

INSTYTUT INFORMATYKI
Wydział Informatyki AGH



Ćwiczenie laboratoryjne lub pokaz zdalny

Zastosowanie systemu przerwań i kanałów DMA

Nazwa kodowa: **irq-dma**. Wersja **20231128.0**

Ada Brzoza

ada.brzoza@agh.edu.pl

1 Wstęp

1.1 Cel ćwiczenia

Celem ćwiczenia jest:

- nabycie praktycznych umiejętności posługiwania się systemem przerwań oraz mechanizmami bezpośredniego dostępu do pamięci (DMA) w nowoczesnych mikrokontrolerach,
- weryfikacja wiedzy i ilustracja zagadnień związanych z DMA i systemem przerwań,
- utrwalenie umiejętności związanych z korzystaniem z uprzednio opracowanych rozwiązań.

1.2 Wymagania wstępne

Wymagania wstępne do przeprowadzenia ćwiczenia są następujące:

- umiejętność posługiwania się językiem C,
- rozumienie pojęcia kompilacji skrośnej (*cross-compiling*) i umiejętność posługiwania się narzędziami do tego rodzaju kompilacji,
- umiejętność rozróżnienia typowych interfejsów zewnętrznych komputera PC,
- umiejętność rozróżniania rejestrów wewnętrznych mikroprocesora oraz rejestrów układów peryferyjnych mikrokontrolera lub systemu mikroprocesorowego,
- umiejętność posługiwania się narzędziami i elementami projektu poznanymi przy realizacji poprzednich ćwiczeń.

1.3 Stanowisko testowe i konfiguracja sprzętowa

Ćwiczenie przeprowadzone jest z wykorzystaniem platformy sprzętowej **STM32F429 Nucleo-144** z mikrokontrolerem z rdzeniem ARM Cortex-M4.

1.3.1 Narzędzia programowe

Od strony programowej użyjemy następujących narzędzi:

- zintegrowanego środowiska STM32CubeIDE,
- programu terminalowego, np. **puTTY**.

1.3.2 Konfiguracja sprzętowa

Do realizacji zadania zostanie wykorzystana płytką testowa Nucleo-144 z mikrokontrolerem STM32F429ZI. W celu automatycznego podawania sygnałów mających generować przerwanie wykorzystany został układ czasowo-licznikowy TIM2 skonfigurowany do generowania wolnozmiennego przebiegu PWM w kanale 4, na wyprowadzeniu zewnętrznym PB11 mikrokontrolera. Sygnał prostokątny PWM na PB11 został skierowany do wyprowadzenia PG0 pracującego jako wejście zewnętrznego przerwania EXTI0.

PB11 (*TIM2_CH4*, w programie *PWM_T2CH4_OUT*)



PG0 (*GPIO_EXTI0*, w programie *EXT_INT*)

Do połączenia tych wyprowadzeń służy zworka z przewodu dołączonego do złącz na płytce Nucleo-144.

1.4 Materiały pomocnicze

Do wykonania ćwiczenia niezbędne lub przydatne są następujące materiały:

- projekt startowy *nucleo-basic-irq*,
- dokumentacja systemu operacyjnego FreeRTOS: <https://www.freertos.org/>,
- nota katalogowa (*datasheet*) mikrokontrolera STM32F429ZI,
- instrukcja obsługi (*reference manual*) mikrokontrolera STM32F429ZI,
- instrukcja obsługi (*user manual*) zestawu STM32F429 Nucleo-144,
- zestaw przykładów dla STM32F429 Nucleo-144.

2 Wykonanie ćwiczenia

2.1 Tryb stacjonarny

W trybie stacjonarnym relacjonujemy wykonanie kluczowych elementów ćwiczenia osobie prowadzącej zajęcia oraz przeprowadzamy dyskusję związaną wybranymi aspektami technicznymi realizacji pracy. Dzięki świadomej realizacji zadań osoby realizujące prace powinny być w stanie odpowiedzieć również na pytania osób prowadzących dotyczące aspektów realizacji ćwiczenia. Informacje o tej formie sprawozdawczości zostały zaznaczone dalszej części instrukcji przy pomocy takiego podświetlenia.

2.2 Tryb zdalny

Ta uwaga dotyczy tylko wykonania ćwiczenia w trybie zdalnym. Fragmenty zaznaczone w ten sposób stanowią wytyczne, co należy umieścić w sprawozdaniu **tekstowym i/lub wideo** (nagranie ekranu – *screencast*) z wykonania zadania w trybie zdalnym. Na zajęciach postępujemy wg wytycznych przedstawionych przez osobę prowadzącą na początku realizacji ćwiczenia. Pliki proszę nazwać wg informacji z niniejszej instrukcji. Materiały stanowiące sprawozdanie proszę zamieszczać na Teams, w indywidualnych kanałach zespołów, w sekcji *Pliki*, w utworzonym katalogu o nazwie **sprawozdanie-irq-dma**.

2.3 Import i otwarcie projektu

Import i otwarcie projektu przeprowadzamy w sposób analogiczny jak przy poprzednim ćwiczeniu „Niskopoziomowa implementacja systemu czasu rzeczywistego – na przykładzie FreeRTOS”. Tym razem jednak pobieramy i importujemy projekt startowy *nucleo-basic-irq*.

