po zatwierdzeniu wykonać migawkę z komentarzem „***Zadanie 2.2.2 – wysyłanie w trybie blokowania i przerwania***”.

2.3. Obsługa interfejsu I2C (6 pkt.)

Na płycie *KAmeLeon* dostępne są 2 układy cyfrowe komunikujące się z mikrokontrolerem z użyciem interfejsu I2C. Pierwszy z nich to cyfrowy czujnik temperatury *STLM75M2F* dostępny pod adresem 0x48 lub 0x49 w zależności od konfiguracji zwory JP17, podłączony do interfejsu *I2C1*. Drugi układ to *LSM303C* moduł e-kompasu wyposażony w 3-osiowy akcelerometr (adres 0x1D) i 3-osiowy magnetometr (adres 0x1E) podłączony do interfejsu *I2C3*. Interfejs I2C jest dostępny w mikrokontrolerze STM32L496ZGT6 w czterech modułach. Najważniejsze z nich to wsparcie dla adresowania o długości 7 i 10 bitów, prędkości transmisji do 1 Mb/s, obsługa protokołu SMBus/PMBus. Na płycie *KAmeLeon* interfejs *I2C1* jest użyty na wyprowadzeniach ***PG13*** (*SDA*), ***PG14*** (*SCL*), natomiast interfejs *I2C3* jest dostępny na wyprowadzeniach ***PG7*** (*SCL*) i ***PG8*** (*SDA*).

2.3.1. Konfiguracja modułu I2C1 i odczyt temperatury (3 pkt.)

W celu skonfigurowania modułu *I2C1* należy w projekcie otworzyć plik ***BasicInterfacesHAL.ioc*** co spowoduje otworzenie okna konfiguratora w widoku *Pinout & Configuration*. Należy rozwinąć *Conectivity* i wybrać ***I2C1***. Następnie po prawej stronie w oknie *I2C1 Mode and Configuration* w sekcji *Mode* w polu wyboru *I2C* należy wybrać ***I2C***. Poniżej w sekcji *Configuration* zakładce *Parameter Settings* należy skonfigurować w grupie *Timing Configuration*: *Rise Time* oraz *Fall Time* na ***300***, a pozostałe opcje pozostawić bez zmian. Wyprowadzeniom ***PG13*** i ***PG14*** należy nadać odpowiednio etykiety ***TEMP_LM75_I2C1_SDA*** i ***TEMP_LM75_I2C1_SCL***. Ponadto układ ten podłączony jest do wejścia ***PG15*** (etykieta ***TEMP_LM75_INT***), które służy do poinformowaniu mikrokontrolera o przekroczeniu ustawionej temperatury. Pozostałe parametry należy pozostawić bez zmian. Zapisać konfigurację i zezwolić na wygenerowanie kodu startowego aplikacji. Automatycznie wygenerowany kod ponadto konfiguruje wyprowadzenia mikrokontrolera.



Do obsłużenia czujnika temperatury należy dodać poniższe definicje symboli za pomocą dyrektywy ***#define*** w sekcji /* USER CODE BEGIN PD */. Pierwsza z nich przechowuje adres na magistrali I2C, cztery kolejne to adresy rejestrów wewnętrznych układu, a ostatnia przedstawia rozdzielczość przetwornika. Do odczytu temperatury wystarczy odczyt 2 bajtów z rejestru ***TEMP_LM75_TEMP_REG*** oraz przeliczenie na rzeczywistą wartość odebranych danych. Odczyt nie powinien odbywać się jednak częściej niż co 150 ms. Więcej informacji można znaleźć w dokumentacji do układu.

```
/* USER CODE BEGIN PD */
#define TEMP_LM75_I2C_ADDR        0x48
#define TEMP_LM75_TEMP_REG        0x00
#define TEMP_LM75_CONF_REG        0x01
#define TEMP_LM75_THYS_REG        0x02
#define TEMP_LM75_TOS_REG         0x03

#define TEMP_LM75_RESOLUTION      0.5f
```

