KURS STM32

Wojciech Olech

CZĘŚĆ IV: PRZERWANIA

TEORIA

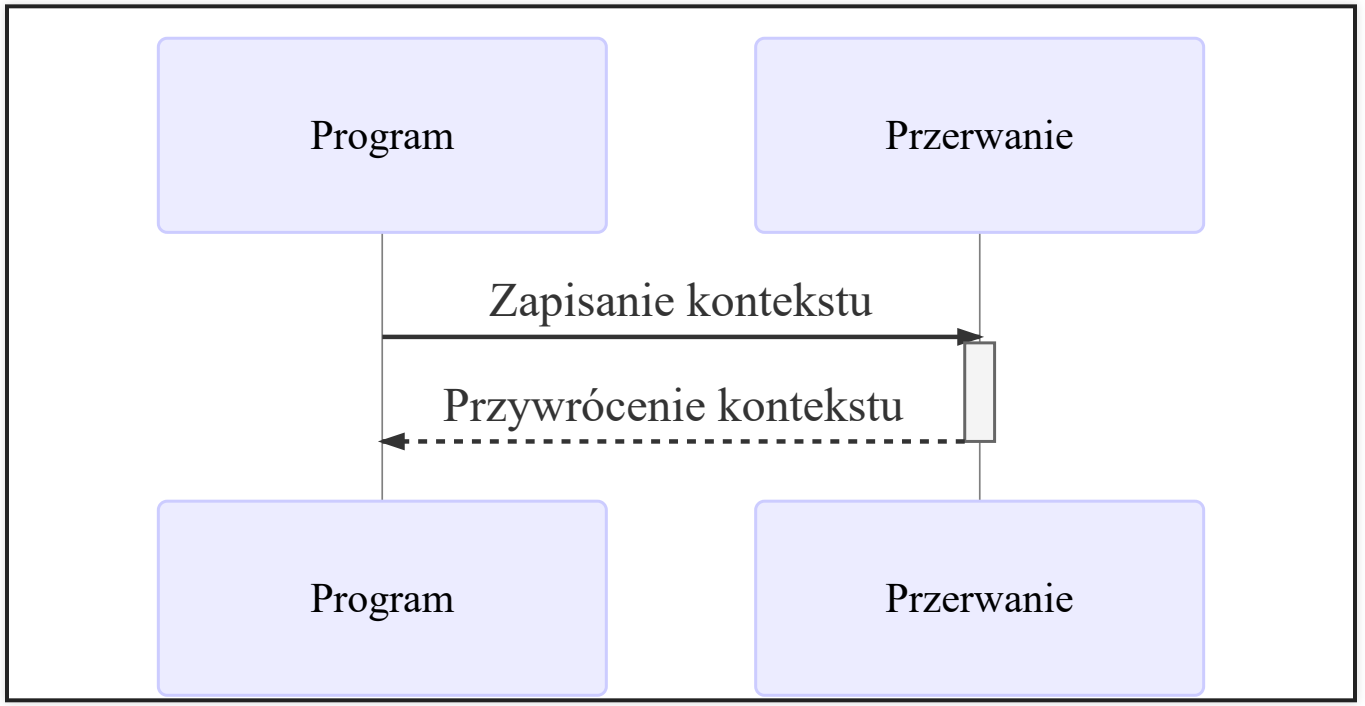
Przerwania są mechanizmem który pozwala na asynchroniczną obsługę zdarzeń mikrokontrolera. Obsługiwane są przez NVIC (Nested Vector Interrupt Controller) który pozwala na zaawansowaną konfigurację między innymi priorytetów przerwań.

W mikrokontrolerach STM32 mamy dostępne do 240 źródeł przerwań, oraz 16 priorytetów konfigurowalnych per przerwanie.