2.4 Obsługa przerwania zewnętrznego EXTI

W tej części ćwiczenia zaimplementujemy obsługę przerwania zewnętrznego EXTI0 (External Interrupt 0) od kontrolerów GPIO. W tym przypadku do generowania tego przerwania zostało skonfigurowane wyprowadzenie nr 0 kontrolera GPIOG, w skrócie PG0.

2.4.1 Konfiguracja GPIO

W projekcie startowym odpowiednia konfiguracja PG0 odbywa się jako jedna z wielu czynności przeprowadzanych w funkcji

```
static void MX_GPIO_Init(void).
```

Sama konfiguracja wyprowadzenia 0 portu GPIOG odbywa się w jej fragmencie:

```
/*Configure GPIO pin : EXT_INT_Pin */
GPIO_InitStruct.Pin = EXT_INT_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(EXT_INT_GPIO_Port, &GPIO_InitStruct);
```

Używane są tutaj także makrodefinicje z *main.h*:

```
#define EXT_INT_Pin GPIO_PIN_0
#define EXT_INT_GPIO_Port GPIOG
```

Pod koniec funkcji **MX_GPIO_Init** następuje włączenie aktywności wejścia PG0 jako EXTI0 przed skonfigurowaniem kontrolera przerw NVIC (*Nested Vectored Interrupt Controller*):

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI0_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(EXTI0_IRQn);
```

Dzięki temu sygnały przerw od EXTI0 będą mogły zostać prawidłowo obsługiwane przez NVIC.

Komentarze:

1. Układy peryferyjne GPIO generują wiele sygnałów przerw, ale nie taką liczbę jak liczba wszystkich dostępnych wyprowadzeń mikrokontrolera (tablica wektorów miałaby wtedy niepotrzebnie duży rozmiar). Część z tych przerw przypisana jest do pojedynczych linii (np. linia 0), a część jest zbiorcza, np. tak jak w przypadku do linii o numerach bitów od 15 do 10. Poniżej przedstawiono fragment pochodzący również z kodu projektu startowego:

```
HAL_NVIC_SetPriority(EXTI15_10_IRQn, 5, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```



Wszystkie tak zebrane bity każdego kontrolera GPIO wygenerują tę samą funkcję obsługi przerwania. Przy takim podejściu musimy sprawdzać w funkcji obsługi przerwania, czy

przerwanie przyszło np. z linii nr 0 kontrolera GPIOA (PA0) czy linii 0 kontrolera GPIOB (PB0), itd., w tym także omawiany tutaj GPIOG (PG0). W omawianym przypadku dodatkowo w funkcji obsługi trzeba sprawdzić, która linia wygenerowała przerwanie, ponieważ ta sama funkcja obsługi przerwania zostanie wywołana dla każdego wyprowadzenia o numerze z przedziału od 10 do 15.

2. Priorytet przerw w mikrokontrolerach ARM Cortex-M może być ustawiany przy pomocy dwóch liczb. W aktualnej konfiguracji posługujemy się tylko jedną liczbą opisującą priorytet przerwania, tutaj 5. **W przypadku STM32 im niższa liczba, tym wyższy priorytet przerwania.**

2.4.2 Obsługa wyjątku przy wykorzystaniu bibliotek HAL

Korzystając z kodu wysokopoziomowego, dodanie obsługi wyjątków w większości przypadków jest znacznie uproszczone. Sprawdźmy w praktyce, co mamy dostępne w kodzie startowym *nucleo-basic-irq*.

1. Kompilujemy próbnie projekt klikając ikonkę „młotek”  .
2. W pliku **stm32f4xx_it.c** tworzymy *breakpoint* na początku funkcji **void EXTI0_IRQHandler(void)**
Breakpoint tworzymy klikając dwa razy na numerze linii, na której chcemy zastawić pułapkę-breakpoint.
3. Startujemy sesję debugowania klikając na ikonkę „żuczek”  - przy okazji wgramy plik wynikowy do pamięci Flash. Po uruchomieniu programu powinien on po chwili „złapać się” na przygotowanym breakpointie na **EXTI0_IRQHandler**.

```
/**
 * @brief This function handles EXTI interrupt request.
 * @param GPIO_Pin Specifies the pins connected EXTI line
 * @retval None
 */
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    /* EXTI line interrupt detected */
    if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
    {
        __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
        HAL_GPIO_EXTI_Callback(GPIO_Pin);
    }
}

/**
 * @brief EXTI line detection callbacks.
 * @param GPIO_Pin Specifies the pins connected EXTI line
 * @retval None
 */
__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    /* Prevent unused argument(s) compilation warning */
    UNUSED(GPIO_Pin);
    /* NOTE: This function Should not be modified, when the callback is needed,
     the HAL_GPIO_EXTI_Callback could be implemented in the user file */
}
```

4. Obserwujemy przebieg programu, w tym przejście przez „słabą” (*weak*) funkcję **void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)**. Osoba prowadząca zajęcia może zapytać, jak działa mechanizm funkcji z atrybutem **weak**. Dlatego, na podstawie obserwacji

działania i dostępnych informacji próbujemy dowiedzieć się o co chodzi ;)