Powyższe definicje zostaną użyte w funkcjach, które zapewniają odczyt ze wskazanego rejestru. Należy je dopisać do sekcji /* USER CODE BEGIN 0 */. Pierwsze dwie funkcje (***LM75_ReadReg8*** i ***LM75_ReadReg16***) umożliwiają odczyt odpowiednio 8 i 16 bitów ze wskazanego w parametrze funkcji rejestru układu czujnika. W celu komunikacji z układem użyta została biblioteczna funkcja ***HAL_I2C_Mem_Read***, która najpierw wpisuje numer rejestru, a później odczytuje dane. Funkcje ***TEMP_GetInt*** oraz ***TEMP_GetFloat*** pozwalają na odczytanie rzeczywistej temperatury wyrażonej w stopniach Celsjusza, przy czym druga funkcja zwraca wartość z dokładnością ± 0,5°C.

```
/* USER CODE BEGIN 0 */
uint8_t LM75_ReadReg8(uint8_t regAddr) {
  uint8_t regValue = 0;
  if(HAL_I2C_Mem_Read(&hi2c1, TEMP_LM75_I2C_ADDR << 1, regAddr, 1, &regValue, 1, 1000) != HAL_OK) {
    printf("Error reading register 0x%02X\r\n", regAddr);
  } else {
    printf("Register 0x%02X = 0x%02X\r\n", regAddr, regValue);
  }

  return regValue;
}

uint16_t LM75_ReadReg16(uint8_t regAddr) {
  uint8_t regValue[2];
  uint16_t value = 0;

  if(HAL_I2C_Mem_Read(&hi2c1, TEMP_LM75_I2C_ADDR << 1, regAddr, 1, regValue, 2, 1000) != HAL_OK) {
    printf("Error reading register 0x%02X\r\n", regAddr);
  } else {
    value = (((uint16_t)regValue[0] << 8) | regValue[1]);
    printf("Register 0x%02X = 0x%04X\r\n", regAddr, value);
  }

  return value;
}

int16_t TEMP_GetInt(void){
  return (LM75_ReadReg16(TEMP_LM75_TEMP_REG) >> 8);
}

float TEMP_GetFloat(void){
  return ((LM75_ReadReg16(TEMP_LM75_TEMP_REG) >> 8) * TEMP_LM75_RESOLUTION);
}
```

W celu sprawdzenia poprawności odczytu temperatury należy zmodyfikować zawartość pętli ***while*** w funkcji ***main*** na poniższą. Wartość rzeczywista temperatury jest odczytywana z użyciem funkcji ***TEMP_GetFloat*** i zapisywana do zmiennej ***temp***. Następnie w celu wyświetlenia jej na wyświetlaczu 7-segmentowym pomnożona jest przez 10 i przekazana jako pierwszy parametr funkcji ***BSP_SEG_DisplayFixedPoint*** po wcześniejszej zmianie na liczbę całkowitą. Z racji, że drugi parametr ustawiony został na 1 to kropka dziesiętna zapali się na module 1. Ponadto korzystając z funkcji ***printf*** zademonstrowane jest w jaki sposób można zmienić liczbę zmiennoprzecinkową na dwie liczby całkowite.

```
/* USER CODE BEGIN WHILE */
float temp;
while (1) {
  temp = TEMP_GetFloat();
  BSP_SEG_DisplayFixedPoint((int16_t)(temp * 10.0f), 1);
  printf("Temperature: %d,%d [degC]\r\n", (int)temp, (temp == roundf(temp) ? 0 : 5));
  HAL_Delay(1000);
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

Przygotowany program należy skompilować, uruchomić i przetestować. Zmianę temperatury można wywołać przez przyłożenie palca do czujnika temperatury. Działanie programu przedstawić prowadzącemu do ocenienia. Po zatwierdzeniu wykonać migawkę z komentarzem „***Zadanie 2.3.1 – konfiguracja modułu I2C1 i odczyt temperatury***”.

2.3.2. Konfiguracja modułu I2C3 i odczyt z układu akcelerometru (3 pkt.)

W celu skonfigurowania modułu *I2C3* należy w projekcie otworzyć plik ***BasicInterfacesHAL.ioc*** co spowoduje otworzenie okna konfiguratora w widoku *Pinout & Configuration*. Należy rozwinąć *Conectivity* i wybrać ***I2C3***. Następnie po prawej stronie w oknie *I2C3 Mode and Configuration* w sekcji *Mode* w polu wyboru *I2C* należy wybrać ***I2C***. Poniżej w sekcji *Configuration* zakładce *Parameter Settings* należy skonfigurować w grupie *Timing Configuration: I2C Speed Mode* ustawić na ***Fast Mode***, *Rise Time* oraz *Fall Time* ustawić na **300**, a pozostałe opcje pozostawić bez zmian. Następnie należy odszukać wyprowadzenie ***PG7***, wybrać opcje ***I2C3_SCL*** i nadać etykietę ***LSM303C_I2C3_SCL***. Natomiast wyprowadzeniu ***PG8*** nadać etykietę ***LSM303C_I2C3_SDA***. Wprowadzone zmiany należy zapisać w celu wygenerowania niezbędnych zmian w konfiguracji projektu.

W celu przetestowania funkcjonalności należy uzupełnić plik ***main.c*** przez: dodanie odwołania do pliku nagłówkowego ***stdio.h***, ***string.h, math.h*** w sekcji /* USER CODE BEGIN Includes */. W dalszej kolejności należy dodać plik nagłówkowy ***lsm303c.h*** dyrektywą *#include* w sekcji /* USER CODE BEGIN Includes */. Zawiera on zdefiniowane symbole reprezentujące adresy rejestrów oraz większość możliwych wartości jakie mogą być w tych rejestrach ustawione. Następnie z racji, że można pozyskać dane o przyspieszeniu w trzech osiach i są one dostępne w postaci liczby typu *int16_t* zdefiniować należy strukturę ***ACC_RawData_st*** w sekcji /* USER CODE BEGIN PTD */. Pozwoli ona na opakowanie odczytanych wartości w jedną zmienną.

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
typedef struct {
  int16_t x;
  int16_t y;
  int16_t z;
} ACC_RawData_st;
```

