Ćwiczenie 5

PROGRAMOWANIE SYSTEMÓW WBUDOWANYCH

Obsługa i konfiguracja komunikacji szeregowej za pośrednictwem układu UART mikrokontrolera Cortex-M4 STM32F303RE lub STM32F411RE.

Celem ćwiczenia jest zapoznanie studenta z metodyką programowania, projektowania i tworzenia aplikacji na mikrokontrolery rodziny ARM Cortex-M4,. W trakcie ćwiczenia student nabędzie podstawowe informacje dotyczące środowiska i umiejętności posługiwania się nim oraz programowania komunikacji szeregowej - UART.

Zakład Systemów Informacyjno-Pomiarowych

IETiSIP, Wydział Elektryczny, PW

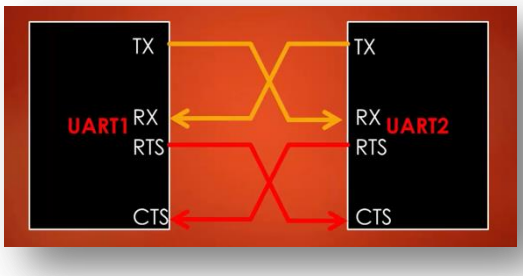
Ćwiczenie 5 polegać będzie na konfiguracji komunikacji szeregowej za pośrednictwem układu UART i nawiązaniu połączenia z komputerem PC.

Komunikacja szeregowa **UART** jest komunikacją asynchroniczną powszechnie wykorzystywaną przez komputery klasy PC. Współczesne mikrokontrolery, w tym i STM32 posiadają możliwość łączności (i bardzo często z niej korzystają) z wykorzystaniem linii zegarowej, taktującej wykorzystującej układ **USART** (o możliwościach jak rozszerzonych względem **USART**)

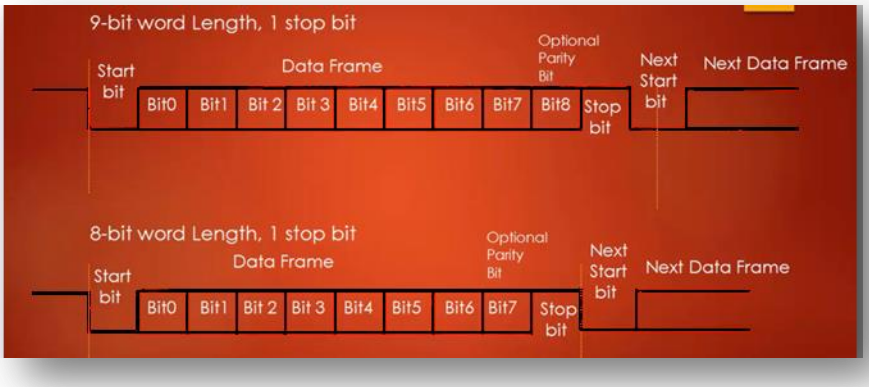
UART - Universal Asynchronous Receiver and Transmitter
USART - Universal Synchronus and Asynchronous Receiver and Transmitter

Charakterystyka interfejsu:
- prędkości: 9600 b/s do 115200 b/s
- nieadresowalny: 1 nadajnik -> 1 odbiornik
- linie danych: -15…-3V -> 1 logiczna (logika ujemna) – konwerter ADM232
- linie sterujące: 3…15V -> 1 logiczna (logika dodatnia) – konwerter ADM232
- zasięg do 15m (19200 b/s)
- niesymetryczny obwód transmisji

Układ UART wyposażony jest w dwie linie, za pomocą których realizowane jest nadawanie (Tx) i odbieranie (Rx) danych. W przypadku transmisji asynchronicznej nadajnik i odbiornik muszą zostać uprzednio odpowiednio i jednakowo zaprogramowane. Dotyczy to prędkości transmisji, liczby bitów danych (od 5 do 9), podstawowego mechanizmu kontroli poprawności komunikacji – parzystości (suma bitów ustawionych – 1 łącznie z bitem parzystości może być parzysta albo nieparzysta), liczby bitów stopu (1, 1.5, 2).