5. Napiszmy teraz w module *main.c* własną funkcję obsługi przerwania zewnętrznego. Proszę zwrócić uwagę, że nazwa tej funkcji zawiera **EXTI** (od **EX**Ternal **I**nterrupt), a nie „exit” (jak wyjście). Napisanie funkcji o innej nazwie niż dokładnie taka jak podana poniżej sprawi, że nasza funkcja nie zostanie w ogóle „zauważona” przez kompilator i linker jako zastępującą obserwowaną wcześniej funkcję z atrybutem *weak*.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    xprintf("EXTI0!\n");
}
```

6. Widzimy, że dzięki temu, że w module sterownika HAL funkcja ta miała atrybut *weak*, napisanie własnej funkcji o identycznym prototypie nie spowoduje pojawienia się błędu linkera. Teraz nasza funkcja jest bez atrybutu `__weak`, więc jest „silniejsza” i zostanie automatycznie wybrana do wywołania, gdy pojawi się przerwanie.
7. Tę funkcję zmodyfikujemy później, by zintegrować ją z mechanizmami synchronizacji systemu operacyjnego FreeRTOS.

Na platformie UPEL w zadaniu dotyczącym realizacji niniejszego ćwiczenia umieszczamy plik zmodyfikowany w tym punkcie plik *main.c* (lub copy-paste jego treści), jednak o nazwie zmienionej na **main_exti0.c**.

2.4.3 Obsługa handlera domyślnego (*default handler*)

Przeprowadzimy teraz eksperyment, który pomoże nam odpowiedzieć na pytanie: co się stanie, jeśli pojawi się wyjątek lub przerwanie, ale nie zaimplementujemy wysokopoziomowej funkcji obsługi tego przerwania?


Spróbujmy zatem tymczasowo „zakomentować” lub „deaktywować” (przez `#if 0` / `#endif`) funkcję **EXTI0_IRQHandler** z modułu *stm32f4xx_it.c*:

```
/**
 * @brief This function handles EXTI line0 interrupt.
 */
#if 0
void EXTI0_IRQHandler(void)
{
    /* USER CODE BEGIN EXTI0_IRQn 0 */

    /* USER CODE END EXTI0_IRQn 0 */
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
    /* USER CODE BEGIN EXTI0_IRQn 1 */

    /* USER CODE END EXTI0_IRQn 1 */
}
#endif
```

Jeśli w tym momencie skompilujemy program i wgramy go do pamięci Flash mikrokontrolera, to przy starcie programu zostanie zainicjalizowane przerwanie zewnętrzne od wyprowadzenia PG0. Jednak bez odpowiedniej funkcji wysokopoziomowej od razu zauważymy, że system „zawiesi się”.

Dlaczego tak się dzieje? Dzięki możliwości debuggowania łatwo możemy sprawdzić, dlaczego program zawiesił się: w stanie „zawieszenia” wystarczy wcisnąć przycisk „pauzy”  (*suspend*). Wbrew pozorom „zawieszenie się” systemu to dobry objaw, oznaczający, że przerwanie pojawia się, jednak brakuje nam pewnego elementu: zdefiniowanej funkcji jego obsługi. Zwróćmy uwagę na treść pliku **startup_stm32f429xx.s** projektu.

```
/**
 * @brief This is the code that gets called when the processor receives an
 *         unexpected interrupt. This simply enters an infinite loop,
preserving
 *         the system state for examination by a debugger.
 * @param None
 * @retval None
 */
.section .text.Default_Handler,"ax",%progbits
Default_Handler:
Infinite_Loop:
b Infinite_Loop
.size Default_Handler,.-Default_Handler
```

Mamy tutaj nieskończoną pętlę – ciągle wykonywany skok (*branch*, instrukcja **b**) do etykiety **Infinite_Loop**.

Z kolei nieco dalej, pod etykietą **g_pfnVectors** znajdują się stałe będące etykietami odpowiadającymi poszczególnym funkcjom obsługi różnych wyjątków (oczywiście poza elementem **_estack** informującym mikroprocesor o początkowym ustawieniu wskaźnika stosu).

g_pfnVectors:

```
.word _estack
.word Reset_Handler

.word NMI_Handler
.word HardFault_Handler
.word MemManage_Handler
.word BusFault_Handler
.word UsageFault_Handler
.word 0
.word 0
.word 0
.word 0
.word SVC_Handler
.word DebugMon_Handler
.word 0
.word PendSV_Handler
.word SysTick_Handler

/* External Interrupts */
.word WWDG_IRQHandler /* Window WatchDog */
.word PVD_IRQHandler /* PVD through EXTI Line detection */
/*
.word TAMP_STAMP_IRQHandler /* Tamper and TimeStamps through
the EXTI line */
.word RTC_WKUP_IRQHandler /* RTC Wakeup through the EXTI
line */
.word FLASH_IRQHandler /* FLASH */
.word RCC_IRQHandler /* RCC */
.word EXTI0_IRQHandler /* EXTI Line0 */
.word EXTI1_IRQHandler /* EXTI Line1 */
```

```

.word    EXTI2_IRQHandler    /* EXTI Line2    */
.word    EXTI3_IRQHandler    /* EXTI Line3    */

```

Wśród nich znajdziemy także interesującą nas **EXTI0_IRQHandler**. Pod każdą z tych stałych kryje się adres mówiący rdzeniowi, gdzie ma wykonać skok programu w momencie wystąpienia danego wyjątku. Mikroprocesor automatycznie skoczy do tego adresu, gdy wystąpi określony wyjątek. W dalszej części pliku widzimy jednak przypisania, np.