PROCESY WYKONYWANIA PRZERWAŃ

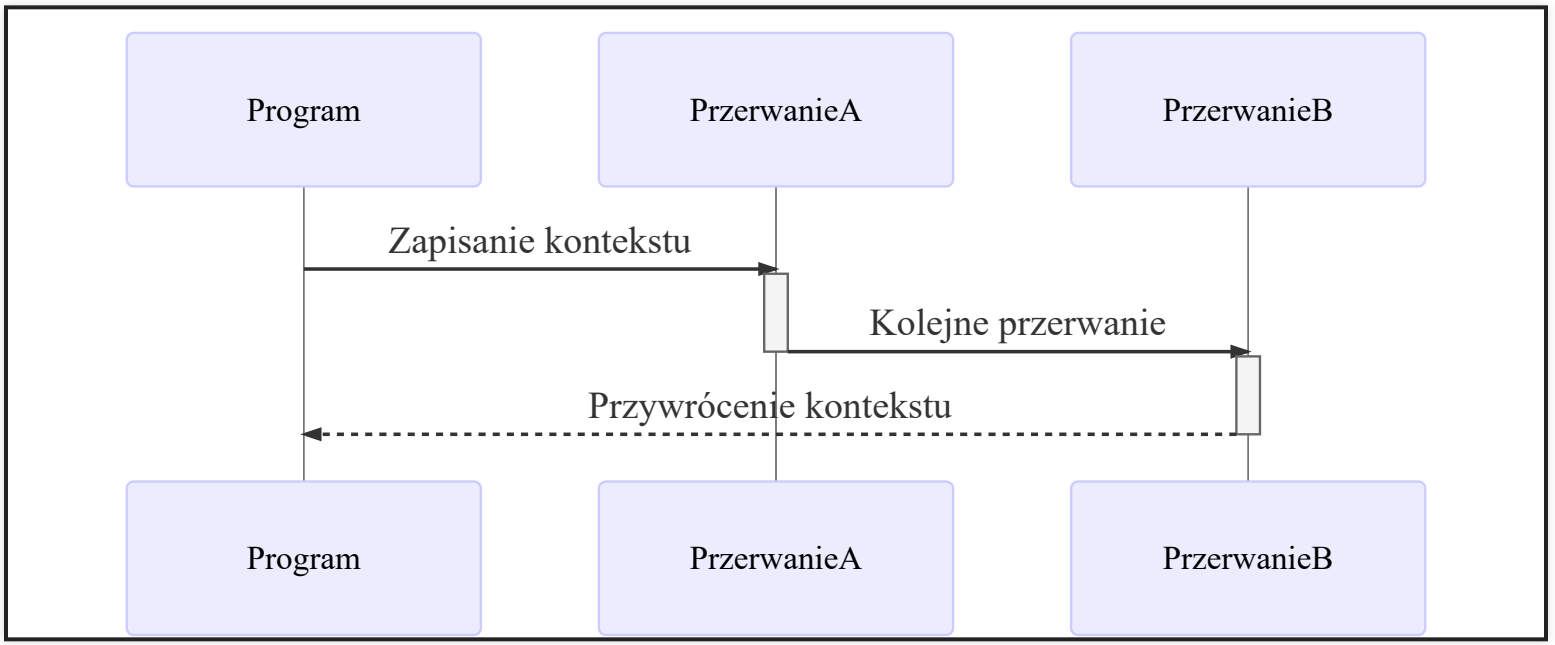
POJEDYNCZE PRZERWANIE

W momencie wywołania przez procesor przerwania, procesor zapisuje swój stan, następnie skacze do funkcji obsługującej dane przerwanie, wykonuje ją, a potem przywraca poprzedni stan i kontynuuje wykonywanie poprzedniego kodu.



