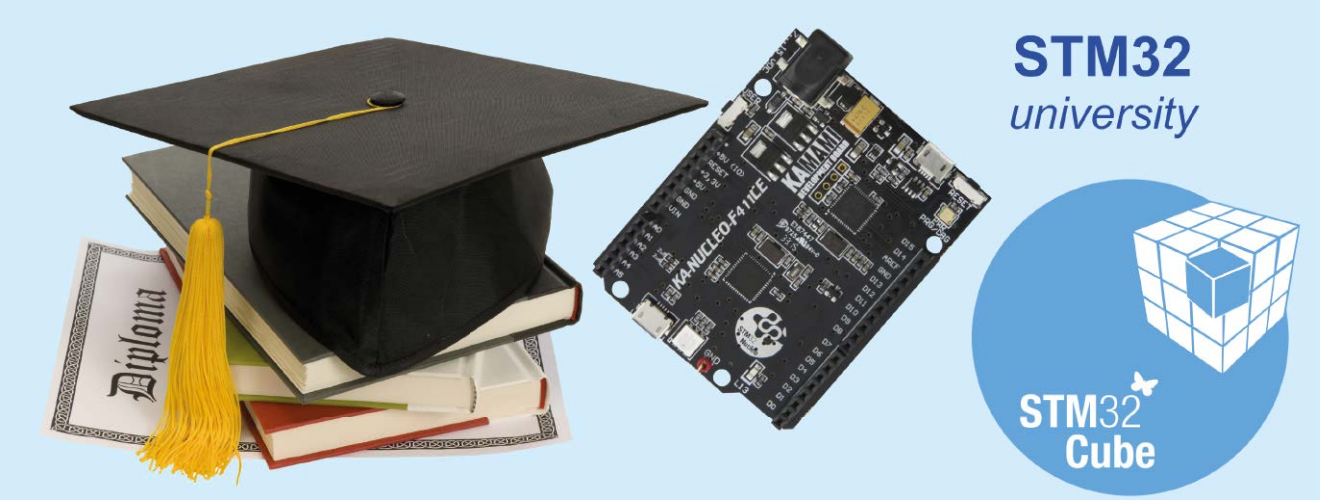


Programowanie układu STM32F4 (3)

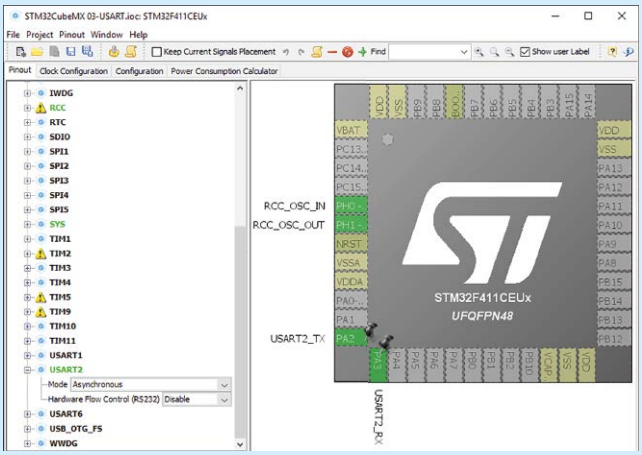
W tej części cyklu umożliwimy naszemu mikrokontrolerowi kontakt z komputerem PC. Wykorzystamy w tym celu interfejs USART oraz znajdujący się na płytce rozwojowej programator ST-LINK lub osobny adapter UART/USB. Za pomocą interfejsu USART możemy również sterować wieloma dostępnymi na rynku układami rozszerzającymi funkcjonalność naszego procesora – na przykład modemami GSM lub modułami z ESP8266 pełniącym funkcję karty sieciowej Wi-Fi.

Znajdujący się na płytce programator, to nic innego, jak kolejny układ z serii STM32, z wyprowadzonym złączem USB od strony użytkownika oraz interfejsami SWD i UART przyłączonymi do układu, który programujemy. Ma on wgrane oprogramowanie emulujący programator ST-LINK, a po przyłączeniu do komputera, przedstawia się jako trzy niezależne urządzenia USB – pamięć masowa, właściwy programator oraz port szeregowy (COM).