```

.weak    EXTI0_IRQHandler
.thumb_set EXTI0_IRQHandler,Default_Handler

```

Informują one kompilator, że wpisane uprzednio stałe/adresy są „słabe” (*weak*) i domyślnie będą zastąpione skokiem do **Default_Handler** chyba, że funkcja o danej konkretnej nazwie (tutaj **EXTI0_IRQHandler**) jawnie zostanie zdefiniowana w kompilowanych plikach. W naszym przypadku ta funkcja jeszcze nie została zdefiniowana, więc w wyniku podania aktywnego zbocza (opadającego) na na EXTI0 mikroprocesor skoczył do etykiety **Default_Handler**. Z kolei pod etykietą **Default_Handler** znajdziemy pustą, nieskończoną pętlę:

```

Infinite_Loop:
b Infinite_Loop

```

Możemy spróbować dopisać skok do własnej funkcji np. **My_Default_Handler** informującej o wystąpieniu domyślnego wyjątku:

```

Default_Handler:
b My_Default_Handler
Infinite_Loop:
b Infinite_Loop

```

Aby utworzyć funkcję własnego handlera domyślnego wyjątku, należy oczywiście (np. w module **main.c** lub, lepiej, w **Src/stm32f7xx_it.c**) napisać definicję tej funkcji, np.

```

void My_Default_Handler(void)
{
    xprintf("default handler!\n");
    while(1);
}

```

i skompilować projekt. Po wgraniu na płytkę przekonamy się, że po wciśnięciu przycisku USER, mikroprocesor przechodzi do funkcji **My_Default_Handler**. Jeśli nie korzystamy z debugera sprzętowego, tylko sprawdzamy działanie np. przy pomocy portu szeregowego i terminala, warto mieć taką funkcję zaimplementowaną, aby od razu było wiadomo, że pojawia się nieobsługiwane przerwanie. Aktualnie w ćwiczeniu, jej implementacja nie jest obowiązkowa, choć należy wiedzieć jak ją wykonać.

Uwaga: deklaracja funkcji **xprintf** znajduje się w pliku nagłówkowym **term_io.h** – dołączenie tego pliku na początku modułu **Src/stm32f4xx_it.c** zapobiegnie ostrzeżeniom kompilatora.

2.4.4 Przywrócenie obsługi wyjątku od EXTI0

Przywróćmy teraz wcześniejszą konfigurację, w której mamy zaimplementowaną i „widoczną” w kodzie funkcję `EXTI0_IRQHandler`. Ona z kolei, przez warstwy bibliotek HAL wywoła nam callback:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    xprintf("EXTI0!\n");
}
```

Sprawdzamy, czy wszystko działa poprawnie.

Uwaga: w docelowej wersji oprogramowania pozostawienie komunikatu generowanego z *printf* wewnątrz funkcji obsługi przerwania jest najczęściej niezalecane lub nawet niedopuszczalne, ponieważ z dużym prawdopodobieństwem będzie ono zajmowało dużo czasu mikroprocesora. Z drugiej strony w funkcji obsługi przerwania mikroprocesor nie może obsłużyć innych przerw o niższych priorytetach co powoduje degradację jego możliwości reakcji w czasie rzeczywistym. Jest to szczególnie ważne np. w zastosowaniach przemysłowych lub IoT, gdy nasz system mikroprocesorowy ma służyć do sterowania pewnym procesem lub interakcji z otoczeniem.

2.4.5 Synchronizacja wyjątku i zadania systemu FreeRTOS

Funkcje obsługi przerwania powinny zajmować jak najmniej czasu mikroprocesora. Dlatego dobrą praktyką jest wykonywanie w tych funkcjach tylko najbardziej krytycznych czasowo czynności. Jeśli potrzeba wykonać także inne, bardziej czasochłonne fragmenty kodu, warto jest je przenieść z funkcji obsługi przerwania do pętli głównej lub zadania systemu operacyjnego. W przypadku jednowątkowego programu z pętlą główną, można to uczynić przy pomocy prostej flagi (zmiennej, w której przechowujemy 1 bit informacji) np. ustawianej w funkcji obsługi przerwania, a kasowanej w pętli głównej. W przypadku systemu operacyjnego FreeRTOS mamy do dyspozycji znacznie ciekawszy i często stosowany w praktyce mechanizm. Otóż możemy zastosować semafor (a w nowszych wersjach systemu mamy także specjalne notyfikacje dla zadań), który będziemy ustawiać w funkcji obsługi przerwania i „wychwytywać” jego aktywność w zadaniu. Dodatkowo, zadanie systemu operacyjnego w takiej konfiguracji może czekać na gotowość semafora nie zajmując czasu systemu czyli będąc w podobnym stanie jakby trwało wywołanie `vTaskDelay`. Taki mechanizm zaimplementujemy w dalszej części niniejszego podpunktu.

UWAGA! Na tym etapie należy zajrzeć do oficjalnej dokumentacji systemu operacyjnego FreeRTOS, funkcja API: **`xSemaphoreGiveFromISR`**. Znajdziemy tam przykład kodu odpowiadający realizowanemu zadaniu.