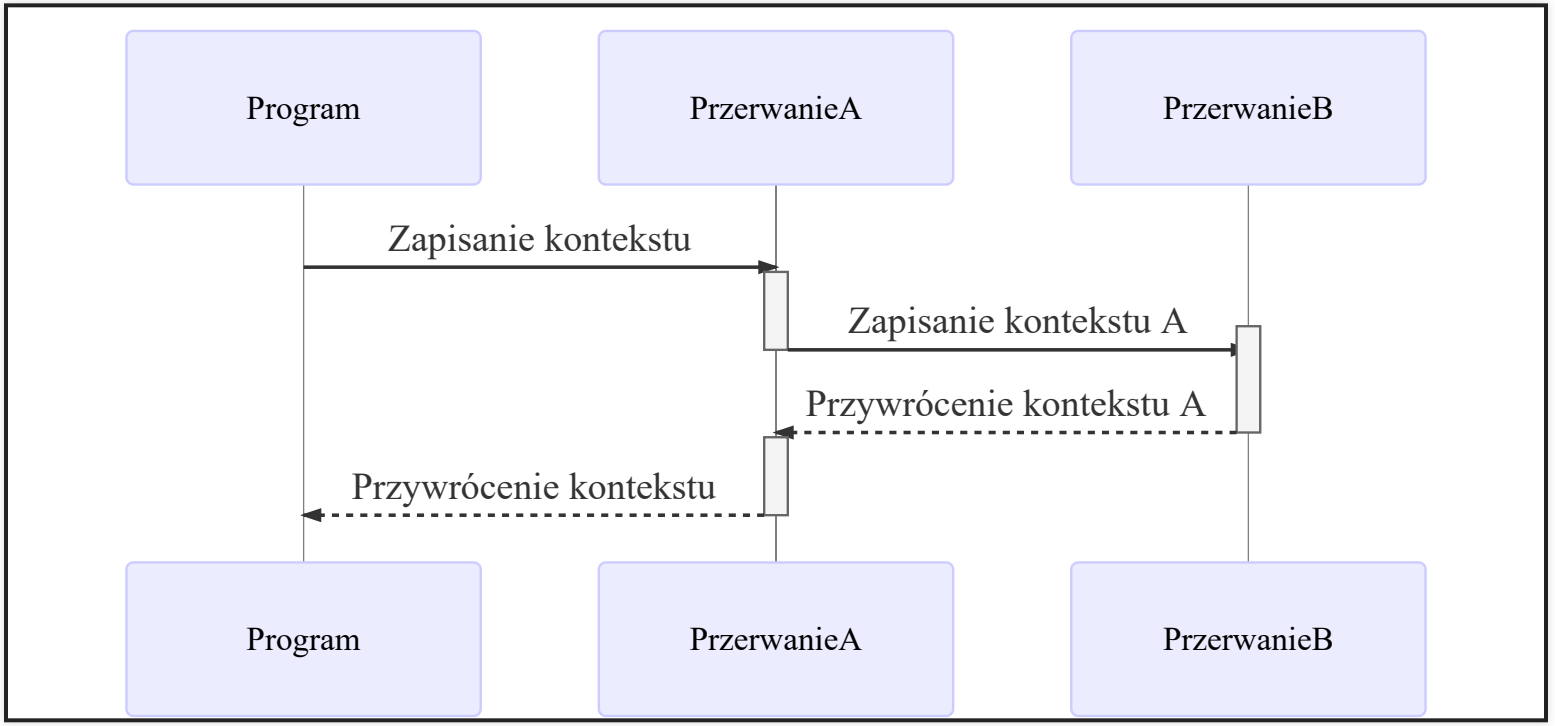
KOLEJKOWANIE (TAIL-CHAINING) PRZERWAŃ

Jeśli podczas wykonywania się funkcji przerwania, zostanie wywołane nowe przerwanie z takim samym lub niższym priorytetem, zostanie ono zakolejkowane przez procesor i wykonane po zakończeniu działania poprzedniej funkcji przerwania. W poniższym diagramie, przerwanie B ma taki sam lub niższy priorytet niż przerwanie A



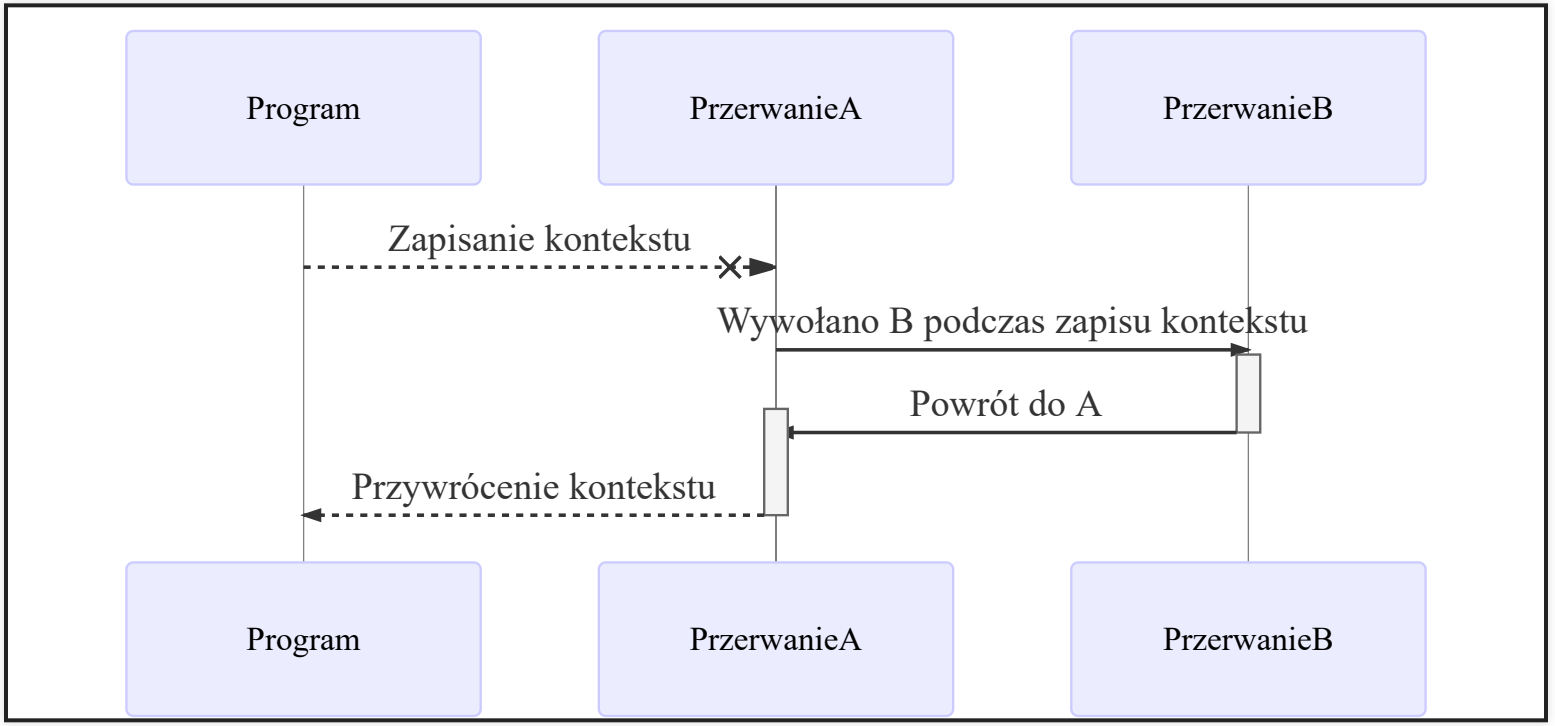
ZAGNIEŻDŻANIE (NESTING) PRZERWAŃ

Jeśli podczas wykonywania się funkcji przerwania, zostanie wywołane nowe przerwanie z wyższym priorytetm, procesor zapisze kontekst programu i skoczy do funkcji tego przerwania.