Na używanej przeze mnie płytce KA-NUCLEO-F411CE również i z głównego układu wyprowadzono USB dostępne dla użytkownika, więc moglibyśmy je użyć w roli interfejsu komunikacyjnego i nie korzystać z adaptera lub programatora. Znajomość konfiguracji interfejsu UART będzie nam jednak potrzebna w kolejnych częściach kursu, do komunikacji z innymi układami.

Wykrywanie przez system portu COM to nie przypadek. Interfejs USART jest częściowo kompatybilny z dawnym standardem RS232. Różnica polega na wykorzystywaniu innych poziomów napięć – standardowej logiki układu (u nas CMOS – 0 i 3,3 V) oraz kilku dodatkowych wyprowadzeniach w dawnym standardzie. Format przesyłanej ramki jest ten sam i za pomocą popularnego układu MAX232 możemy dołączyć nasz układ do portu COM komputera.

Wspomniałem już o interfejsach UART i USART. Czym jednak się one różnią? Interfejs UART jest interfejsem asynchronicznym, co oznacza, że urządzenia po obu stronach mogą zacząć nadawać w dowolnej chwili i nie wymieniają się sygnałem zegara. UART wykorzystuje jedynie dwa wyprowadzenia – jedno nadawcze, a drugie odbiorcze (łącząc układy należy jeszcze pamiętać o połączeniu ich mas). Interfejs USRT – oprócz sygnałów nadawczego i odbiorczego – wysyła również przebieg zegarowy określający jasno, w których momentach transmitowane są poszczególne bity. Jest on łatwiejszy w implementacji sprzętowej i zabezpiecza nas przed rozsynchronizowaniem się transmisji przy jej większych prędkościach, jednak używa dodatkowego wyprowadzenia.



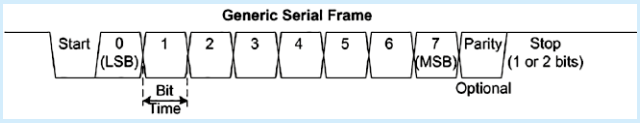
Rysunek 1. Konfiguracja pinów w programie STM32CubeMX

Akronim USART to zbiorcza nazwa określająca oba standardy oraz interfejs mikrokontrolera, który może pracować w obu wymienionych trybach.

Niegdyś bardzo popularny interfejs RS232 miał więcej wyprowadzeń – min. sygnały Ready to Send oraz Clear to Send, sygnalizujące kolejno, że dane urządzenia ma w swoim buforze dane gotowe do wysłania oraz że przeciwne może je w tej chwili odebrać (w buforze jest na to miejsce). Standard RS232 pierwotnie był używany do łączenia komputerów z modemami, a dalej poprzez linie telefoniczne z innymi komputerami.

W środowisku Arduino, interfejs UART znalazł zastosowanie do programowania mikrokontrolerów z wgranym bootloaderem oraz do debugowania – okienko Serial Monitor. My użyjemy „lekkiego", darmowego programu **PuTTY**. Możemy go pobrać ze strony ***https://***

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: **ftp://ep.com.pl**, user: **33948**, pass: **5gcckdmq**



Rysunek 2. Ramka danych interfejsu UART (źródło: https://goo.gl/siS1h1)

goo.gl/YRIlCP (potrzebujemy jedynie pliku putty.exe). Program ten jest dostępny również dla systemów Unixowych, jednak ich użytkownikom polecam konsolowy **minicom** lub **screen**.

Pierwszy projekt

W dalszej części tego artykułu, omówimy działanie przerwań oraz wykorzystamy je, aby nie blokować pracy procesora, w trakcie oczekiwania na polecenia wysyłane z komputera. Nasz pierwszy projekt będzie jednak bardzo prymitywny – będzie on odbierał z komputera tekst wprowadzony przez użytkownika i odsyłał go, poprzedzając słowem „Odebrano: [...]", aktywnie czekając na odbiór danych. Będzie to również bardzo dobry przykład tego, jak nie należy używać interfejsu UART…

Uruchamiamy oprogramowanie STM32CubeMX, tworzymy nowy projekt oraz wybieramy układ. Dla przypomnienia, przykłady z niniejszego cyklu wykonywane są na płytce rozwojowej KA-NUCLEO-F411CE, z układem STM32F411CEU6.