Kolejnym elementem do dodania jest definicja symbolu ***ACC_LSM303C_I2C_ADDR***, który określa adres na magistrali I2C.

```
#define ACC_LSM303C_I2C_ADDR    0x1D
```

Następnie należy przepisać kod poniższych dwóch funkcji. Pierwsza z nich ***LSM303C_Init*** sprawdza w pierwszej kolejności czy wykryto układ przez odczyt jego identyfikatora ***ID***, a jeżeli identyfikacja wypadła pozytywnie wykonywana jest konfiguracja rejestru *CTRL_REG1_A* (częstotliwość próbkowania, wybór aktywnych osi) oraz rejestru ***CTRL_REG4_A*** (zakres pracy, automatyczna inkrementacja adresu rejestru podczas odczytu). Funkcja ***LSM303C_ReadAccRawData*** dokonuje odczytu 6 kolejnych bajtów począwszy od adresu rejestru ***OUT_X_L_A***, które zapisywane są do struktury ***acc***. W każdej funkcji w przypadku wystąpienia błędu komunikacji za pośrednictwem magistrali I2C program zostanie przekierowany do funkcji ***Error_Handler*** i wyśle stosowny komunikat przez interfejs szeregowy.

W celu sprawdzenia poprawności odczytu przyspieszenia należy zmodyfikować zawartość pętli ***while*** w funkcji ***main*** na poniższą. W pierwszej kolejności deklarujemy zmienną ***acc*** do której będziemy wczytywać wartości przyspieszenia. Następnie jednorazowo wywołana jest funkcja ***LSM303C_Init***, która przyjmuje dwa parametry. Pierwszy z nich określa częstotliwość próbkowania (ang. *Output Data Rate* - ODR), która może przyjąć wartość 10, 50, 100, 200, 400 lub 800 Hz. Drugi parametr określa zakres pomiarowy wyrażony w jednostkach przyspieszenia ziemskiego *g* (1*g* = 9,81 m/s^2) i może wynieść ±2, ±4 lub ±8 *g*. W każdej iteracji pętli ***while*** następuje zmiana stanu diody *LED0*, odczytanie wartości przyspieszenia z użyciem funkcji ***LSM303C_ReadAccDataRaw*** wysłanie za pośrednictwem interfejsu szeregowego pracującego w trybie

```
/* USER CODE BEGIN 0 */
int32_t LSM303C_Init(uint8_t accDataRate, uint8_t accFullScale) {
  uint8_t tmpReg = 0x00;
  int32_t status;

  // Read device ID and check if the device is correct
  if((status = HAL_I2C_Mem_Read(&hi2c3, ACC_LSM303C_I2C_ADDR << 1,
          LSM303C_WHO_AM_I_ADDR, 1, &tmpReg, 1, 1000)) != HAL_OK) {
    printf("Error: I2C3! - %x\r\n", (unsigned int)status);
    Error_Handler();
  } else {
    if(tmpReg != LMS303C_ACC_ID) {
      printf("Error: LSM303C not found! (0x%02X)\r\n", tmpReg);
      Error_Handler();
    } else {
      printf("LSM303C found! (0x%02X)\r\n", tmpReg);
      // Enable all axes and set data rate
      tmpReg = (accDataRate | LSM303C_ACC_AXES_ENABLE);
      status |= HAL_I2C_Mem_Write(&hi2c3, ACC_LSM303C_I2C_ADDR << 1,
              LSM303C_CTRL_REG1_A, 1, &tmpReg, 1, 1000);
      // Set full scale
      tmpReg = accFullScale | 0x04;
      status |= HAL_I2C_Mem_Write(&hi2c3, ACC_LSM303C_I2C_ADDR << 1,
              LSM303C_CTRL_REG4_A, 1, &tmpReg, 1, 1000);
    }
  }

  return status;
}

void LSM303C_ReadAccRawData(ACC_RawData_st *accData) {
  int32_t status;

  if((status = HAL_I2C_Mem_Read(&hi2c3, ACC_LSM303C_I2C_ADDR << 1,
          LSM303C_OUT_X_L_A | 0x80, 1, (uint8_t *)accData, 6, 1000)) != HAL_OK) {
    printf("Error: I2C3! - %x\r\n", (unsigned int)status);
    Error_Handler();
  }
}
```