PRZERWANIA WYWOŁANE PODCZAS ZAPISU KONTEKSTU

Jeśli przerwanie o wyższym priorytecie zostanie wywołane podczas zapisywania się kontekstu programu wywołanego innym przerwaniem, to procesor skoczy od razu do przerwania o wyższym priorytecie



STANY PRZERWAŃ

Każde przerwanie może być skonfigurowane jako włączone albo wyłączone przez program z użyciem funkcji CMSISowej:

- `void NVIC_EnableIRQ(IRQn_t IRQn)`
- `void NVIC_DisableIRQ(IRQn_t IRQn)`

Każde przerwanie posiada flagę, która mówi czy jest oczekujące czy nie. Wyłączenie przerwania powoduje jedynie że procesor nie skacze do funkcji je obsługujacej, ale nadal ustawia flagę oczekiwania, więc po włączeniu go bez czyszczenia jej zostanie wykonany skok do procedury obsługi. Można ją odczytać używając funkcji `uint32_t NVIC_GetPending(IRQn_t IRQn)`, i ustawić funkcjami

- `void NVIC_SetPendingIRQ(IRQn_t IRQn)`
- `void NVIC_ClearPendingIRQ(IRQn_t IRQn)`

PRZERWANIA ZEWNĘTRZNE

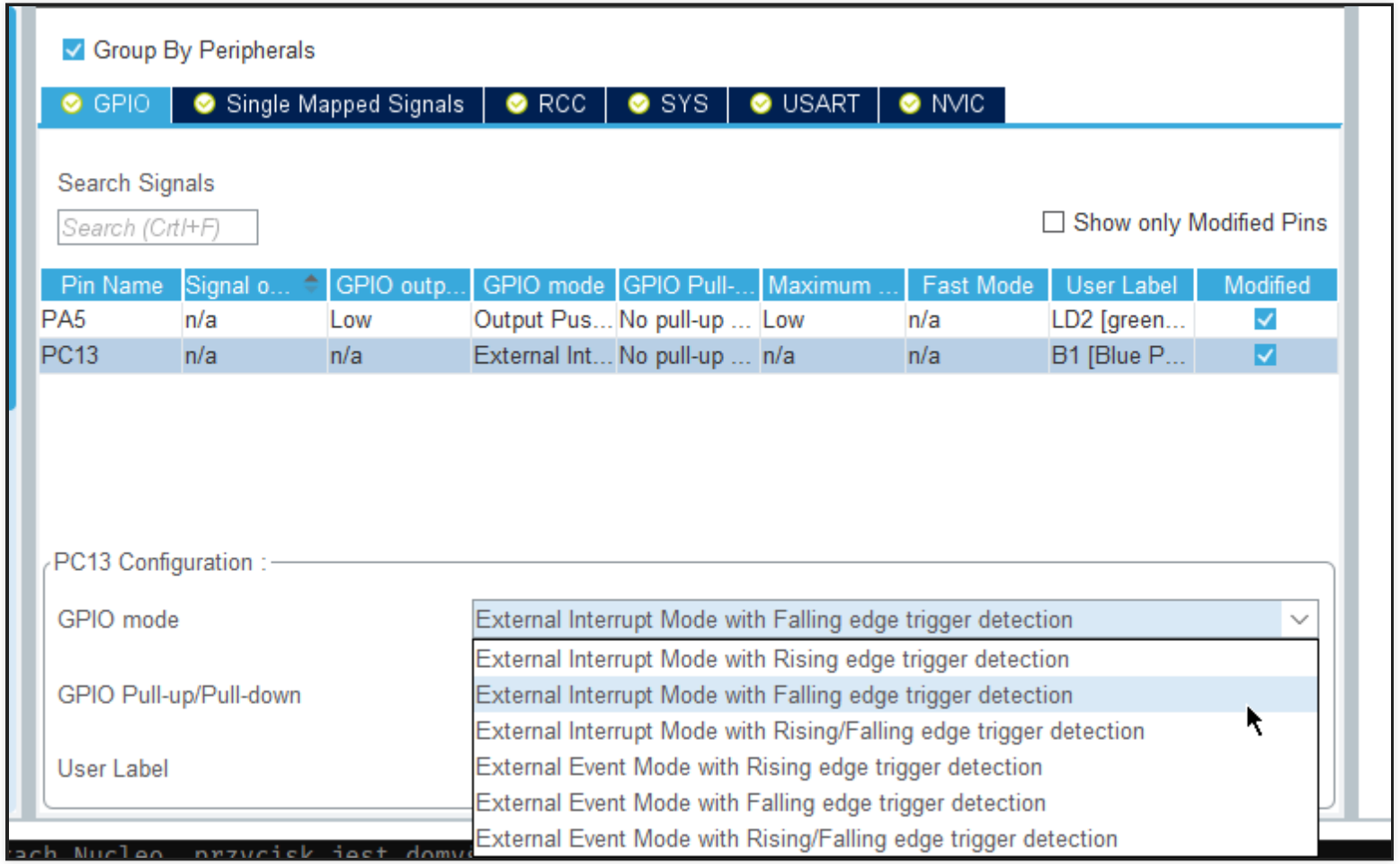
Przerwania są wywoływane zazwyczaj przez peryferia - na przykład magistrale komunikacyjne (SPI, I2C, USART), zegary, lub GPIO.

Jednym z najprostszych rodzajów przerwania jest przerwanie GPIO. Wywołuje się ono w momencie zmiany stanu pinu cyfrowego i jest używane między innymi do obsługi zewnętrznych zdarzeń (na przykład kiedy moduł ma ważną informację do przesłania, może zmienić stan na jednym z pinów co można łatwo przechwycić przerwaniem).