W module `main.c` mamy utworzony semafor (obiekt typu `xSemaphoreHandle`), który będzie wznawiał wykonywanie zadania. Ma on nazwę `sem_EXTI` i jest zmienną globalną. Semafor `sem_EXTI` jest inicjalizowany w `StartDefaultTask` przez wywołanie `xSemaphoreCreateBinary`, a następnie w programie następuje sprawdzenie, czy obiekt `sem_EXTI` został poprawnie zainicjalizowany, tj. czy ma wartość inną niż `NULL`. Dodatkowo zajmujemy semafor, ponieważ liczymy, że zostanie on zwolniony w funkcji obsługi przerwania, co stanie się sygnałem do kontynuacji zadania.

```
sem_EXTI = xSemaphoreCreateBinary();
```

```
if(sem_EXTI==NULL) xprintf("Error: sem_EXTI not created\n");
xSemaphoreTake(sem_EXTI,0);
```

Co należy zrobić:

1. W funkcji obsługi przerwania zastępujemy dotychczasowy kod wyświetlający komunikat na terminalu (**printf**) wywołaniem funkcji „dającej” semafor. Tutaj ważna uwaga: aby cały mechanizm mógł działać poprawnie, musi to być specjalna odmiana tej funkcji, tj. **xSemaphoreGiveFromISR**, bezpieczna w wywołaniu wewnątrz funkcji obsługi przerwania (m.in. nie może zawierać opóźnień). Dodatkowo potrzebna jest zmienna (najlepiej typu **BaseType_t**) np. o nazwie **switch_required** informująca, czy należy wykonać przełączanie zadań w wyniku wystąpienia przerwania. Zmienna ta może być ustawiana przez funkcję **xSemaphoreGiveFromISR**. Na samym końcu funkcji obsługi przerwania wartość tej zmiennej powinna zostać sprawdzona w funkcji **portYIELD_FROM_ISR()** z argumentem będącym wartością **switch_required**. Warto sprawdzić, co kryje się pod tymi makrodefinicjami w pliku *portmacro.h*.

Przykład implementacji funkcji obsługi przerwania od EXTI0 znajduje się poniżej:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    BaseType_t xHigherPriorityTaskWoken;
    xSemaphoreGiveFromISR( sem_EXTI, &xHigherPriorityTaskWoken );
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

2. W zadaniu **ledTask** w module *main.c* proszę zaimplementować oczekiwanie na odbiór semafora **sem_EXTI** przez odpowiednie wywołanie **xSemaphoreTake**. Jako parametr określający maksymalny czas oczekiwania na semafor (*timeout*) powinna zostać wpisana stała **portMAX_DELAY**. Oznacza to nieskończenie długi czas oczekiwania - dlatego nie będzie wymagany blok warunkowy. Jeśli w zadaniu otrzymamy semafor (**xSemaphoreTake** zwróci wartość „prawda”), jednokrotnie powinna zabłysnąć dioda LED. Możemy także wypisać dodatkowy komunikat, aby zaprezentować, że zaimplementowany mechanizm zgłaszania przerw działa poprawnie.

Z kolei zadanie **ledTask** może mieć teraz następującą przykładową postać:

```
static void ledTask(void* p)
{
    while(1)
    {
        xprintf("waiting for sem_EXTI...\n");
        xSemaphoreTake(sem_EXTI,portMAX_DELAY);
        xprintf("sem_EXTI has been set :)\n");

        HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_SET);
        vTaskDelay(500);
        HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_RESET);
        vTaskDelay(850);
    }
}
```

Uwaga: jeśli semafor nie był wcześniej zainicjalizowany przez wywołanie **xSemaphoreCreateBinary** (ma wartość NULL) to próba jego ustawienia zakończy się wyjątkiem i

„zawieszeniem” działania programu w pustej pętli obsługi tego wyjątku.

Jako efekty realizacji tego podpunktu, proszę zaprezentować osobie prowadzącej działanie oprogramowania oraz na platformie UPEL proszę umieścić:

- plik graficzny **semaphore.png** lub **.jpg** ze zrzutem ekranu, na którym będzie widoczne okno terminala wyświetlającego komunikaty generowane w zadaniu **ledTask** przy wykonaniu oprogramowania z tego punktu,
- kopię pliku **main.c** lub copy-paste jego treści utworzonej przy realizacji niniejszego punktu, z nazwą zmodyfikowaną na **main_sem.c**.

2.5 Zastosowanie przerwań w interfejsach komunikacyjnych

W podstawowej wersji oprogramowania do odbioru danych z interfejsu UART1 zastosować funkcję **inkey** z modułu **dbg.c**.

```
char inkey(void)
{
    uint32_t flags = pUart->Instance->SR;
    if((flags & UART_FLAG_RXNE) || (flags & UART_FLAG_ORE))
    {
        __HAL_UART_CLEAR_OREFLAG(pUart);
        return (pUart->Instance->DR);
    }
    else
        return 0;
}
```

Ta funkcja wykorzystuje podejście **polling** polegające na ciągłym sprawdzaniu czy nie pojawiły się nowe dane. Pojawienie się nowych danych sygnalizowane jest aktywnością flagi RXNE (*RX Not Empty*). Zbyt rzadkie wywoływanie funkcji odczytującej **inkey** doprowadzi do „zgubienia” nadchodzących znaków i uaktywnienia flagi ORE (*Overflow*) w interfejsie szeregowym.

Z powodu szeregu niedoskonałości w powyższym podejściu, w ramach ćwiczenia spróbujemy uczynić odbiór znaków bardziej niezawodnym. Jednym ze sposobów jest zastosowanie systemu przerwań.