Na ekranie zostanie wyświetlony obraz wyprowadzeń układu (**rysunek 1**). Musimy teraz zdecydować, na jakich pinach chcemy uruchomić interfejs. Rzecz jasna, nie na każdym z nich możemy to zrobić. Wykorzystywany przeze mnie układ ma 3 osobne interfejsy USART, które możemy uruchamiać na 5 różnych parach pinów. Jeśli korzystamy z płytki KA-NUCLEO i chcemy użyć w roli adaptera UART <–>USB wbudowany programator, używane przez nas wyprowadzenia to: PA2 w roli pinu nadawczego i PA3 w roli pinu odbiorczego. Jeśli korzystamy z innej płytki lub chcemy przyłączyć adapter USB albo układ MAX232 do wyprowadzeń płytki, musimy wyszukać wyprowadzeń w jej datasheecie. Z poziomu STM32CubeMX możemy natomiast sprawdzić, na których pinach układu można uruchomić interfejs – klikamy w tym celu lewym przyciskiem myszy na piny i szukamy funkcji alternatywnej o nazwie „USARTx_TX" lub „USARTx_RX". Gdy chcemy ustawić wybrane piny, wybieramy z menu te funkcje. Do działania interfejsu UART potrzebujemy pinu TX oraz RX. Dla interfejsu USRT, dodatkowo pinu CK – zegara.

Po wybraniu odpowiednich pinów, z listy po lewej stronie wyszukujemy wybranego interfejsu *USARTx*, gdzie *x* to jego numer i wybieramy tryb pracy – w polu *Mode* ustawiamy opcję *Asynchronous* lub *Synchronous*. Możemy także dodać omawiane wcześniej piny RTS i CTS – odpowiada za to pole *Hardware Flow Control (RS232)*. Na razie jednak ustawmy tryb pracy na *Asynchronous* i nie włączajmy pinów RTS/CTS.

W kolejnym kroku konfigurujemy pętlę PLL, w sposób identyczny, jak opisany w pierwszej części kursu. Najpierw, w zakładce *Pinout*, na liście po lewej stronie odszukujemy pozycję *Peripherals* → *RCC* i z listy rozwijanej w polu *High Speed Clock (HSE)* wybieramy pozycję *Crystal/Ceramic Resonator*. Następnie, w zakładce *Clock Configuration*, w polu *PLL Source Mux*, jako źródło wybieramy *HSA* i ustawiamy mu prawidłową częstotliwość (na płytce KA-NUCLEO jest to 8 MHz), w polu *System Clock Mux* wybieramy opcję *PLLCLK*. Teraz w polu *HCLK* ustawiamy pożądaną częstotliwość. Do nauki, możemy wybrać maksymalną dostępną (dla układu na płytce KA-NUCLEO będzie to 100 MHz).

Dalej, przechodzimy do zakładki *Configuration*. W polu *Connectivity* pojawił się nowy interfejs USART. U mnie – *USART2*. Aby go skonfigurować, klikamy przycisk *USARTx*. Możemy teraz ustawić szybkość nadawania i odbioru, a także inne parametry przesyłanych ramek danych. Popularnymi szybkościami pracy są 115200 bps oraz 9600 bps. Jeśli chcemy sterować układem, który ma interfejs USART, powinniśmy sprawdzić, jaką szybkość ustawić w datasheecie tego układu. Podobnie ma się sprawa z liczbą bitów stopu i także bitem parzystości.

Ramka interfejsu USART (**rysunek 2**) składa się z pojedynczego bitu startu ramki (zawsze logicznego zera), ośmiu bitów danych, bitu parzystości oraz jednego lub dwóch bitów stopu (logicznych jedynek). Bit parzystości obliczany jest z przesyłanych bitów danych – jeśli występuje w nich parzysta ilość jedynek, to bit parzystości będzie logiczną jedynką, przeciwnie – zerem. Domyślnie, jego obliczanie jest wyłączone, możemy je włączyć, ustawiając w polu *Parity* wartość *Odd*. Możliwe jest także ustawienie bitu „nieparzystości" działającego przeciwnie do bitu parzystości – wtedy wybieramy wartość *Even* (**rysunek 3**).