Żeby skonfigurować pin w trybie przerwania, należy na graficznym przedstawieniu procesora wybrać interesujący nas pin i ustawić go w tryb `GPIO_EXTIn`, gdzie `n` to numer przerwania jakim zostanie obsłużony.

Na płytkach Nucleo, przycisk jest domyślnie ustawiony w tryb obsługi za pomocą przerwania (`GPIO_EXTI13`)

Następnie należy wybrać jakiego rodzaju jest przerwanie i kiedy ma być generowane.

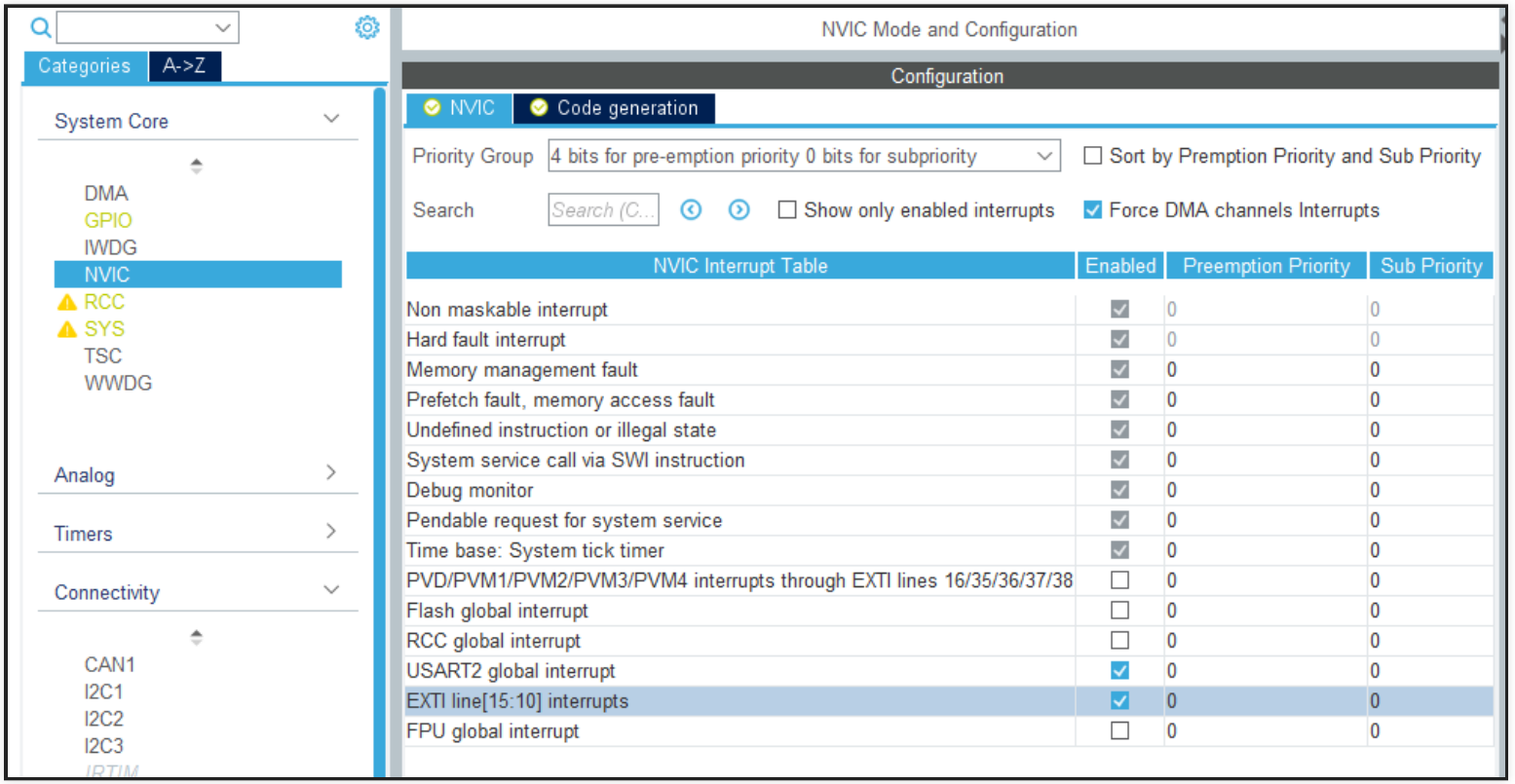


External Interrupt Mode oznacza tryb przerwania, ten tryb nas interesuje jeśli chcemy wywoływać funkcję w momencie zmiany stanu na pinie.
External Event Mode używany jest do wybudzania procesora ze stanu uśpienia w momencie zmiany stanu na przycisku i nie będziemy go teraz używali.

Możemy również wybrać kiedy ma być wywoływane przerwanie:

- *Rising edge trigger* - przerwanie zostanie wywołane w momencie zmiany stanu z niskiego na wysoki (zbocze wznoszące)
- *Falling edge trigger* - przerwanie zostanie wywołane w momencie zmiany stanu z wysokiego na niski (zbocze opadające)
- *Rising/Falling edge trigger* - przerwanie zostanie wywołane w momencie zmiany stanu na jakikolwiek

Domyślnie przycisk jest skonfigurowany w trybie przerwania, ale kod obsługi przerwania może nie być generowany. Żeby wygenerować kod obsługi przerwania, należy zaznaczyć tą opcję w ustawieniach NVIC (kontrolera przerwań).