2.5.1 Włączenie przerwania

Interfejs UART3 w projekcie **nucleo-basic-irq** jest zainicjalizowany w zakresie podstawowym. Aby włączyć wybrane źródło przerwań, wystarczy uaktywnić odpowiadający mu bit, tutaj RXNE. Przy zastosowaniu bibliotek HAL włączanie przerwań odbywa się przy pomocy makrodefinicji **__HAL_UART_ENABLE_IT**. Jako argumenty przekazujemy wskaźnik na zainicjalizowany obiekt reprezentujący interfejs UART oraz nazwę flagi generującej przerwanie:

```
__HAL_UART_ENABLE_IT(&huart3, UART_IT_RXNE);
```

Zalecane jest wywołanie tego makra *tuż przed* uruchomieniem systemu FreeRTOS, np. w sekcji **USER CODE BEGIN / END 2**, w funkcji **main**..:

```
/* USER CODE BEGIN 2 */
debug_init(&huart3);
xprintf(ANSI_BG_MAGENTA "\nNucleo-144 F429ZI nucleo-basic-irq project" ANSI_BG_DEFAULT "\n");
xprintf(ANSI_FG_YELLOW "Funkcja xprintf działa." ANSI_FG_DEFAULT "\n");
printf(ANSI_FG_GREEN "Funkcja printf tez działa." ANSI_FG_DEFAULT "\n");
```

```

__HAL_UART_ENABLE_IT(&huart3, UART_IT_RXNE);
/* USER CODE END 2 */

/* USER CODE BEGIN RTOS_MUTEX */
/* add mutexes, ... */
/* USER CODE END RTOS_MUTEX */

/* USER CODE BEGIN RTOS_SEMAPHORES */
/* add semaphores, ... */
/* USER CODE END RTOS_SEMAPHORES */

/* USER CODE BEGIN RTOS_TIMERS */
/* start timers, add new ones, ... */
/* USER CODE END RTOS_TIMERS */

/* Create the thread(s) */
/* definition and creation of defaultTask */
osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 1024);
defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

```

Wyłączenie przerwania odbędzie się przez wywołanie analogicznej makrodefinicji `__HAL_UART_DISABLE_IT`.

2.5.2 Funkcja obsługi przerwania

Funkcje obsługi przerwania w projekcie startowym znajdują się w module `Src/stm32f4xx_it.c`. Za bezpośrednią obsługę przerwania od USART3 odpowiada następująca funkcja:

```

/**
 * @brief This function handles USART3 global interrupt.
 */
void USART3_IRQHandler(void)
{
    /* USER CODE BEGIN USART3_IRQn 0 */

    /* USER CODE END USART3_IRQn 0 */
    HAL_UART_IRQHandler(&huart3);
    /* USER CODE BEGIN USART3_IRQn 1 */

    /* USER CODE END USART3_IRQn 1 */
}

```

Widzimy, że element biblioteki HAL korzysta już z tej funkcji (wywołanie `HAL_UART_IRQHandler(&huart3);`), ale mimo to mamy możliwość dopisania do niej własnego kodu wg wygenerowanych komentarzy.

Wykonanie zadania:

1. W module `main.c` stworzymy funkcję:

```
void uart_rxirq_callback(void)
```

Będzie to nasza nowa funkcja obsługi przerwania.

2. W module `Src/stm32f4xx_it.c` modyfikujemy funkcję `USART3_IRQHandler` w taki sposób, aby w jej wnętrzu wywoływana była wyłącznie funkcja `uart_rxirq_callback` a następnie następowało wyjście z niej przy pomocy `return`. Zamiast tego, można także „wykomentować” wywołanie tymczasowo niepotrzebnego `HAL_UART_IRQHandler`.

Uwaga: Nie jest zalecane usuwanie dotychczasowego kodu `USART3_IRQHandler`, ponieważ będzie on potrzebny w dalszej części ćwiczenia.

```

/**
 * @brief This function handles USART3 global interrupt.
 */
void USART3_IRQHandler(void)
{
    /* USER CODE BEGIN USART3_IRQn 0 */
    uart_rxirq_callback();
    return;
    /* USER CODE END USART3_IRQn 0 */
    HAL_UART_IRQHandler(&huart3);
    /* USER CODE BEGIN USART3_IRQn 1 */

    /* USER CODE END USART3_IRQn 1 */
}

```

3. W module `stm32f4xx_it.c` dodajemy deklarację **extern** naszej funkcji obsługi przerwania:

```
extern void uart_rxirq_callback(void);
```

4. W module `mian.c` w funkcji `uart_rxirq_callback` tworzymy własny kod zarządzający odbiorem znaku. Sprawdzamy, czy uaktywniony jest bit RXNE:

```
if(__HAL_UART_GET_IT_SOURCE(&huart3,UART_IT_RXNE))
```

5. Jeśli tak, to odczytujemy znak z rejestru DR:

```
char chr = huart3.Instance->DR;
```

6. Możemy wyświetlić tymczasowy komunikat (uwaga jak w punkcie 2.4.4), np.

```
xprintf("RXNE, chr = %c = %02X\n",chr,(uint8_t)chr);
```

Cała funkcja na dotychczasowym etapie:

```

void uart_rxirq_callback(void)
{
    if(__HAL_UART_GET_IT_SOURCE(&huart3,UART_IT_RXNE))
    {
        char chr = huart3.Instance->DR;
        xprintf("RXNE, chr = %c = %02X\n",chr,(uint8_t)chr);
    }
}

```

Na tym etapie należy zaprezentować działanie oprogramowania osobie prowadzącej oraz na platformie UPEL proszę zamieścić następujące pliki:

- statyczny zrzut ekranu o nazwie `rxirq_callback.png` lub `.jpg`, na którym będzie widoczne okno terminala PuTTY wyświetlające komunikaty z funkcji `uart_rxirq_callback`,
- aktualną kopię pliku `stm32f4xx_it.c` z nazwą zmodyfikowaną na

`uart_rxirq_stm32f4xx_it.c` ,

- aktualną kopię pliku `main.c` z nazwą zmodyfikowaną na `uart_rxirq_main.c`.