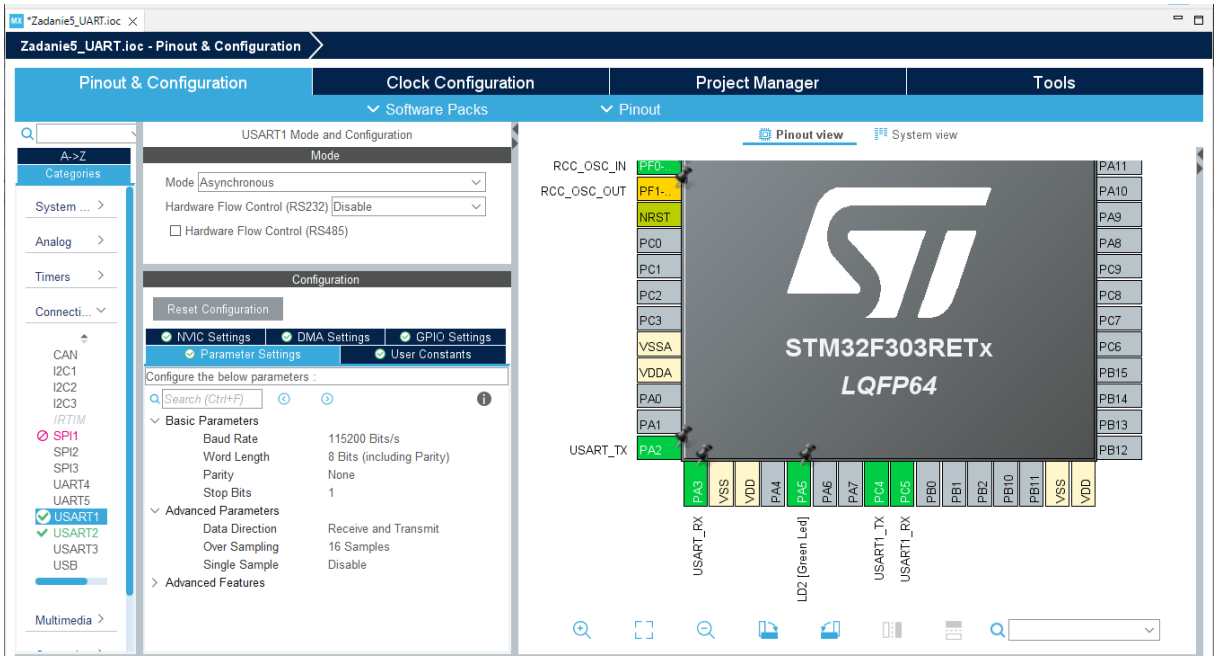
Rysunek 1. Połączenie układów UART do komunikacji nadawczo-odbiorczej. W wersji uproszczonej można wykorzystać jedynie linie RX oraz TX. Linie RTS oraz CTS jeśli są wyprowadzone zwiera się w jednym i drugie urządzeniu [Źródło: http://fastbitlab.com/stm32-pll-programming-fundamentals-2].



Rysunek 2. Ramka danych interfejsu UART
[Źródło: http://fastbitlab.com/stm32-pll-programming-fundamentals-2].

Na rysunku 2 przedstawiona jest ramka danych interfejsu szeregowego UART. Stan bezczynności reprezentowany jest utrzymującym się wysokim poziomem napięcia na wyjściu układu. Pojawiający się bit startu oznacza zmianę stanu wysokiego na niski na linii danych. Bit/bity stopu mają charakter przeciwny czyli zmianę na/utrzymanie stanu wysokiego na transmisyjnej linii danych.

Konfiguracja komunikacji USART w ćwiczeniu wygląda następująco. W zakładce **Pinout&Configuration-> Connectivity** należy włączyć **USART1** i sprawdzić ustawienia jak na rysunku 3. Linie **Tx** i **Rx** dostępne są na wyprowadzeniach odpowiednio PC4 oraz PC5 (STM32F303).



Rysunek 3. Konfiguracja USART1.

Parametry łącza należy ustawić następująco:

- **Baud Rate -** prędkość transmisji. Dla wielu urządzeń wartością maksymalną jest 115200 i taką należy wybrać chyba, że drugi punkt końcowy obsługuje mniejsze wartości.
- **Word Length –** liczba bitów danych włączając bit parzystości, które przesyłane są w ramce. Warto pozostawić wartość domyślną tj. 8 bitów razem z bitem parzystości.
- **Parity –** włączenie mechanizmu sprawdzającego poprawność transmisji. W ćwiczeniu pozostaje wyłączona.
- **Stop Bits -** wartością typową jest 1 stop bitu.
- **Data Direction –** możliwość wybrania transmisji dwu- ale też i jednokierunkowej**,** Tu też pozostaje wartość domyślna tj. **Receive and Transmit**.
- **Over Sampling –** możliwość wielokrotnego odczytu wartości poszczególnych bitów w ramce. Ma to na celu redukcję potencjalnych błędów. Tu również warto pozostawić wartość domyślną tj. 16.

Pozostałe parametry również pozostają ustawione na wartości domyślne. Zatem konfiguracja portu szeregowego polegała właściwie tylko na jego włączeniu (i ustawieniu prędkości). Należy pamiętać aby dokładnie te same parametry ustawić w drugim urządzeniu komunikacyjnym.