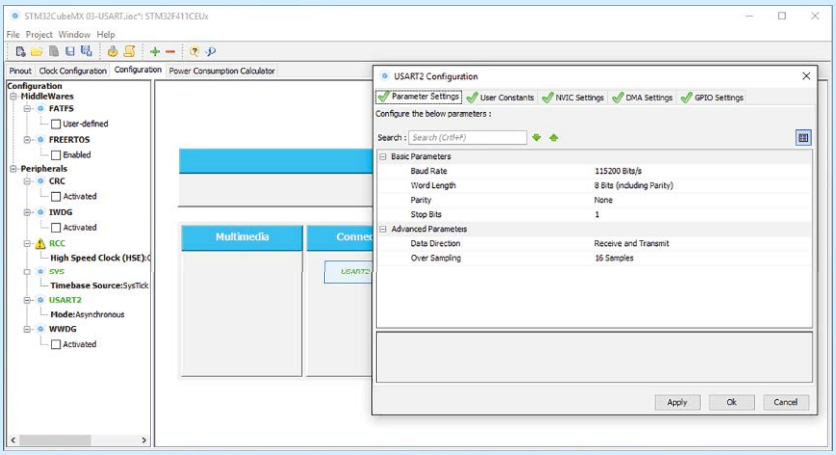
Na potrzeby komunikacji z komputerem, pozostawiamy domyślne ustawienia. Możemy już wygenerować projekt dla środowiska System Workbench for STM32 – klikamy w ikonę zębatki na pasku narzędziowym, zmieniamy wartość pola *Toolchain/IDE* na *SW4STM32* i klikamy przycisk OK, następnie importujemy projekt w tymże środowisku – zamykamy planszę powitalną, klikamy PPM w puste pole w ramce *Project Explorer* i wybieramy kolejno: *Import...* → *General* → *Existing Project into Workspace*. Odszukujemy nasz projekt na dysku i klikamy *OK*.

Następnie otwieramy plik *Src/main.c* i w sekcji *USER CODE 0* dopisujemy funkcje z **listingu 1**. Początkowo, ułatwią nam one komunikacje z komputerem. W dalszej części artykułu omówione zostaną także bezpośrednie wywołania HAL-a. Do sekcji *USER CODE 3* wpisujemy poniższy kod – wywołanie zadeklarowanych wcześniej funkcji:

```
char buffer[1024];
uart_read_line(&huart2, buffer, 1024);
uart_write(&huart2, „Odebrano: „);
uart_write_line(&huart2, buffer);
```

Program będzie odbierał od użytkownika tekst, aż do klawisza Enter na klawiaturze, a następnie odeśle go z powrotem, poprzedzając słowem „Odebrano: [...]".

W pierwszej linii powyższego kodu, tworzymy tablicę 1024 komórek typu char (przechowujących kolejne znaki ASCII wprowadzonego tekstu – litery, cyfry oraz symbole). Następnie wywołujemy funkcję *uart_read_line()*. Funkcja ta przyjmuje na wejściu wskaźnik na bufor i w trakcie swojego działania zapisuje do niego kolejne odebrane znaki, aż do odebrania znaku nowej linii lub przekroczenia długości bufora – wtedy w miejscu, w którym kończy się nasz ciąg znaków, w kolejnej komórce bufora, zapisywany jest znak \0'



Rysunek 3. Konfiguracja interfejsu UART w programie STM32CubeMX

– null-terminator, oznaczający koniec ciągu. Funkcja uart_read_line(), poza wskaźnikiem na bufor, przyjmuje na wejściu również wskaźnik na strukturę konfiguracyjną interfejsu UART (%huartX, gdzie X to nr interfejsu, struktura ta jest generowana automatycznie przez CubeMX). Następnie, w niemal identyczny sposób, korzystając z funkcji uart_write() oraz uart_write_line() wyświetlamy napis „Odebrano: [...]" oraz odebrany ciąg zwieńczony znakami końca linii („\r\n" – kody ASCII powodujące powrót kursora na początek linii oraz przejścia do nowej).