2.5.3 Skierowanie odebranych znaków do kolejki

Mając niezawodnie działającą funkcję odbierającą znaki i system operacyjny, możemy każdy odebrany znak umieścić w kolejce, co zapewni nam bardzo elegancki system buforowania nadchodzących danych. Dzięki temu problemy z potencjalnym „gubieniem” znaków opisane w punkcie 2.5 praktycznie mogą przestać nas dotyczyć.

Korzystając z dokumentacji systemu operacyjnego FreeRTOS oraz dotychczasowych doświadczeń należy w prawidłowy sposób dodać nadchodzący znak z interfejsu UART1 do kolejki `loopQueue` i sprawdzić działanie programu. Należy pamiętać o zastosowaniu odpowiedniej, bezpiecznej funkcji dodawania do kolejki z poziomu funkcji obsługi przerwania (`xQueueSendFromISR`) oraz obsłudze ewentualnej zmiany kontekstu (`portYIELD_FROM_ISR`).

Dane z kolejki będą teraz odbierane w zadaniu `messageTask`.

Na platformie upel proszę zamieścić kopię pliku `main.c` zmodyfikowanego w niniejszym punkcie, pod nazwą `queue_main.c`.

2.5.4 Usunięcie tymczasowych modyfikacji

Po implementacji i zaprezentowaniu działania utworzonego systemu osobie prowadzącej zajęcia, cofamy wykonane krytyczne modyfikacje projektu i przygotowujemy projekt do realizacji dalszych ćwiczeń.

1. W module `Src/stm32f4xx_it.c` w funkcji `USART3_IRQHandler` przywracamy „oficjalne” działanie z wywołaniem funkcji sterownika HAL: `HAL_UART_IRQHandler(&huart3);`.

```
/**
 * @brief This function handles USART3 global interrupt.
 */
void USART3_IRQHandler(void)
{
    /* USER CODE BEGIN USART3_IRQn 0 */

    /* USER CODE END USART3_IRQn 0 */
    HAL_UART_IRQHandler(&huart3);
    /* USER CODE BEGIN USART3_IRQn 1 */

    /* USER CODE END USART3_IRQn 1 */
}
```

2. W module `main.c` usuwamy wcześniej umieszczone wywołanie makrodefinicji włączającej przerwanie od RXNE:

~~`HAL_UART_ENABLE_IT(&huart3, UART_IT_RXNE);`~~

2.6 Obsługa kanału DMA

W tej części ćwiczenia dodamy do sterownika portu szeregowego UART3/USART3 funkcjonalność odbioru danych przez kanał DMA. Sterownika *dbgu.c* nie musimy zmieniać – dodamy tylko dodatkową funkcję inicjalizującą kanał DMA np. w module *main.c*. W ten sposób dane wpisane przy pomocy terminala trafią przez interfejs USART3 bezpośrednio do pamięci operacyjnej mikrokontrolera.

2.6.1 Bufor w pamięci operacyjnej

Aby transfer DMA mógł się odbywać, musimy utworzyć bufor. Może to być zwykła tablica elementów typu *char*. Rozmiar bufora dobrze jest zdefiniować jako stałą (np. makrodefinicją), ponieważ będziemy się do niego kilkakrotnie odnosić w kodzie programu przy realizacji ćwiczenia. Rozmiar bufora do realizacji ćwiczenia także nie powinien być ustawiony na zbyt dużą wartość, aby bufor dało się w całości łatwo wyświetlić np. na wyświetlaczu LCD płytki testowej lub w jednej linii programu terminalowego. Przykład:

```
#define UART_RX_DATA_SIZE      10
static uint8_t uartRxData[UART_RX_DATA_SIZE];
```

2.6.2 Inicjalizacja kontrolera DMA

W tej części zainicjalizujemy kontroler DMA w taki sposób, aby utworzyć tzw. bufor kołowy (*circular buffer*). Dane odbierane z interfejsu UART3/USART3 będą wpisywane automatycznie do utworzonego bufora odbiorczego. Po osiągnięciu końca bufora, sprzętowy „wskaźnik” modułu DMA powinien wrócić na początek bufora i tam zapisywać dane.

W module *main.c* należy utworzyć na razie pustą funkcję *setup_rx_dma* – w niej zaimplementujemy inicjalizację kontrolera DMA. Funkcję wywołujemy w *main* po inicjalizacji sterowników (np. po *debug_init*) ale przed uruchomieniem systemu operacyjnego (przed *vTaskStartScheduler*).

Przy kodzie inicjalizującym wg opisu, kanał DMA będzie działał na zasadzie bufora kołowego, tj. dane będą zapisywane znak po znaku aż do końca bufora, a następnie kontroler automatycznie wróci na początek bufora.

Sterowniki niskopoziomowe dostarczone domyślnie z projektem *nucleo-basic-irq* wykonują inicjalizację DMA w module *Src/stm32f4xx_hal_msp.c* w funkcji *HAL_UART_MspInit*. Inicjalizacja ta nie powoduje uzyskania mechanizmu działania bufora kołowego, jednak może być dobrym punktem startu do wzorowania się w pisaniu własnego sterownika.