Zadania do zrealizowania w ramach ćwiczenia:

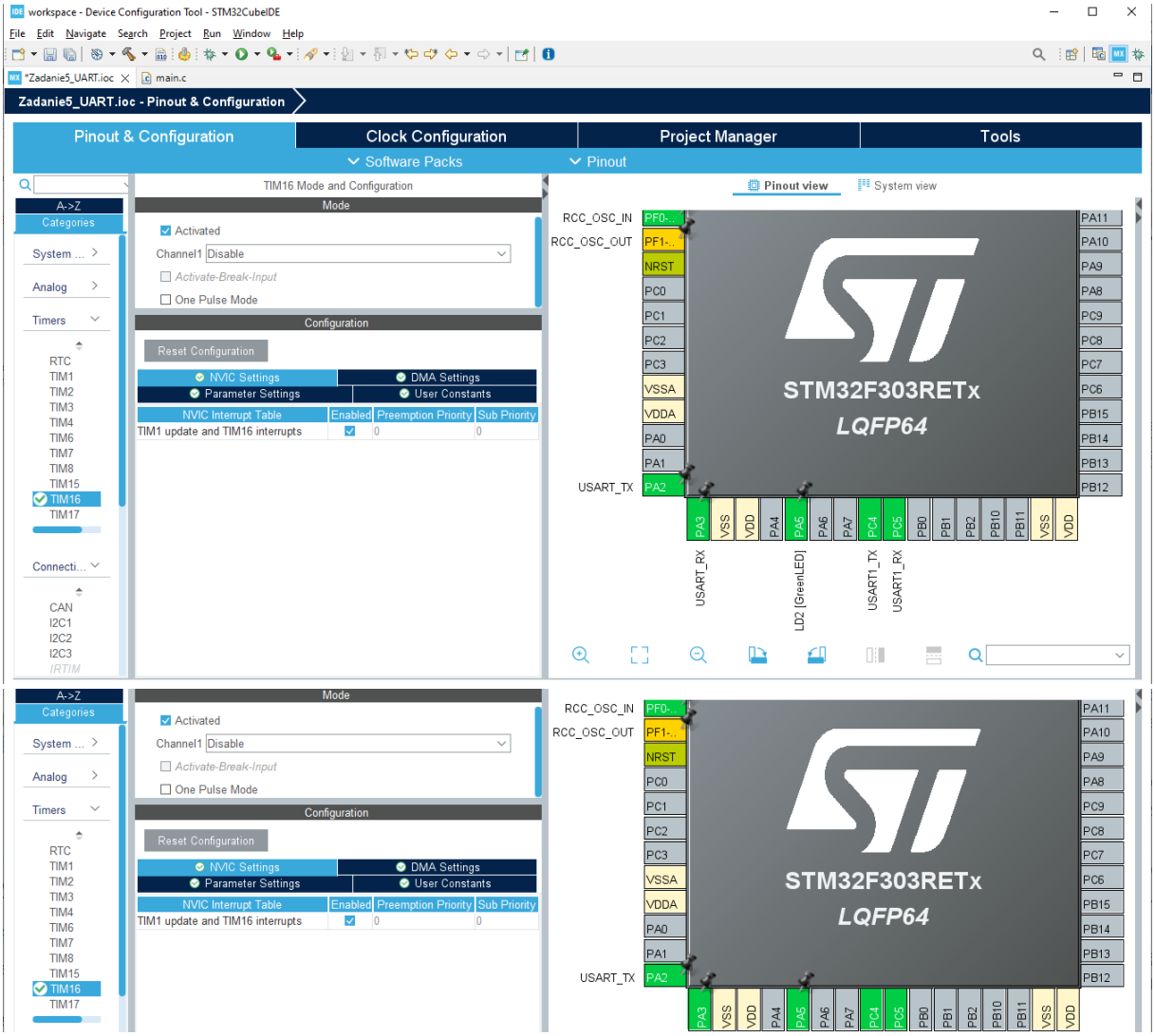
- wysyłanie komunikatów STM->PC
- odbieranie komunikatów PC ->STM (przerwania)
    - komunikaty krótkie o stałej długości jednego bajtu
    - komunikaty o długości większej niż jeden bajt
        - stała długość komunikatu
        - zmienna długość, kody sterujące
            - założenie pewnej maksymalnej długości, założenie znaku terminującego – końca komendy i uzupełnianie zerami pozostałej części, wysyłanie zawsze całego bufora uzupełnionego zerami na końcu
            - wysyłanie znak po znaku aż do momentu wysłania/odebrania założonego znaku terminującego
    - transmisja z wykorzystaniem DMA

W ćwiczeniu przesyłane dane będą miały postać prostych komunikatów zawierających informację o kolejnym numerze danej transmisji.

Wysyłanie komunikatów STM->PC:

Cykliczne wysyłanie komunikatów z STM można zrealizować korzystając z podstawowej funkcji HAL_Delay, warto jednak wykorzystać timer – rysunek

4 (STM32F303 – TIM16 w sposób analogiczny ćwiczenia z timerami) i skonfigurować go tak aby generował przerwanie np. co 1s (ustawienia PSC oraz ARR).



