Okładka 1

Łatwiejszy niż dla Arduino

Praktyczny kurs programowania STM32

z wykorzystaniem CubeIDE oraz HAL i bibliotek LL

Sławomir Skrzyński
sas.ze@vp.pl
Wersja z dnia 7.12.2024 godzina 12:39

Wstęp:

Jeśli chcesz realizować zaawansowane projekty jak:

AVT3275 czy http://kolejki.prv.pl/dekoderySTM32/ to książka dla Ciebie!

Przedpłaty na książkę można uiszczać wpłacając 60zł przez serwis: https://suppi.pl/ermik
W wiadomości dla autora należy napisać „Książka kurs STM32” oraz swój e-mail. Osoby, które uiszczą przedpłatę otrzymają finalną wersję książki w PDF bez względu na jej końcową cenę. Ponadto będą dostawać kolejne rozdziały w trakcie ich powstawania jeszcze przed ukazaniem się finalnego produktu.

Książka jest przeznaczona dla zaawansowanych użytkowników STM32 jakkolwiek początkujący tez mogą z niej skorzystać wspierając się kursami dostępnymi na YouTube (także autora) czy książkami dla początkujących. Nie ukrywam też, że nie jest to kurs C. Bez jego znajomości ciężko będzie zrozumieć wiele zagadnień. Główny nacisk położono na praktyczne aspekty programowania aczkolwiek w wielu wypadkach będzie trochę „wiedzy akademickiej” czasem niezbędnej aby uzyskać zamierzony efekt. Książka jest rozwinięciem internetowych kursów:
Kurs STM32 z wykorzystaniem CubeIDE
Kurs C jakiego jeszcze nie było!
Kurs LL - Programowania STM32 w wykorzystaniem bibliotek LL
Książka jest napisana nietypowo i integralną jej częścią są filmy w liczbie 92 o łącznym czasie trwania xxxx minut.
W książce poruszone będzie wiele zagadnień pomijanych w innych publikacja uważanych za trudne jak HOST USB czy slave SPI / I2C. Prawdą jest, że trudne wydają się tematy, których się nie rozumie ale gdy zostaną wytłumaczone w prosty sposób okazują się banalnie proste.

Autor książki jest znany z niebanalnych projektów takich jak Centrala telefoniczna CTAC: http://es2.noip.pl/ctac64/ centrala 2x4: http://er-mik.prv.pl/ct2x4/ Domofon GSM: http://er-

mik.prv.pl/projekty_edw/2019-04_EP_(SaS)_Domofon_GSM.pdf czy precyzyjny Licznik częstotliwości i czasu AVT-3275: https://www.youtube.com/playlist?list=PLdtkbzWTUVMl67a8ouZUBjK8GDIzT1naZ

Poza czystym programowaniem w książce będą poruszane tematy sprzętowe jak podłączanie zewnętrznych układów, wyświetlaczy czy czujników. Ze względu na to, ze autor pisząc na STM32 korzystał zarówno z HAL (najwięcej) jak i bibliotek LL oraz używał rejestrów, poruszy „owiany" tajemnicą temat bibliotek LL. Warto zapoznać się z materiałem dostępnym w: https://www.youtube.com/playlist?list=PLdtkbzWTUVMkX_pBDn6tHqxY0u_d_kZ07 poruszający temat LL w mikrokontrolerach o małych zasobach (STM32 C0 mającymi zastąpić dużo gorsze i droższe 8-bit uC jak AVR, Xmega czy PIC).

Ze względu na to, że autor bardzo długo używał AVR (od około 2008 do 2018 roku, wcześniej 1994-2008 8051 ale także 6502, Z80, Z8, PIC) często będzie porównywał STM32 do AVR ukazując jego słabości tak jak i do Arduino (na które to napisał kilka bibliotek), które jest przeznaczone raczej dla dzieci ale niestety pojawiają się próby jego użycia w komercyjnych zastosowaniach.

Do przejścia kursu niezbędny będzie zakup kilku płytek NUCLEO i DISCOVERY STM32. Jestem przekonany, że uda mi się uzyskać rabat na płytki od dystrybutorów dla osób, które zakupią książkę.