Przykład kodu inicjalizującego kanał DMA dla UART3/USART3 do trybu bufora kołowego znajduje się poniżej:

```
static void setup_rx_dma(void)
{
    hdma_usart3_rx.Instance = DMA1_Stream1;
    hdma_usart3_rx.Init.Channel = DMA_CHANNEL_4;
    hdma_usart3_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;
    hdma_usart3_rx.Init.PeriphInc = DMA_PINC_DISABLE;
    hdma_usart3_rx.Init.MemInc = DMA_MINC_ENABLE;
    hdma_usart3_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
```



```

hdma_usart3_rx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
hdma_usart3_rx.Init.Mode = DMA_CIRCULAR;
hdma_usart3_rx.Init.Priority = DMA_PRIORITY_LOW;
hdma_usart3_rx.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
//inicjalizacja kanału DMA z parametrami j.w.
if (HAL_DMA_Init(&hdma_usart3_rx) != HAL_OK)
{
    xprintf("error setting up UART3 DMA\n");
}
__HAL_LINKDMA(&huart3, hdmarx, hdma_usart3_rx);
//usuwamy "zalegające" znaki z rejestru odbiorczego UART
__HAL_UART_FLUSH_DRREGISTER(&huart3);
//start transferu DMA
HAL_UART_Receive_DMA(&huart3, uartRxData, UART_RX_DATA_SIZE);
}

```

Na koniec funkcji inicjalizującej widzimy wywołanie rozpoczynające właściwy transfer DMA:

```
HAL_UART_Receive_DMA(&huart3, uartRxData, UART_RX_DATA_SIZE);
```

Do powyższej funkcji podajemy:

- wskaźnik do obiektu reprezentującego interfejs UART,
- wskaźnik do bufora na dane utworzonego w punkcie 2.6.1,
- rozmiar bufora również zdefiniowany wg opisu z 2.6.1.

2.6.3 Prezentacja działania

Należy utworzyć nowe lub wykorzystać uprzednio utworzone zadanie systemu FreeRTOS, w którym będziemy na bieżąco pokazywać na terminalu co kilkaset tyknięć dwie informacje:

- liczbę znaków, która pozostała do zapełnienia bufora,
- zawartość całego bufora.

Wskazówka: Liczbę znaków do zapełnienia możemy sprawdzić korzystając z makrodefinicji `__HAL_DMA_GET_COUNTER`. Jej implementację możemy sprawdzić w pliku

`Drivers/STM32F4xx_HAL_Driver/Inc/stm32f4xx_hal_dma.h`.

W celu sprawdzenia działania, wysyłamy z terminala znaki i obserwujemy zmianę zapełnienia bufora oraz jego zawartość.

Proszę zaprezentować zaimplementowany odbiór znaków przez kanał DMA oraz proszę zamieścić na UPEL aktualną kopię pliku `main.c` nazwaną `dma_main.c`, z modyfikacjami wprowadzonymi na etapie realizacji niniejszego punktu instrukcji.

2.6.4 Implementacja kolejki - nieobowiązkowa łamigłówka dla osób zainteresowanych

Znając mechanizm działania bufora kołowego zasilanego w dane przez kontroler DMA, można spróbować zaimplementować kolejkę korzystając z wbudowanych w mikrokontroler sprzętowych mechanizmów DMA (nie używamy tutaj mechanizmu kolejek systemu FreeRTOS). Do

„sprzętowo-wspomaganej” kolejki dane będzie wpisywał kontroler DMA (z terminala), natomiast odczyt będzie się odbywał np. w wyniku cyklicznego pojawiania się przerwania EXTI0. Każde kolejne wystąpienie przerwania od EXTI0 powinno wymuszać odczyt jednego znaku z kolejki oraz wyświetlać odczytany z kolejki znak na terminalu. Jeśli w buforze kołowym nie będzie nowych (jeszcze nie wyświetlonych) znaków, to piszemy komunikat mówiący np. że „bufor jest pusty”.

3 Zagadnienia

Wymienione w niniejszej sekcji zagadnienia podsumowujące mają służyć do utrwalenia zdobytej wiedzy i umiejętności. Zalecane jest ich samodzielne opracowanie na podstawie dostępnych materiałów i doświadczeń zdobytych na laboratorium.

1. Mechanizm obsługi przerw w wykorzystywanych na laboratorium projektach *nucleo-basic* oraz mechanizm obsługi wyjątków przy zastosowaniu mikrokontrolerach STM32F4x:
 - a. tablica wektorów: co zawiera, gdzie się znajduje?
 - b. wyjątek domyślny (*Default Handler*): do czego jest przydatny, czy i jak można „podmienić” jego obsługę?
 - c. konieczność sprawdzania i kasowania bitów statusowych generujących wyjątki od układów peryferyjnych – co się stanie, jeśli ich nie skasujemy?
2. Mechanizm działania i możliwości kontrolera DMA w mikrokontrolerach STM32F4x zastosowanych w ćwiczeniu:
 - a. które rodzaje transferów umożliwia: układ peryferyjny-pamięć, pamięć-pamięć?
 - b. czy i w których sytuacjach może generować sygnały przerw?
3. Rozumienie sensu, korzyści i ew. problemów wynikających z zastosowania:
 - a. systemu przerw,
 - b. transferów DMA,
 - c. integracji systemu obsługi przerw z mechanizmami synchronizacji systemu operacyjnego FreeRTOS