Rysunek 4. Okno konfiguracyjne układu timera z aktywacją przerwania.

Dodatkowo można wykorzystać linię cyfrową do której podłączona jest dioda LED i w ten sposób sygnalizować chwile przerwań timer'owych i transmisji. W przykładzie nadana jest etykieta **GreenLED**. Po wygenerowaniu kodu pojawi się nowa funkcja obsługująca przerwanie timer'owe wywoływane cyklicznie po upływie zadanego okresu. Funkcja ta nazywa się `HAL_TIM_PeriodElapsedCallback` i została umieszczona w pliku z funkcjami do obsługi timerów tj. stm32f3xx_hal_tim.c. Można też odnaleźć ją wybierając z meny **Search->File**... . Można tę funkcję zdefiniować (i zdeklarować) w pliku main.c, tak aby w ćwiczeniu cały kod źródłowy znajdował w jednym miejscu.

Funkcja przerwania „obsługuje” kilka układów licznikowych zatem można „ograniczyć jej działanie do reakcji na zdarzenia pochodzące jedynie ze skonfigurowanego układu, którym jest w przykładzie timer **TIM16**. Zatem pierwsza testowa wersja funkcji może mieć postać

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
	if(htim->Instance==TIM16)
	{
		HAL_GPIO_TogglePin(GPIOA,GreenLED_Pin);
	}
}
```

Pamiętać należy jeszcze aby w funkcji **main** uruchomić działanie timera:

```
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start_IT(&htim16);
  /* USER CODE END 2 */
```

Teraz można zaprogramować wysyłanie komunikatów do komputera PC. Komunikaty wysyłane mają być cyklicznie z informacją o numerze transmisji. Zatem kod obsługujący przerwanie timerowe może mieć poniższą postać:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
	if(htim->Instance==TIM16)
	{
		/* licznik wiadomości, static aby zmienna pamiętała aktulaną wartość
		pomiędzy wywołaniami */
		static uint16_t licznik = 0;
		/*tablica na komunikaty, rozmiar z zapasem tak aby pomieścić każdy
		komunikat*/
		uint8_t komunikat[50];
		/*rozmiar bieżącego komunikatu*/
		uint16_t rozmiar = 0;

		licznik++;
		/*Formatowanie komunikatu wyjściowego*/
		rozmiar = sprintf((char *)komunikat, "Komunikat numer: %d.\n\r",
		licznik);
		/*transmisja danych UART
		HAL_UART_Transmit(huart, pData, Size, Timeout);*/
		HAL_UART_Transmit(&huart1, (uint8_t *)komunikat, rozmiar,1000);
		HAL_GPIO_TogglePin(GPIOA,GreenLED_Pin);
	}
}
```

Szczegółowy opis funkcji znajduje się na końcu instrukcji.

Działanie programu można sprawdzić uruchamiając na komputerze własną aplikację napisaną w **LabVIEW** albo program **RealTerm** (uruchomiony jako

administrator). Należy sprawdzić w Menedżerze urządzeń numer portu komunikacyjnego przypisanego do konwertera USB->UART i w programie **RealTerm** należy w zakładce port ustawić parametry transmisji takie jak w środowisku **STM32CubeIDE** i zatwierdzić przyciskiem **Change**.

Odbieranie komunikatów PC->STM:

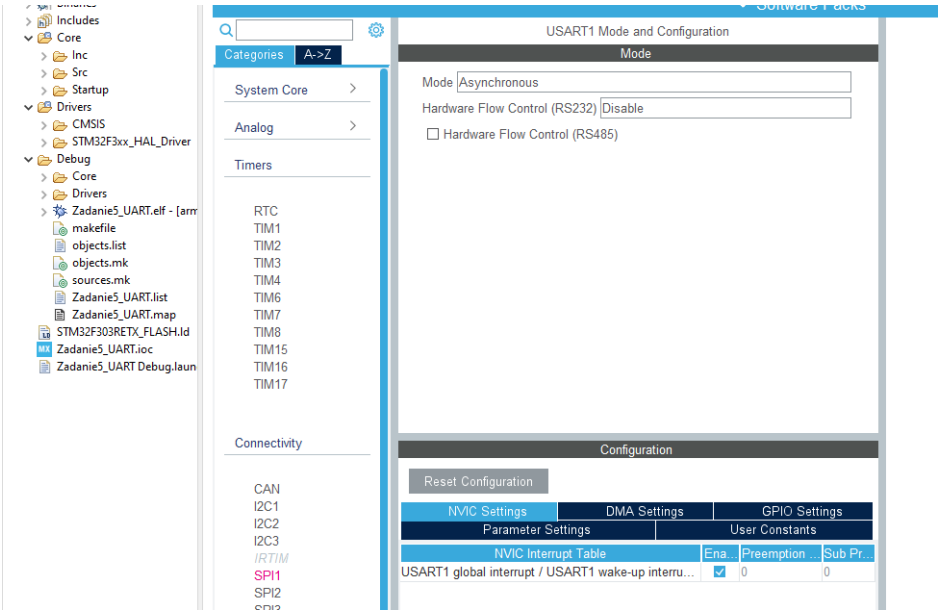
Na początek należy zdefiniować zmienną, w której przechowywane będą „odebrane znaki”

```
/* USER CODE BEGIN PV */
uint8_t odebrana;
/* USER CODE END PV */
```