Po wygenerowaniu kodu projektu, w pliku `stm32_**xx_it.c` powinna pojawić się taka funkcja:

```
/**
  * @brief This function handles EXTI line[15:10] interrupts.
  */
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

Ta funkcja wywołuje się kiedy przycisk zostaje wciśnięty. Nie będziemy jej jednak modyfikować, ponieważ wywołuje ona handler który obsługuje w pełni przerwanie (czyści jego flagi) i wywołuje finalnie *słabą* (oznaczoną modyfikatorem `weak`) funkcję `HAL_GPIO_EXTI_Callback(GPIO_Pin)`, którą nadpiszemy w naszym kodzie.

W pliku `main.c` stworzymy funkcję o tej nazwie która sprawdzi czy przerwanie przyszło od przycisku, a jeśli tak to zamiga diodą. Powinna się ona wywołać po każdym wciśnięciu przycisku.

```
void HAL_GPIO_EXTI_Callback(uint16_t pin) {
    if (pin == B1_Pin) {
        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    }
}
```

W ten sposób stworzyliśmy asynchroniczną obsługę przycisku. Niezależnie od tego co procesor robi w danym momencie, kiedy wciśniemy przycisk skoczy on do funkcji obsługi przerwania, która wywoła naszą funkcję i zmieni stan diody.

UŻYWANIE PRZERWAŃ NA PRZYKŁADZIE UARTA

Jednym z popularnych zastosowań przerwań jest używanie ich do asynchronicznej (nieblokującej) obsługi seriala. Tutaj użyjemy ich żeby odczytywać z UARTa tekst do napotkania znaku nowej linii.

W tym celu weźmiemy kod z zadania 4 z poprzedniego rozdziału i zmodyfikujemy go w taki sposób, żeby odbierał wiadomość asynchronicznie

DOSTĘPNE PRZERWANIA UARTA

UART posiada kilka przerwań które wywołują się w określonych warunkach. Dwa podstawowe przerwania jakie będą nas interesować w większości przypadków, to

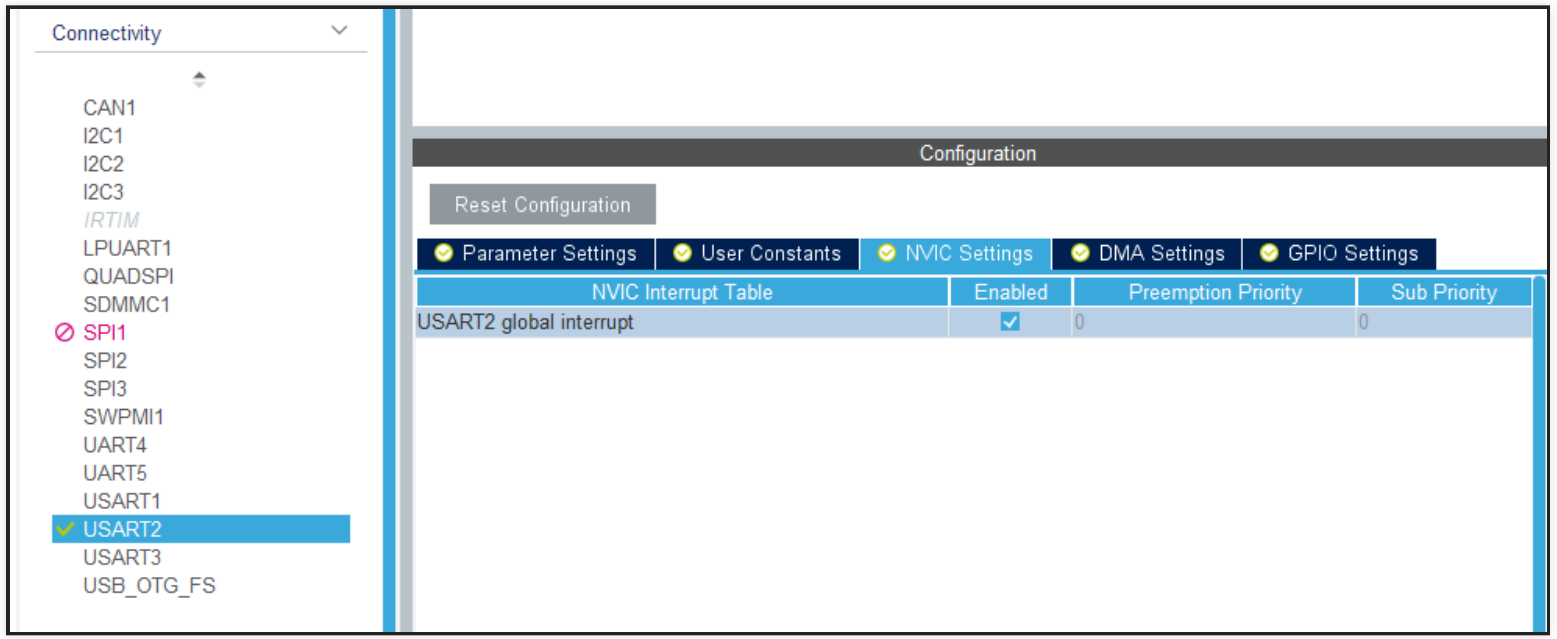
- `HAL_UART_TxCpltCallback` - przerwanie wywołane w momencie zakończenia odbioru danych
- `HAL_UART_RxCpltCallback` - przerwanie wywołane w momencie zakończenia wysyłania danych

Oprócz tego, mamy również dostępne przerwania które wywołają się w połowie odczytu danych (`HAL_UART_(Tx/Rx)HalfCpltCallback`), błędu transmisji (`HAL_UART_ErrorCallback`) i zatrzymania transmisji (`HAL_UART_AbortCpltCallback` i `HAL_UART_Abort(Transmit/Receive)CpltCallback`)

Wszystkie te przerwania są *słabymi* funkcjami, co znaczy że żeby je nadpisać wystarczy stworzyć funkcję o takiej samej sygnaturze gdzieś w naszym projekcie i wtedy zostanie ona wywołana automatycznie w momencie wywołania przerwania.

Za wstępną obsługę przerwań seriala odpowiada funkcja `HAL_UART_IRQHandler` która wywołuje się automatycznie w momencie wywołania przerwania UARTa. Odczytuje ona stan rejestrów UARTa i na ich podstawie, oraz danych w strukturze odpowiadającej za UARTa, automatycznie obsługuje przerwanie i wywołuje odpowiednie handlery jeśli to konieczne.

Żeby przerwania USARTa działały, należy aktywować jego globalne przerwanie w Device Managerze. Jeśli tego nie zrobimy, przerwanie nie zostanie nigdy obsłużone bo nie zostanie wygenerowany kod jego obsługi (znajdujący się w pliku `stm32**xx_it.c`). Należy upewnić się też że przerwanie jest włączone w zakładce NVIC.



ZADANIE: ZMIENIĆ KOD ODCZYTYWANIA DANYCH Z ZADANIA 4 Z POPRZEDNIEGO ROZDZIAŁU NA ASYNCHRONICZNY

Nasz kod pętli głównej powinien wyglądać mniej-więcej w ten sposób:

```
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
    if (readSerialUntil(&huart2, serial_buffer, 128, '\r', HAL_MAX_DELAY)) {
        printf("[%lu]: %s\n", HAL_GetTick(), serial_buffer);
        memset(serial_buffer, 0, 128);
    }
}
/* USER CODE END 3 */
```

Jednak to podejście blokuje cały program do momentu zakończenia odczytu, czego chcemy uniknąć. Musimy w takim razie zmienić funkcję `readSerialUntil` na inną, która zamiast czekać na odczytanie, po prostu rozpocznie odczyt. Następnie procesor może sprawdzać w przerwaniu UARTa (które będzie wywoływać się automatycznie) jakie znaki odczytał, a następnie w momencie znalezienia znaku nowej linii przerwać odczyt.

W międzyczasie nasz program może robić cokolwiek innego, oraz sprawdzać czy transmisja się zakończyła.

Tak samo jak przy pollingowym odczytywaniu, musimy podać ilość znaków do odczytania i przerwanie końca transmisji wykona się dopiero w momencie odczytania podanej ilości bajtów. Tak więc, musimy znowu odczytywać znak po znaku.

Zacznijmy od napisania obsługi naszych przerwań. Nasz program ma czytać znak po znaku z seriala, do momentu napotkania znaku nowej linii. Jeśli go napotka, powinien powiadomić jakoś o tym resztę programu i zakończyć odczyt. Do powiadamiania możemy użyć globalnej flagi typu `bool` (z nagłówka `stdbool.h`).

Jeśli poprosimy serial o odczytanie jednego znaku używając przerwania, to po odczytaniu go wykona się przerwanie `HAL_UART_RxCpltCallback`. W tym przerwaniu musimy zawrzeć nasz kod.

Musimy też trzymać gdzieś ilość odczytanych znaków żeby wiedzieć gdzie mają trafiać kolejne znaki w buforze. Stwórzmy sobie strukturę w której zawrzemy wszystkie informacje:

```
typedef struct {
    size_t max_size;
    size_t char_count;
    char* buffer;
    bool transmission_finished;
} SerialRxBuffer;
```

Stwórzmy globalną instancję tej struktury, ponieważ nie mamy możliwości przekazywania argumentów do funkcji przerwań. Możemy teraz przystąpić do pisania funkcji przerwania.

```
SerialRxBuffer serial_buffer;

// [...]

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    // Sprawdź czy nie przekraczasz bufora lub czy nie spotkales znaku konca linii
    if (serial_buffer.char_count == serial_buffer.max_size - 1 || serial_buffer.buffer[serial_buffer.char_count] == '\r') {
        // Bufor pelny lub odebrano koniec linii - dodaj znak konca stringa i zakoncz odbior
        serial_buffer.buffer[serial_buffer.char_count++] = '\0';
        serial_buffer.transmission_finished = true;
    } else {
        // Kontynuuj transmisje
        serial_buffer.char_count++;
        HAL_UART_Receive_IT(huart, serial_buffer.buffer + serial_buffer.char_count, 1);
    }
}
```

Mamy już gotową obsługę przerwania, teraz należy w funkcji `main` inicjalizować naszą strukturę z buforem i wywołać funkcję `HAL_UART_Receive_IT` żeby zacząć odbiór.

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
serial_buffer.buffer = data_buffer;
serial_buffer.char_count = 0;
serial_buffer.max_size = 128;
serial_buffer.transmission_finished = false;
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
    HAL_UART_Receive_IT(&huart2, serial_buffer.buffer, 1);
    if (serial_buffer.transmission_finished) {
        printf("[%lu]: %s\n", HAL_GetTick(), serial_buffer.buffer);
        serial_buffer.transmission_finished = false;
        serial_buffer.char_count = 0;
    }

    HAL_Delay(100);
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
}
/* USER CODE END 3 */
```

Jak widzimy, powyższy kod miga diodą i jednocześnie cały czas odbiera dane, wysyłając je do komputera po odebraniu pełnej linii. Funkcja która rozpoczyna transmisję nie blokuje nam programu, wszystko dzieje się w tle.