bezpośredniego dostępu do pamięci DMA. Tryb ten różni się od trybu IT tym, że przesłanie danych z pamięci do rejestru wysyłania nie wymaga wykonywania instrukcji przez rdzeń procesora, lecz dzieje się sprzętowo.

Przygotowany program należy skompilować, uruchomić i przetestować we współpracy z programem obsługującym interfejs szeregowy np. *Tera Term* oraz programem *Serial Plot* umożliwiającym wizualizację odebranych danych. W katalogu *SerialPlotCfg* umieszczona został plik konfigurujący program, który można wczytać za pomocą menu *File->Load File* i wskazanie pliku *Zad-2.3.2.ini*. Działanie programu przedstawić prowadzącemu do ocenienia. Po zatwierdzeniu wykonać migawkę z komentarzem „***Zadanie 2.3.2 – konfiguracja modułu I2C3 i odczyt z układu akcelerometru***”.

```
/* USER CODE BEGIN WHILE */
ACC_RawData_st acc;
LSM303C_Init(LSM303C_ACC_ODR_10_HZ, LSM303C_ACC_FULLSCALE_2G);
while (1) {
  LED_Toggle(0);
  LSM303C_ReadAccRawData(&acc);
  HAL_UART_Transmit_DMA(&hlpuart1, (uint8_t *)&acc, 6);
  HAL_Delay(100);
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```

3. Zadania rozszerzające do realizacji (14 pkt.)

W rozdziale tym przedstawione zostały zadania dodatkowe, za realizację których można podnieść ocenę końcową za wykonane ćwiczenie. Ich realizację należy wykonać w dotychczasowym projekcie.

3.1. Rozszerzona obsługa interfejsu szeregowego

3.1.1. Konsola wiersza poleceń

W zadaniu tym należy zaimplementować możliwość wykonywania komend przesłanych z komputera, które spowodują wykonanie odpowiedniej akcji przez mikrokontroler. Komendy można podzielić na dwa rodzaje ustawiające (‘*=*’) oraz pytające (‘*?*’). Należy zaimplementować obsługę komendy ustawiającej w postaci „*<komenda>=<wartość>**r**n*”. Gdzie komenda to „*LED*”, a wartość może być liczbą od 0 do 7 i spowoduje zmianę stanu odpowiedniej diody. Ponadto należy zaimplementować obsługę komendy pytającej w postaci „*<komenda>?**r**n*”. Gdzie komenda to „*TEM*” i w odpowiedzi otrzymujemy wartość temperatury z czujnika *STLM75M2F*. Wykonanie komendy potwierdzane jest odesłaniem do komputera „OK*r**n*”. Znak końca linii jest informacją o długości komendy.

Przygotowany program należy skompilować, uruchomić i przetestować. Działanie programu przedstawić prowadzącemu zajęcia. Po zatwierdzeniu i ocenieniu wykonać migawkę z komentarzem „***Zadanie 3.1.1 – konsola wiersza poleceń***”.

3.1.2. Generowanie próbek funkcji sinus

W zadaniu tym należy przygotować program, który będzie wysyłał do komputera kolejne próbki funkcji sinus wyznaczane dla kąta, który będzie zwiększany o stałą wartość koku fazy (*phaseStep*) 100 razy na sekundę. Krok fazy będzie domyślnie ustawiony tak aby 100 próbek reprezentowało 1 okres funkcji sinus ($phaseStep = 2\pi f/100$, dla jednego okresu $f = 1$). Za pomocą joysticka należy umożliwić zmianę wartości zmiennej *f* w zakresie od 0 do 100*,* którą później należy uwzględnić w wyznaczeniu nowej wartości zmiennej *phaseStep*. Wartość zmiennej *f* należy prezentować na wyświetlaczu 7-segmentowym. Wysyłana wartość powinna być reprezentowana w formacie tekstowym z dokładnością trzech miejsc o przecinku.