Należy również uruchomić mechanizm obioru danych w przerwaniach

```
/* USER CODE BEGIN 2 */
 HAL_UART_Receive_IT(&huart1, &odebrana, 1);
/* USER CODE END 2 */
```

Jeżeli odbieranie danych ma odbywać się w przerwaniach należy stosowne przerwanie aktywować – rysunek 5. W tym celu należy przejść do **Pinout&Configuration-> Connectivity** dla aktywnego **USART1** w zakładce **NVIC Settings** i uruchomić mechanizm przerwań.



Rysunek 5. Aktywacja przerwania USART1.

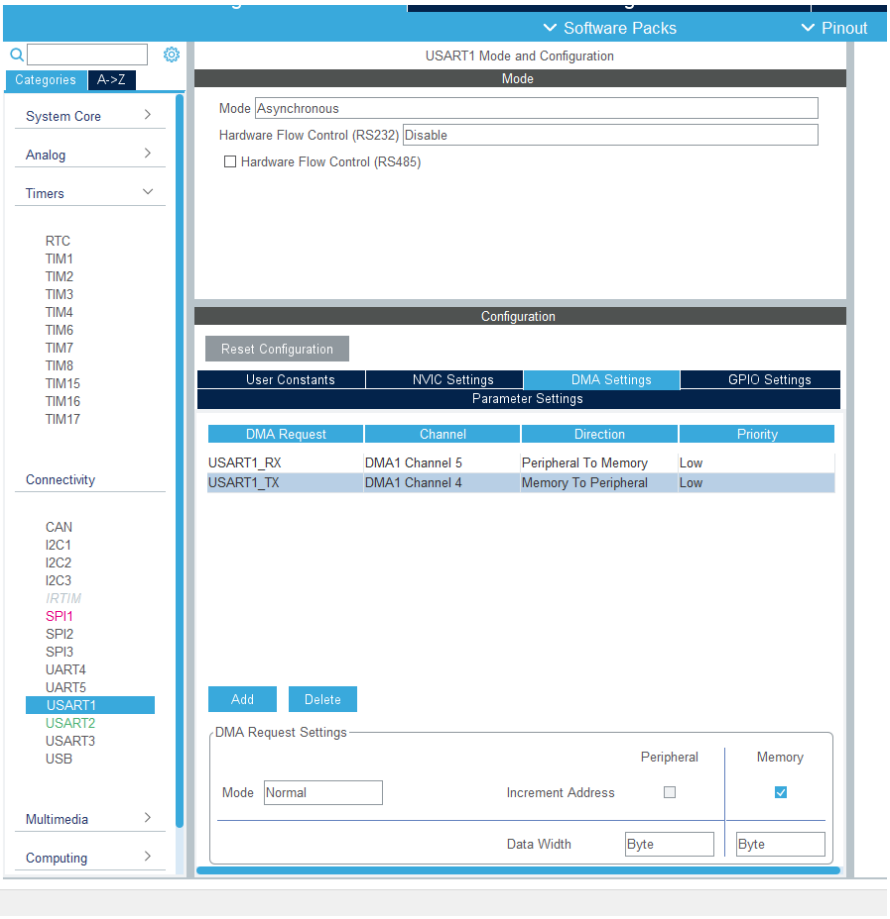
Odbieranie w przerwaniach odbywać się będzie za pomocą dedykowanej funkcji `HAL_UART_RxCpltCallback`, którą podobnie jak poprzednio można przenieść do funkcji pliku main.c (oryginalnie w pliku stm32f3xx_hal_uart.c). Struktura funkcji może być taka, że jeśli odebrany został znak ’1’ STM32 odpowiada komputerowi PC np.: ciągiem „SET”, jeśli zaś odebrany został znak ’0’ odpowiedź brzmi „RESET” i jest każdorazowo sygnalizowana (potwierdzana) stanem diody LED na płytce. Odebrany inny znak również powinien zostać potwierdzony stosowną odpowiedzią.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
	/*tablica na komunikaty, rozmiar z zapasem tak aby pomieścić każdy
	komunikat*/
	uint8_t komunikatNAD[50];
	/*rozmiar bieżącego komunikatu */
	uint16_t rozmiarNAD = 0;

	if(huart->Instance==USART1)
	{
		switch (odebrana)
		{

		/*Odebrany komunikat to 0*/
		case '0':
			/*Formatowanie komunikatu wyjściowego*/
			rozmiarNAD = sprintf ...
			HAL_GPIO_WritePin
			break;