Listing 1. Modyfikacja sekcji USER CODE 0

```
/* USER CODE BEGIN 0 */
void uart_write(UART_HandleTypeDef * handler, char * text) {
    HAL_UART_Transmit(handler, text, strlen(text), 1000);
}

void uart_write_line(UART_HandleTypeDef * handler, char * text) {
    HAL_UART_Transmit(handler, text, strlen(text), 1000);
    HAL_UART_Transmit(handler, „\r\n", 2, 1000);
}
uart_read_line(UART_HandleTypeDef * handler, char * buffer, uint16_t buffer_size) {
    HAL_StatusTypeDef status;
    char current_char;
    uint16_t char_counter = 0;
    while (char_counter < buffer_size - 1) {
        status = HAL_UART_Receive(handler, &current_char, 1, 1);
        if (status == HAL_OK) {
            if (current_char == ,\r' || current_char == ,\n')
                if (char_counter == 0) continue; else break;
            *(buffer + char_counter++) = current_char;
        }
    }
    *(buffer + char_counter) = ,\0';
}
/* USER CODE END 0 */
```

Teraz możemy skompilować kod i wgrać go na układ (ikony młotka i robaka na pasku menu) oraz uruchomić program PuTTY i „porozmawiać" z mikrokontrolerem. W oknie programu PuTTY, w polu *Connection Type* wybieramy opcję *Serial*, w polu *Speed* wpisujemy wybraną szybkość połączenia (domyślnie 115200 bps). W polu *Serial line* podajemy nazwę portu szeregowego – **rysunek 4**. Tą ostatnią poznamy w systemowym Menadżerze Urządzeń (**rysunek 5**), w sekcji *Porty (COM i LPT)*, pod Windowsem oraz w logu kernela, w systemach Unixowych (Linuksowe polecenie dmesg).

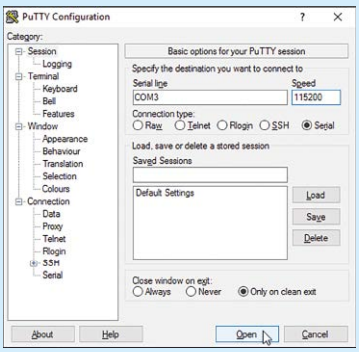
W ten sposób, możemy przesyłać z naszego mikrokontrolera do komputera informacje o parametrach jego pracy, wartościach odebranych z czujników, czy odbierać od komputera polecenia (**rysunek 6**). W dalszej części spróbujemy wysterować diodę RGB na podstawie odebranych od użytkownika poleceń zwiększenia lub zmniejszenia jasności kolejnych barw składowych. Do zamiany wartości numerycznych, na ciągi znaków możemy wykorzystać funkcję sprintf(), działającą podobnie jak funkcja printf(), z tą różnicą, że zwraca wynik w postaci ciągu znaków, który następnie wysyłamy przez UART, a nie wyświetla go.

Sterowanie kolorem świecenia diody RGB z komputera

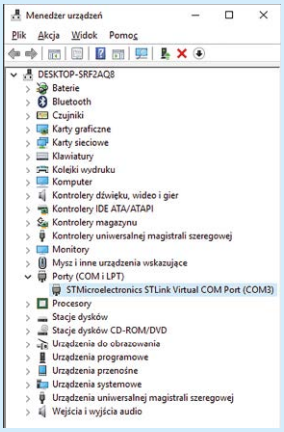
Uruchamiamy CubeMX i otwieramy w nim nasz projekt. Zgodnie z instrukcją z poprzedniej części, ustawiamy piny diody RGB jako wyjściowe piny generatora PWM, konfigurujemy licznik oraz generujemy projekt z wprowadzonymi zmianami. Jeśli stosowaliśmy się do sekcji *USER CODE*, nie powinniśmy teraz utracić naszego kodu. Ponownie uruchamiamy System Workbench lub odświeżamy listę plików (polecenie Refresh w Project Explorer). W sekcji *USER CODE 0* wpisujemy fragment kodu z **listingu 2**, obsługujący korekcję gamma. Funkcji z poprzedniego etapu możemy się już pozbyć (wrócimy do nich w kolejnej części cyklu). Do sekcji *USER CODE 2* wpisujemy poniższy kod z **listingu 3**, powodujący uruchomienie generatora PWM na pinach. W sekcji *USER CODE 3* wpisujemy kod z **listingu 4** służący do odbierania danych i zmiany koloru świecenia LED.