Przygotowany program należy skompilować, uruchomić i przetestować. Działanie programu przedstawić prowadzącemu zajęcia. Po zatwierdzeniu i ocenieniu wykonać migawkę z komentarzem „***Zadanie 3.1.2 – generowanie próbek funkcji sinus***”.

3.2. Rozszerzona obsługa interfejsu I2C

3.2.1. Zaawansowane funkcje czujnika temperatury STLM75F

Na bazie zadania 2.3.1 oraz w oparciu o dokumentację układu STLM75M2F należy skonfigurować wartości rejestrów T_{HYS} = 26,0 °C oraz T_{HYS} = 27,0 °C. Linii *LM75_INT* podłączona do portu PG15 mikrokontrolera będzie ustawiana w stan niski za każdym razem po przekroczeniu temperatury 27,0 °C i ustawiana w stan wysoki, kiedy temperatura spadnie poniżej 26,0 °C. Z racji, że pomiar odbywa się automatycznie i trwa 150 ms można przyjąć, że należy sprawdzać stan portu PB15 co 200 ms. Do tego celu można rozbudować funkcję *HAL_SYSTICK_Callback* tak aby sprawdzać stan PG15 i kiedy jest niski zapalić diodę LED7 i zgasić ją, kiedy na PG15 będzie stan wysoki. Cały czas na wyświetlaczu 7-segmentowym odświeżana jest wartość odczytywanej temperatury co 1000 ms

oraz będzie ona wysyłana za pośrednictwem interfejsu szeregowego w postaci tekstowej z dokładnością jednego miejsca po przecinku.

Przygotowany program należy skompilować, uruchomić i przetestować przez przyłożenie palca do czujnika temperatury. Spowodować powinno to wzrost temperatury. Działanie programu przedstawić prowadzącemu zajęcia. Po zatwierdzeniu i ocenieniu wykonać migawkę z komentarzem „***Zadanie 3.2.1 – alarm temperaturowy***".

3.2.2. Zaawansowane funkcje czujnika przyspieszenia LSM303C

Na bazie zadania 2.3.1 oraz w oparciu o dokumentację układu *LSM303C* należy skonfigurować układ do pracy z kolejką *FIFO* w trybie *Stream*, która umożliwia zbieranie do 32 próbek i jednocześnie umożliwia rzadsze odpytywanie mikrokontrolera o przesłanie danych. W tym celu należy skonfigurować rejestr *CTRL_REG3_A* = 0x80, *FIFO_CTRL* = 0x40. Przy częstotliwości próbkowania 10 Hz można ograniczyć częstotliwość odczytu danych o przyspieszeniu do pojedynczego wywołania funkcji raz na sekundę. Liczbę próbek jakie znajdują się w kolejce można odczytać z rejestru *FIFO_SRC* na podstawie pięciu najmniej znaczących bitów. Odebrane dane należy zapisywać do tablicy typu *ACC_RawData_st*, która może pomieścić co najmniej 10 próbek. Następnie próbki należy przesłać z użyciem interfejsu szeregowego do komputera.

Przygotowany program należy skompilować, uruchomić i przetestować. Działanie programu przedstawić prowadzącemu zajęcia. Po zatwierdzeniu i ocenieniu wykonać migawkę z komentarzem „***Zadanie 3.2.2 – praca z buforowaniem próbek***".