		/*Odebrany komunikat to 0*/
		case '1':
			/*Formatowanie komunikatu wyjściowego*/
			rozmiarNAD = sprintf ...
			HAL_GPIO_WritePin
			break;
			/*Odebrany inny komunikat*/
		default:
			/*Formatowanie komunikatu wyjściowego*/
			rozmiarNAD = sprintf ...
			break;
		}

		HAL_UART_Transmit
		HAL_UART_Receive_IT
		}
}
```

W programie **RealTerm** należy w zakładce **Send** można wysyłać komendy.

Transmisja DMA:

Przerwania pozostają włączone. Nalezy przejść do **Pinout&Configuration-> Connectivity** dla aktywnego **USART1** w zakładce **DMA Settings** i dodać następujące żądania dla DMA – rysunek 6.



Rysunek 6. Konfiguracja DMA

W kodzie należy uruchomić mechanizm DMA:

```
/* USER CODE BEGIN 2 */
 HAL_UART_Receive_DMA(&huart1, &odebrana, 1);
/* USER CODE END 2 */
```

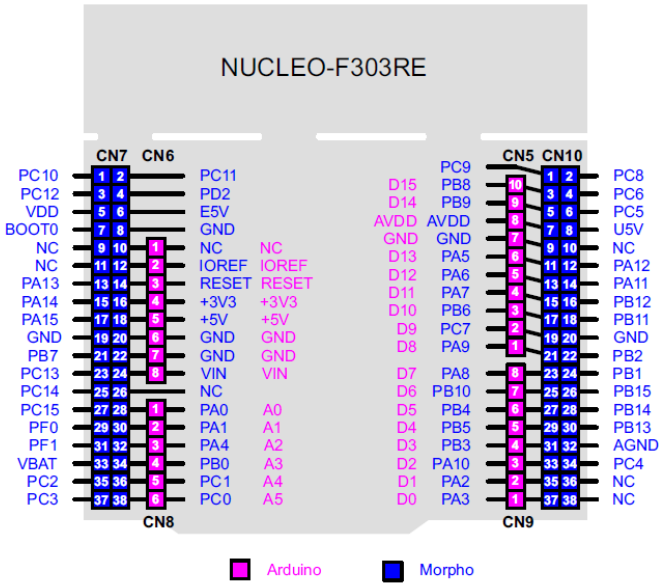
Zaś funkcja **HAL_UART_RxCpltCallback** powinna może mieć poniższą postać:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
	/*tablica na komunikaty, rozmiar z zapasem tak aby pomieścić każdy
	komunikat*/
	uint8_t komunikatNAD[50];
	/*rozmiar bieżącego komunikatu */
	uint16_t rozmiarNAD = 0;

	if(huart->Instance==USART1)
	{
		rozmiarNAD = sprintf((char *)komunikatNAD, "Odebrany komunikat:
		%c\n\r", odebrana);
		HAL_UART_Transmit_DMA(&huart1, komunikatNAD, rozmiarNAD);
		HAL_UART_Receive_DMA(&huart1, &odebrana, 1);
		HAL_GPIO_TogglePin(GPIOA,GreenLED_Pin);
	}
}
```

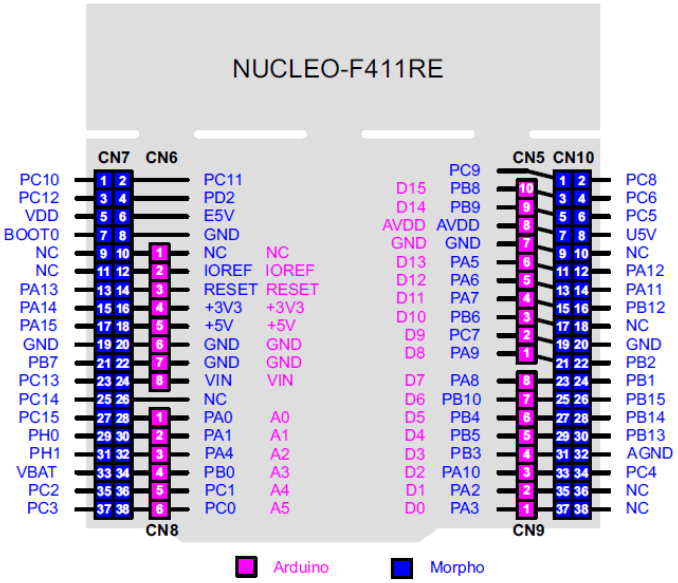
Powyższy schemat można wykorzystać również jak poprzednio do przesyłania komend.