W pierwszej linii kodu z list. 4, w sekcji *USER CODE 3*, tworzymy bufor o rozmiarze jednego znaku. Następnie odbieramy ten znak od użytkownika do buforu, korzystając z funkcji *HAL_UART_Receive()* – jej kolejnymi parametrami są: wskaźnik na strukturę konfiguracyjną interfejsu, wskaźnik na bufor, długość bufora oraz timeout – czas, po jakim funkcja ma się poddać i zwrócić informację o niepowodzeniu, jeśli nie otrzyma żadnych danych. Dalej sprawdzamy, czy funkcja zwróciła informację o odebraniu danych i na podstawie odebranego znaku decydujemy, jasność, jakiej barwy zwiększamy lub zmniejszamy. Jeśli wychodzimy przy tym poza skalę – od 0 do 1, w kolejnych liniach kodu, powracamy do właściwego zakresu. Na koniec, korzystając z omówionej, w poprzedniej części kursu funkcji *set_led_brightness()*, ustawiamy jasność poszczególnych kanałów diody RGB.

Kompilujemy oraz uruchamiamy program na mikrokontrolerze. Jeśli środowisko nadal nie widzi części plików lub zmiennych, możemy zmusić je do przeindeksowania zawartości projektu – wybieramy w tym celu z menu, w górnej części okna, polecenie: *Project → C/C++ Index → Freshen All Files*. Należy też pamiętać o błędzie związanym z biblioteką „m". Sposób jego rozwiązania został podany w poprzedniej części – w ustawieniach linkera musimy odznaczyć opcję dołączania biblioteki „m" oraz dodać ją ręcznie na liście dołączanych bibliotek.



Rysunek 4. Ustawienia połączenia w PuTTY

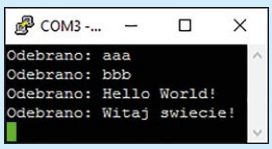


Rysunek 5. Menedżer Urządzeń

Od teraz, po wciśnięciu w programie PuTTY klawiszy „q", „w" i „e", zwiększymy jasność świecenia poszczególnych kolorów składowych diody RGB – odpowiednio: czerwonego, zielonego i niebieskiego. Analogicznie, klawiszami „a", „s" i „d" zmniejszymy ich jasność.

Przerwania

W tej chwili, przez większość czasu swojej pracy, nasz procesor oczekuje na odebranie od użytkownika danych. Po ich odebraniu, interpretuje je, wykonuje stosowną akcje i powraca do oczekiwania. To rozwiązanie sprawdza się w przypadku prostych ćwiczeń. Co jednak w sytuacji, gdy chcielibyśmy w międzyczasie wykonywać jakieś inne operacje? Przykładowo – obsłużyć połączenia TCP/IP, przerysowywać zawartość wyświetlacza czy zmierzyć odległość od przeszkody, do której zbliża się nasz robot itp.



Rysunek 6. Komunikacja z mikrokontrolerem w PuTTY

Z pomocą przychodzą nam przerwania. Możemy ustawić interfejs UART tak, aby w momencie otrzymania od urządzenia po drugiej

Listing 2. Obsługa korekcji gamma

```
/* USER CODE BEGIN 0 */
void set_led_brightness(TIM_HandleTypeDef * timer, uint32_t channel, double brightness) {
    int32_t value = powf(brightness, 2.2) * 49999;
    __HAL_TIM_SET_COMPARE(timer, channel, value);
}
/* USER CODE END 0 */
```

Listing 3. Uruchomienie generatora PWM

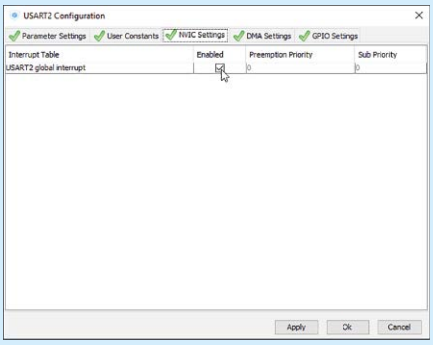
```
/* USER CODE BEGIN 2 */
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_1);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_2);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_3);
/* USER CODE END 2 */
```