Należy jednak pamiętać o kilku rzeczach dotyczących obsługi przerwań i magistrali komunikacyjnych (nie tylko seriala) z ich użyciem:

1. Funkcje przerwań powinny być jak najmniejsze, żeby nie blokować programu
2. Istnieją lepsze metody odczytu z seriala - można całą procedurę napisać lepiej, bez funkcji HALowych, ale jest to rzadko wykorzystywane (patrz punkt 3).
3. Najlepszy sposób obsługi magistrali komunikacyjnych to zazwyczaj obsługa poprzez DMA, ponieważ odciąża procesor i działa nawet po zatrzymaniu programu przez debuger. Dodatkowo, w przypadku seriala można użyć przerwania *idle line*.

RĘCZNA OBSŁUGA PRZERWAŃ

HAL czasami nie obsługuje automatycznie wszystkich przerwań które są dostępne dla danego peryferium, ale zazwyczaj zostawia niskopoziomowe funkcje/makra które ułatwiają ich używanie. Jednym z takich przerwań jest serialowe przerwanie *idle line*, które uruchamia się w momencie w którym na magistrali USART nie zostało nic odebrane przez okres 1 ramki. Można to przerwanie wykorzystać do wydajnego odbioru komunikatów o zmiennej długości.

Makra do obsługi przerwań znajdują się w pliku nagłówkowym HALowego drivera danego peryferium. W naszym przypadku, dla przerwania *idle line*, będziemy potrzebować kilku makr:

- **__HAL_UART_ENABLE_IT** - to makro służy do włączania przerwań poprzez ustawienie odpowiedniego bitu w rejestrze sterującym przerwaniami USARTa
- **__HAL_UART_GET_FLAG** - USART posiada tylko jedno, globalne przerwanie które wywołuje się dla każdego rodzaju przerwania. Żeby dowiedzieć się które przerwanie konkretnie się wydarzyło, musimy sprawdzić stan jego flagi w rejestrach USARTa.
- **__HAL_UART_CLEAR_IDLEFLAG** - to makro czyści bit flagi przerwania *idle line*, co musimy wykonać po każdym obsłużeniu go. HAL robi to automatycznie dla przerwań obsługiwanych przez jego procedury. Jeśli o tym zapomnimy, przerwanie nie wywoła się ponownie do momentu wyczyszczenia tej flagi.
- **UART_FLAG_IDLE** i **UART_IT_IDLE** to makra które identyfikują flagę przerwania *idle line* oraz samo przerwanie w rejestrach USARTa i użyjemy ich razem z wyżej wymienionymi makrami.

Na początek, należy włączyć przerwanie *idle line*. Najlepiej jest zrobić to w funkcji `main` tuż po inicjalizacji USARTa.

```
__HAL_UART_ENABLE_IT(&huart2, UART_IT_IDLE);
```

Następnie, musimy zmodyfikować handler przerwań USARTa w celu obsługi przerwania idle line, ponieważ `HAL_UART_IRQHandler` nie posiada żadnego handlera dla niego.

W sekcji `USER CODE BEGIN USART2_IRQn 1` funkcji `USART2_IRQHandler` dopisujemy

```
if (__HAL_UART_GET_FLAG(&huart2, UART_FLAG_IDLE) == SET) {
    __HAL_UART_CLEAR_IDLEFLAG(&huart2);
    // tutaj możesz obsłużyć swoje dane, my napiszemy odpowiednią funkcję
    handle_uart_idle();
}
```

Przerwanie obsłużone! Zostaje tylko napisać funkcję `handle_uart_idle` w taki sposób, żeby sprawdzała czy komunikat kończy się terminatorem i jeśli tak, to żeby powiadamiała resztę programu o tym że w buforze są dane do obsłużenia.
Mając to przerwanie, możemy kompletnie przerobić obsługę USARTa, ponieważ nie musimy odbierać jednego znaku po drugim - teraz możemy kazać USARTowi odbiór danych o wielkości bufora, zamiast 1 znaku, a w funkcji `handle_uart_idle` możemy sprawdzić czy komunikat doszedł w całości i przerwać odbiór, lub skopiować komunikat do drugiego bufora (tzw. double buffering) i ponowić odbiór.

Jest to jednak poza zakresem tego kursu, ponieważw następnym rozdziale przedstawię podobne rozwiązanie, ale z użyciem DMA, co pozwoli na prawdziwie asynchroniczną obsługę komunikacji z minimalnym udziałem rdzenia Cortex.

DODATKOWE MATERIAŁY

Prezentacja o przerwaniach w mikrokontrolerach ARM:
http://homepage.cem.itesm.mx/carbajal/Microcontrollers/SLIDES/Interrupts.pdf