Komunikaty wieloznakowe:

Na podstawie przykładów komunikacji dla jednego znaku oraz parametrów funkcji **`HAL_UART_Receive_IT`** i **`HAL_UART_Transmit`** dotyczących rozmiaru nadawanych/odbieranych danych proszę zaproponować mechanizm obsługujący tego typu transmisje. Można wykorzystać funkcje opisane na końcu instrukcji



Rysunek 7. Wyprowadzenia na płytce NUCLEO-F303RE. Źródło: STM32 Nucleo-64 boards (MB1136.pdf).



Rysunek 8. Wyprowadzenia na płytce NUCLEO-F411RE. Źródło: STM32 Nucleo-64 boards (MB1136.pdf).

Rysunki 7 i 8 przedstawiają wyprowadzenia dla płytek NUCLEO dla dwóch mikrokontrolerów STM23F411RE i STM23F4303RE. Dla wymienionych mikrokontrolerów przykładowe dostępne linie **USART1** to: PC4(Tx) i PC5(Rx) (STM32F303) oraz PA9(Tx) i PA10(Rx) (STM32F411).

Funkcje wykorzystane w zadaniu:

```
/**
  * @brief  Starts the TIM Base generation in interrupt mode.
  * @param  htim TIM Base handle
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)
{
  uint32_t tmpsmcr;

  /* Check the parameters */
  assert_param(IS_TIM_INSTANCE(htim->Instance));

  /* Check the TIM state */
  if (htim->State != HAL_TIM_STATE_READY)
  {
    return HAL_ERROR;
  }

  /* Set the TIM state */
  htim->State = HAL_TIM_STATE_BUSY;

  /* Enable the TIM Update interrupt */
  __HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE);
```

```
  /* Enable the Peripheral, except in trigger mode where enable is automatically
done with trigger */
  if (IS_TIM_SLAVE_INSTANCE(htim->Instance))
  {
    tmpsmcr = htim->Instance->SMCR & TIM_SMCR_SMS;
    if (!IS_TIM_SLAVEMODE_TRIGGER_ENABLED(tmpsmcr))
    {
      __HAL_TIM_ENABLE(htim);
    }
  }
  else
  {
    __HAL_TIM_ENABLE(htim);
  }

  /* Return function status */
  return HAL_OK;
}
/**
  * @brief Send an amount of data in blocking mode.
  * @note   When UART parity is not enabled (PCE = 0), and Word Length is
configured to 9 bits (M1-M0 = 01),
  *         the sent data is handled as a set of u16. In this case, Size must
indicate the number
  *         of u16 provided through pData.
  * @param huart   UART handle.
  * @param pData   Pointer to data buffer (u8 or u16 data elements).
  * @param Size    Amount of data elements (u8 or u16) to be sent.
  * @param Timeout Timeout duration.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData,
uint16_t Size, uint32_t Timeout)
{
  uint8_t  *pdata8bits;
  uint16_t *pdata16bits;
  uint32_t tickstart;

  /* Check that a Tx process is not already ongoing */
  if (huart->gState == HAL_UART_STATE_READY)
  {
    if ((pData == NULL) || (Size == 0U))
    {
      return  HAL_ERROR;
    }

    __HAL_LOCK(huart);

    huart->ErrorCode = HAL_UART_ERROR_NONE;
    huart->gState = HAL_UART_STATE_BUSY_TX;

    /* Init tickstart for timeout management */
    tickstart = HAL_GetTick();

    huart->TxXferSize  = Size;
    huart->TxXferCount = Size;
```

```
    /* In case of 9bits/No Parity transfer, pData needs to be handled as a
uint16_t pointer */
    if ((huart->Init.WordLength == UART_WORDLENGTH_9B) && (huart->Init.Parity ==
UART_PARITY_NONE))
    {
      pdata8bits  = NULL;
      pdata16bits = (uint16_t *) pData;
    }
    else
    {
      pdata8bits  = pData;
      pdata16bits = NULL;
    }

    __HAL_UNLOCK(huart);

    while (huart->TxXferCount > 0U)
    {
      if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_TXE, RESET, tickstart,
Timeout) != HAL_OK)
      {
        return HAL_TIMEOUT;
      }
      if (pdata8bits == NULL)
      {
        huart->Instance->TDR = (uint16_t)(*pdata16bits & 0x01FFU);
        pdata16bits++;
      }
      else
      {
        huart->Instance->TDR = (uint8_t)(*pdata8bits & 0xFFU);
        pdata8bits++;
      }
      huart->TxXferCount--;
    }

    if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_TC, RESET, tickstart,
Timeout) != HAL_OK)
    {
      return HAL_TIMEOUT;
    }

    /* At end of Tx process, restore huart->gState to Ready */
    huart->gState = HAL_UART_STATE_READY;

    return HAL_OK;
  }
  else
  {
    return HAL_BUSY;
  }
}
/**
  * @brief Receive an amount of data in interrupt mode.
  * @note   When UART parity is not enabled (PCE = 0), and Word Length is
configured to 9 bits (M1-M0 = 01),
```

```
  *         the received data is handled as a set of u16. In this case, Size must
indicate the number
  *         of u16 available through pData.
  * @param huart UART handle.
  * @param pData Pointer to data buffer (u8 or u16 data elements).
  * @param Size  Amount of data elements (u8 or u16) to be received.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData,
uint16_t Size)
{
  /* Check that a Rx process is not already ongoing */
  if (huart->RxState == HAL_UART_STATE_READY)
  {
    if ((pData == NULL) || (Size == 0U))
    {
      return HAL_ERROR;
    }

    __HAL_LOCK(huart);

    /* Set Reception type to Standard reception */
    huart->ReceptionType = HAL_UART_RECEPTION_STANDARD;

    /* Check that USART RTOEN bit is set */
    if (READ_BIT(huart->Instance->CR2, USART_CR2_RTOEN) != 0U)
    {
      /* Enable the UART Receiver Timeout Interrupt */
      ATOMIC_SET_BIT(huart->Instance->CR1, USART_CR1_RTOIE);
    }

    return (UART_Start_Receive_IT(huart, pData, Size));
  }
  else
  {
    return HAL_BUSY;
  }
/**
  * @brief  Toggle the specified GPIO pin.
  * @param  GPIOx where x can be (A..F) to select the GPIO peripheral for STM32F3
family
  * @param  GPIO_Pin specifies the pin to be toggled.
  * @retval None
  */
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  uint32_t odr;

  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  /* get current Ouput Data Register value */
  odr = GPIOx->ODR;

  /* Set selected pins that were at low level, and reset ones that were high */
  GPIOx->BSRR = ((odr & GPIO_Pin) << GPIO_NUMBER) | (~odr & GPIO_Pin);
}
```

```
/**
  * @brief  Set or clear the selected data port bit.
  *
  * @note   This function uses GPIOx_BSRR and GPIOx_BRR registers to allow atomic
read/modify
  *         accesses. In this way, there is no risk of an IRQ occurring between
  *         the read and the modify access.
  *
  * @param  GPIOx where x can be (A..F) to select the GPIO peripheral for STM32F3
family
  * @param  GPIO_Pin specifies the port bit to be written.
  *         This parameter can be one of GPIO_PIN_x where x can be (0..15).
  * @param  PinState specifies the value to be written to the selected bit.
  *         This parameter can be one of the GPIO_PinState enum values:
  *           @arg GPIO_PIN_RESET: to clear the port pin
  *           @arg GPIO_PIN_SET: to set the port pin
  * @retval None
  */
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState
PinState)
{
  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  assert_param(IS_GPIO_PIN_ACTION(PinState));

  if(PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = (uint32_t)GPIO_Pin;
  }
  else
  {
    GPIOx->BRR = (uint32_t)GPIO_Pin;
  }
}
```