Listing 4. Odbiór danych i zmiana koloru świecenia LED

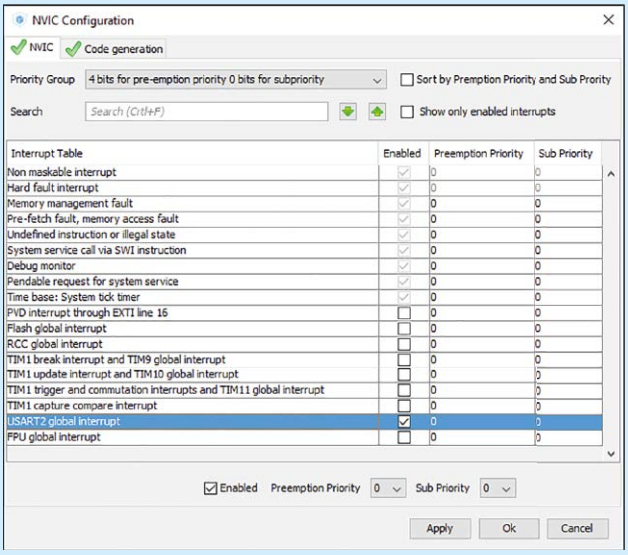
```
uint8_t key[1];
HAL_StatusTypeDef status = HAL_UART_Receive(&huart2, key, 1, 1);

if (status == HAL_OK) switch (key[0]) {
    case 'q': red += 0.01;   break;
    case 'w': green += 0.01; break;
    case 'e': blue += 0.01;  break;
    case 'a': red -= 0.01;   break;
    case 's': green -= 0.01; break;
    case 'd': blue -= 0.01;  break;
}

if (red > 1)   red = 1;   if (red < 0)   red = 0;
if (green > 1) green = 1; if (green < 0) green = 0;
if (blue > 1)  blue = 1;  if (blue < 0)  blue = 0;

set_led_brightness(&htim1, TIM_CHANNEL_1, blue);
set_led_brightness(&htim1, TIM_CHANNEL_2, red);
set_led_brightness(&htim1, TIM_CHANNEL_3, green);
```

stronie danych, kazał procesorowi przerwać wykonywanie obecnej operacji i przetworzył przychodzące dane. Procesor zrzuca wtedy na swój stos zawartość rejestrów (kontekst) oraz przeskakuje pod odpowiedni adres w pamięci kodu, w którym to umieścimy naszą funkcję obsługującą przerwanie. Po jej wykonaniu, procesor przywróci zawartość rejestrów i powróci do normalnej pracy.

Przerwania mogą oczywiście generować również inne interfejsy i peryferiale. Mogą one także być wywoływane z zewnątrz – po zmianie stanu wybranego pinu. Możemy ustawić dowolny licznik, tak, aby wywoływał w ustalonych odstępach czasu przerwania (najczęściej w momencie resetu wartości licznika). Przerwaniom możemy również przypisywać priorytety – te o niższym priorytecie nie mogą przerywać tych o wyższym. Procesor, pomiędzy wykonywaniem rozkazów sprawdza wektor przerwań – jeśli jakieś przerwanie wymaga obsłużenia, w wektorze tym znajdzie się odpowiednio



Rysunek 7. Włączanie obsługi przerwań w ustawieniach interfejsu w programie STM32CubeMX



Rysunek 8. Ustawienia kontrolera przerwań w programie STM-32CubeMX

ustawiona flaga – w ten sposób, przerwania o różnych priorytetach mogą być kolejkowanie.

Aby uruchomić generowanie przerwań przez interfejs UART, wracamy do programu STM32CubeMX i wczytujemy w nim nasz projekt. Następnie przechodzimy do zakładki *Configuration* i wybieramy interfejs USART z sekcji *Connectivity*. W nowootwartym oknie przechodzimy do zakładki *NVIC Settings* i zaznaczamy jedynego checkboxa na liście (w polu *Enable* – **rysunek 7**). Alternatywnie, w zakładce *Configuration*, moglibyśmy wybrać opcje *NVIC* – wtedy wyświetli nam się lista wszystkich możliwych do uruchomienia w danym projekcie przerwań. Możemy tam także ustawiać ich priorytety (**rysunek 8**).