HAL_UARTEx_ReceiveToIdle_DMA(huart, pData, Size)

/* Funkcja służy do odbioru danych przy pomocy DMA, aż do chwili przejścia w stan bezczynny IDLE lub odebrania zadanej liczby bajtów danych w parametrze Size. Do użycia tej funkcji muszą być ustawione przerwania i DMA w UART */

Na przykład

```
HAL_UARTEx_ReceiveToIdle_DMA(&huart1, odebrany_bufor_bajtow, 100);
```

/* W tym przypadku funkcja będzie odbierać dane aż do chwili przejścia w stan bezczynności - IDLE albo do momentu odebrania 100 bajtów danych (po odebraniu 100 danych wywołane zostanie przerwanie). Obsługa przerwania realizowana jest za pomocą poniższej funkcji: */

```
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size)

{
```

```
}

/* Size - liczba bajtów do odebrania. Czyli jeśli odbieranie będzie dotyczyć ciągu
"1234" to Size należy ustawić na 4.

UWAGA: domyślnie ustawienie powoduje wywołanie przerwania również w połowie bufora
odbieranych danych (w podanym przykładzie po 4 bajtach, a poprzednio po 50). W
celu dezaktywacji wywołania przerwania w "półbuforze" należy użyć poniższego
wywołania: */

__HAL_DMA_DISABLE_IT(&hdma_usart1_rx, DMA_IT_HT);

/*Tak więc ostateczna postać instrukcji dla naszej funkcji jest następująca: */

HAL_UARTEx_ReceiveToIdle_DMA(&huart1, oodebrany_bufor_bajtow, 100);

__HAL_DMA_DISABLE_IT(&hdma_usart1_rx, DMA_IT_HT);
```