Teraz możemy już wygenerować projekt i dokonać kilku zmian w kodzie programu, w środowisku System Workbench. Do sekcji *USER CODE 0* dopisujemy funkcję z **listingu 5**. Funkcja ta będzie wywoływana za każdym razem, gdy interfejs USART odbierze dane i wywoła przerwanie. Jej zawartość jest podobna do poprzedniego kodu wywoływanego w pętli. Zawartość sekcji *USER CODE 2* zmieniamy na tę pokazaną na **listingu 6** – dodajemy tutaj instrukcję włączającą obsługę przerwania. Do sekcji *USER CODE PV* dopisujemy kod z **listingu 7** – przenosimy zmienne do sekcji globalnej, tak, aby były dostępne zarówno z poziomu funkcji obsługującej przerwanie, jak również z funkcji *main()*. Sekcję *USER CODE 3* czyścimy – nie potrzebujemy w tej chwili, aby jakikolwiek kod wykonywał się w pętli.

Po uruchomieniu mikrokontrolera, wykonany zostanie kod z sekcji „USER CODE 2". W ostatniej linii tego kodu, korzystając z funkcji HAL_UART_Receive_IT(), włączamy przerwanie interfejsu UART. Składnia tej funkcji jest niemal identyczna jak składnia funkcji odbierającej dane do bufora. Tutaj również wskazujemy, gdzie zapisany ma zostać odebrany ciąg oraz jak długi jest bufor. Przed przeskoczeniem do wykonania funkcji przerwania, do wskazanego bufora zapisane zostaną odebrane dane. Żeby bufor był dostępny z zarówno z poziomu funkcji main(), jak i funkcji obsługującej przerwanie, deklarujemy go jako zmienną globalną w sekcji *USER CODE PV*. Na końcu funkcji obsługującej przerwanie, ponawiamy wywołanie funkcji uruchamiającej przerwanie – włącza ona obsługę przerwania, aż do jego następnego wywołania.

Kody źródłowe przykładów oraz projektu programu STM32CubeMX poszczególnych przykładów, dostępne są na serwerze FTP. Następną część kursu poświęcimy układowi ESP8266. Dodamy dzięki niemu do naszego mikrokontrolera obsługę sieci Wi-Fi oraz stosu TCP/IP. Utworzymy także stronę WWW, za której pomocą będziemy sterowali „naszą" diodą RGB.

Aleksander Kurczyk

Listing 5. Funkcja wywoływana przez przerwanie USART

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    switch (key[0]) {
        case 'q': red += 0.01;   break;
        case 'w': green += 0.01; break;
        case 'e': blue += 0.01;  break;
        case 'a': red -= 0.01;   break;
        case 's': green -= 0.01; break;
        case 'd': blue -= 0.01;  break;
    }
    if (red > 1)   red = 1;   if (red < 0)   red = 0;
    if (green > 1) green = 1; if (green < 0) green = 0;
    if (blue > 1)  blue = 1;  if (blue < 0)  blue = 0;
    set_led_brightness(&htim1, TIM_CHANNEL_1, blue);
    set_led_brightness(&htim1, TIM_CHANNEL_2, red);
    set_led_brightness(&htim1, TIM_CHANNEL_3, green);
    HAL_UART_Receive_IT(&huart2, key, 1);
}
```

Listing 6. Zmiany w sekcji USER CODE 2

```
/* USER CODE BEGIN 2 */
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_1);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_2);
HAL_TIMEx_PWMN_Start(&htim1, TIM_CHANNEL_3);
HAL_UART_Receive_IT(&huart2, key, 1);
/* USER CODE END 2 */
```

Listing 7. Modyfikacja sekcji USER CODE PV

```
/* USER CODE BEGIN PV */
/* Private variables ---...---*/
double red = 0, green = 0, blue = 0;
uint8_t key[1];
/* USER CODE END PV */
```