Ze względu na to, że w książce niemożliwe jest pokazanie nagrań video pojawi się wiele linków do filmów, które nie będą powszechnie dostępne przez wyszukiwarkę (np. w YouTube czy Google) a tylko dla osób, które zakupiły książkę https://www.youtube.com/playlist?list=PLdtkbzWTUVMlDguV5VUNp2qV-mV2EwTEy

Na ile można mi zaufać płacąc za książkę przed jej napisaniem w całości? Proponuję zapoznać się z materiałem dostępnym na „Elektrozgrozie": https://www.elektroda.pl/rtvforum/topic3552750.html WSZYSCY którzy uiścili przedpłatę otrzymali zamawiany wyrób! Mało tego, deklarowałem obudowę bez wyfrezowanych otworów a zamawiający otrzymali zmontowane urządzenie w profesjonalnie wyfrezowanej obudowie!

Jeśli są tematy, których książka nie porusza a chciałbyś się o nich coś dowiedzieć (rozwiązać swój problem) zapraszam do konsultacji on-line. W ten sprawie pisz do autora: sas.ze@vp.pl można też telefonować na numer 22-690-0610 poniedziałek...piątek: 10:00-16:00 sobota 11:00-14:00 a kto lubi FAX to proszę wysyłać na 22-690-0619. Niestety teleksu (dalekopisu) nie posiadam, bo i po co skoro ta usługa (popularna do połowy lat 90) nie jest dostępna

od 2007 roku.

Materiały dodatkowe do książki (grafiki, fotografie, programy, sample, legi analizatora, pdf, itp: https://drive.google.com/Materiały Dodatkowe

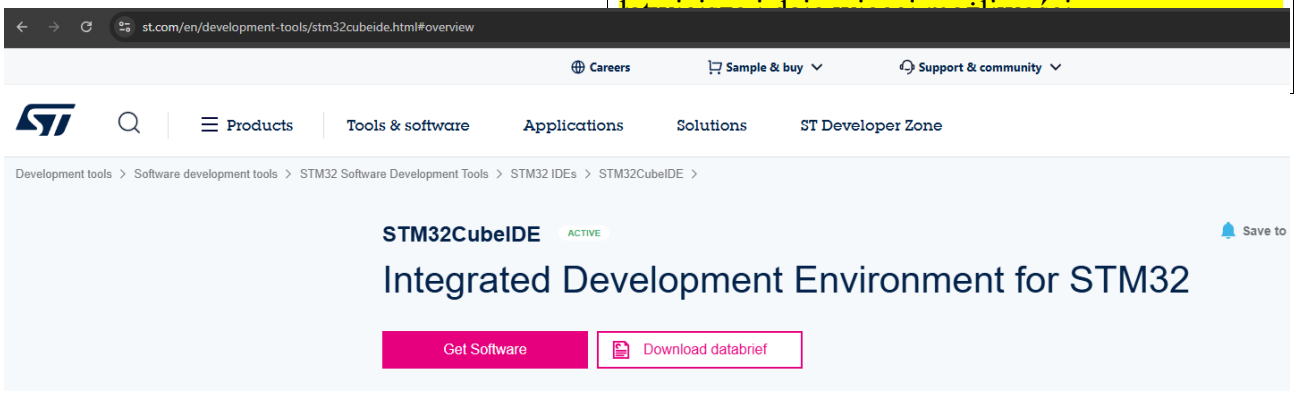
Spis treści

Instalacja i konfiguracja CubeIDE

Pakiet oprogramowania pobieramy ze strony:

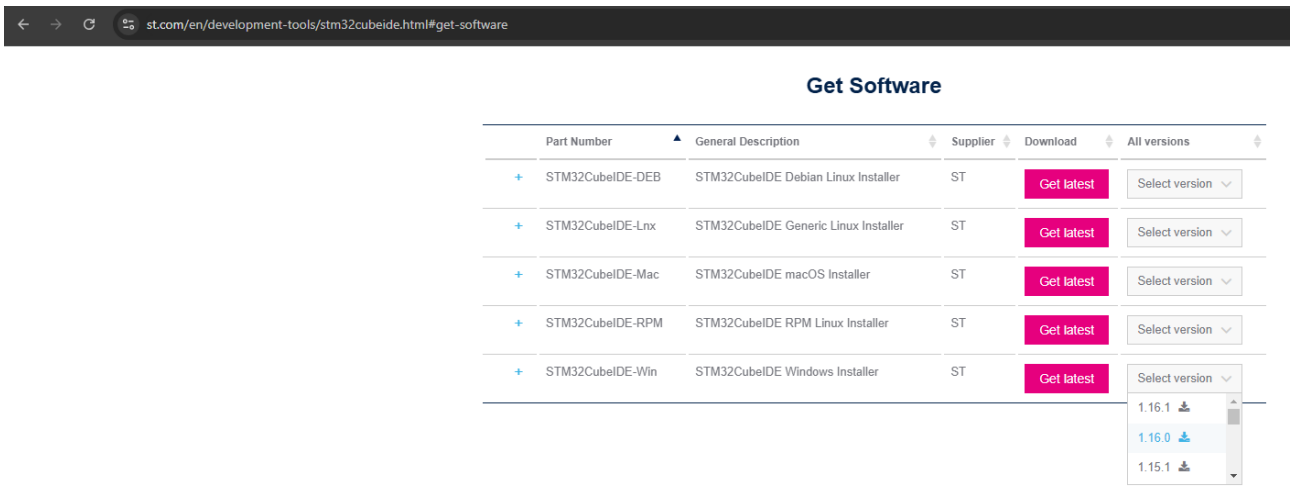
https://www.st.com

Instalacja CubeIDE jest tak samo prosta jak AVR Studio czy Arduino przy czym używanie jest dużo



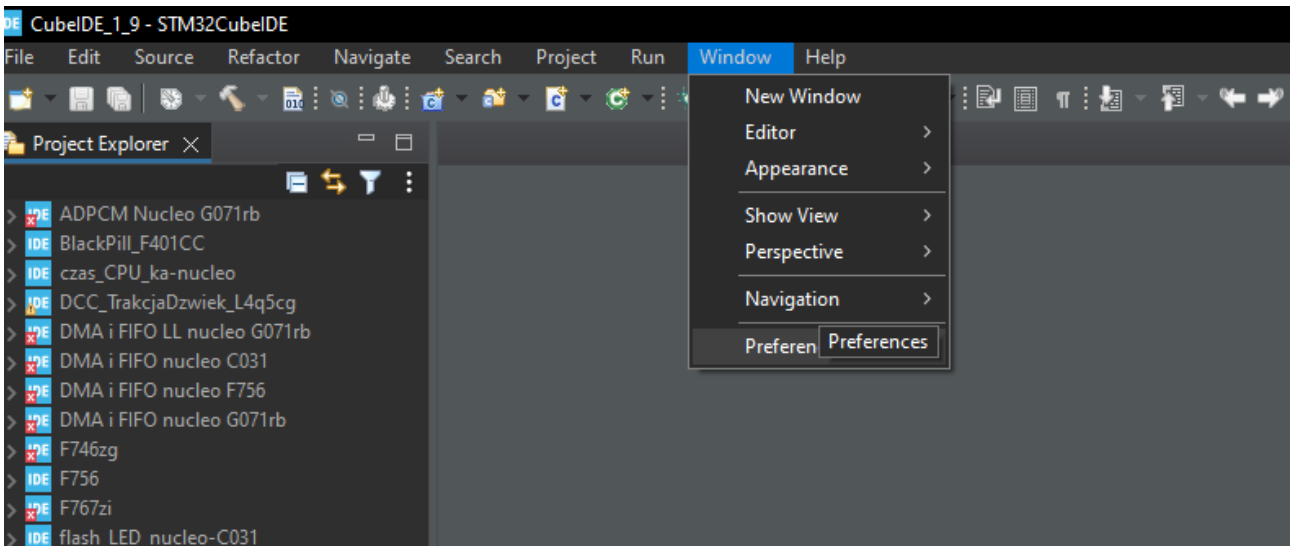
Rysunek 0000

Wybieramy system operacyjny i wersję środowiska na jaką się decydujemy:



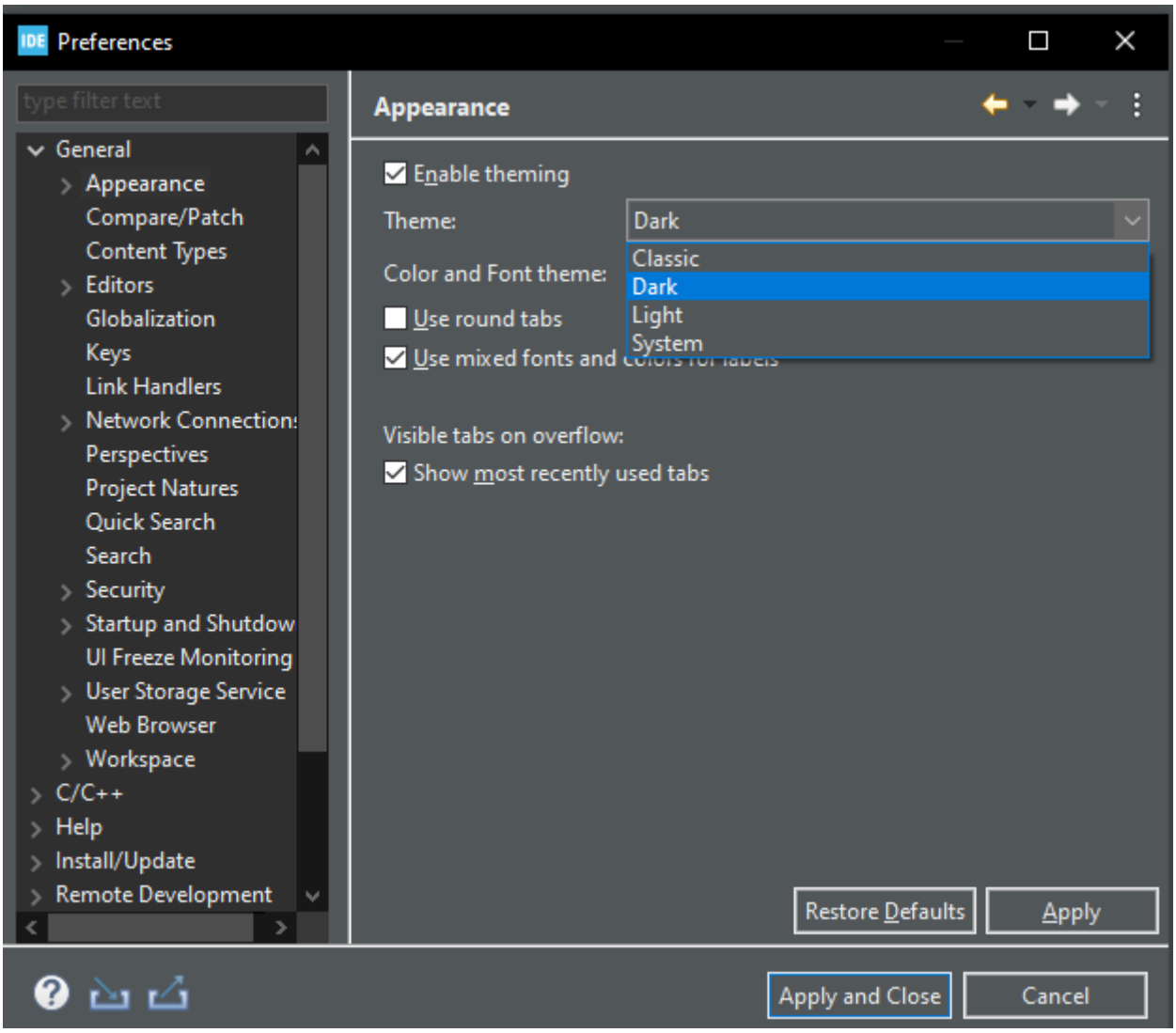
Rysunek 0001

Po zainstalowaniu i uruchomieniu środowiska należy je skonfigurować:



Rysunek 0003

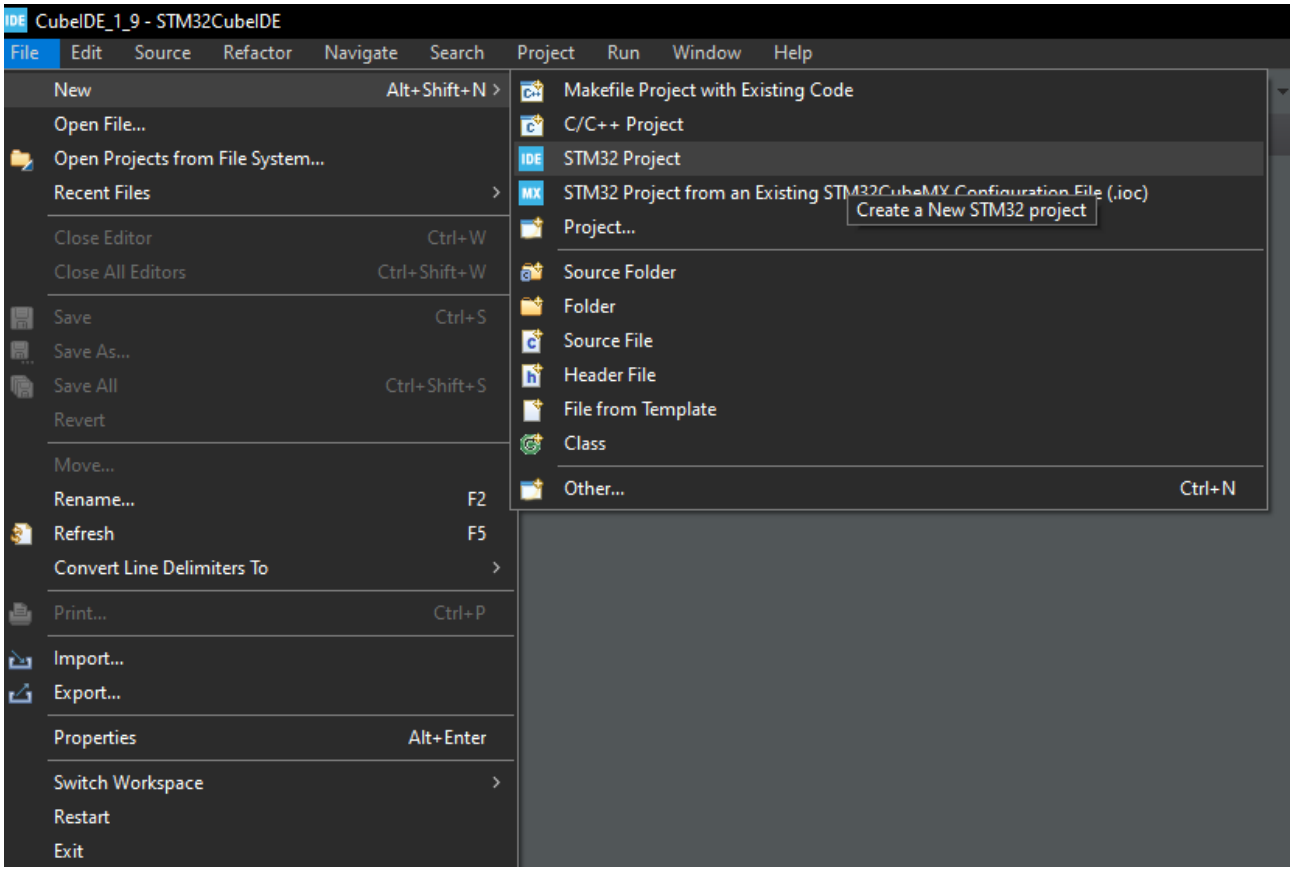
Na początek polecam wybrać tryb ciemny:



Rysunek 0004

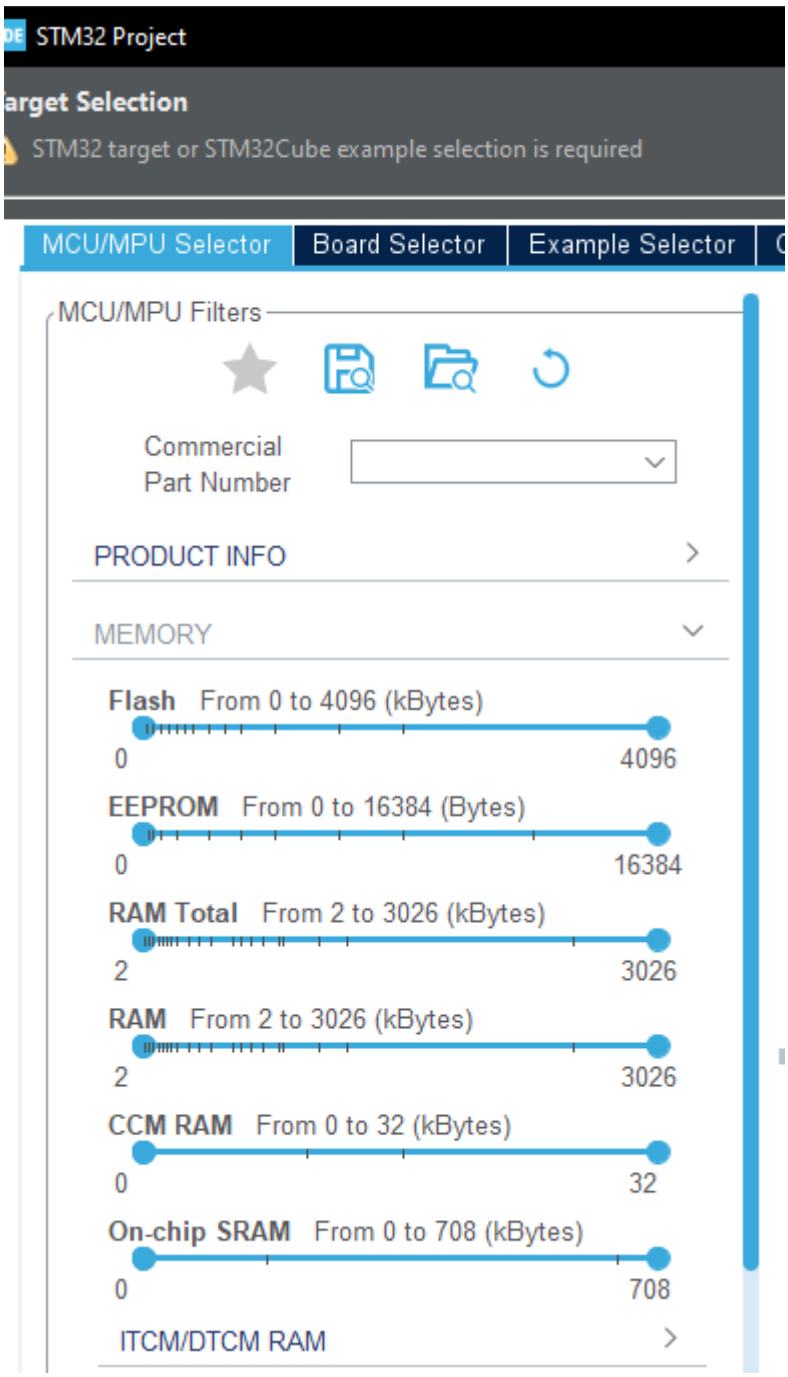
To oczywiście zależy od osobistych preferencji.

Na p[ocxzątek stworzymy pierwszy projekt:



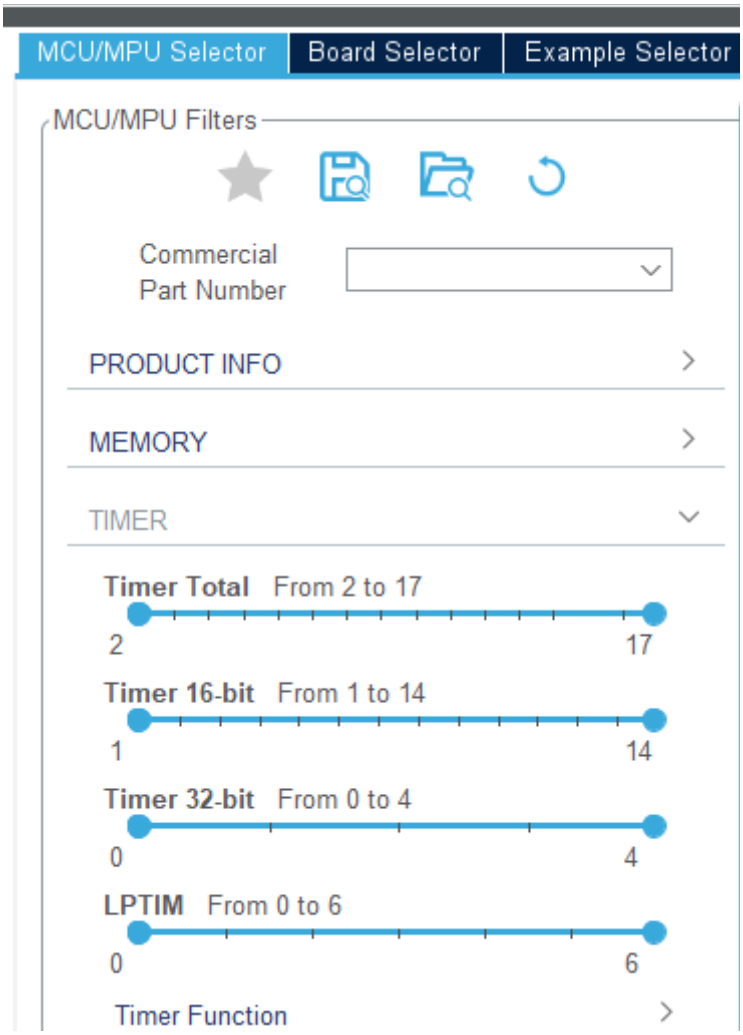
Rysunek 0005

Wyszukiwanie wybranego mikrokontrolera możemy zawęzić pod względem wielkości pamięci:



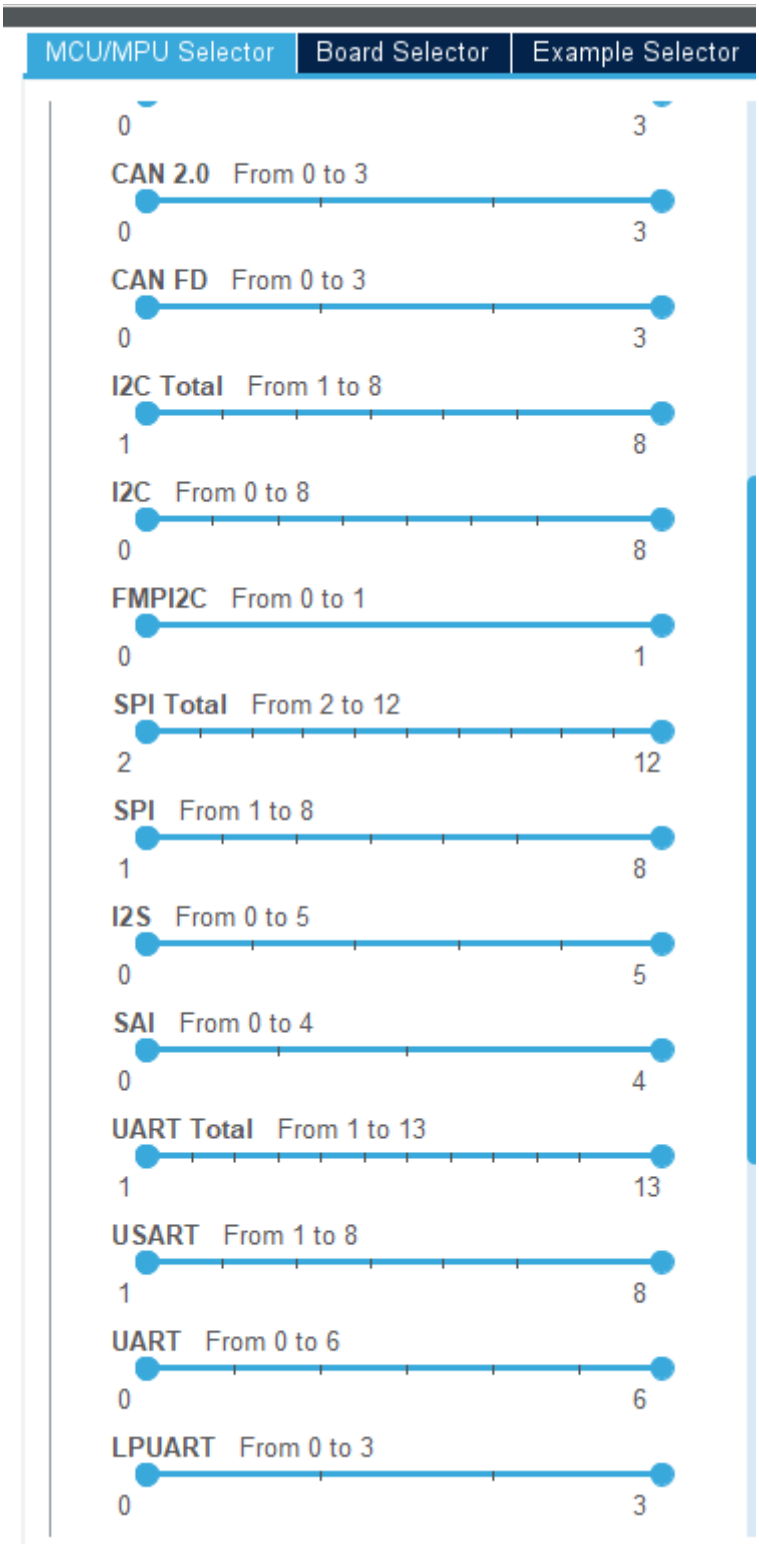
Rysunek 0006

Jak widać na rysunku widać, ze możemy wybrać wielkość pamięci EEPROM. W ten sposób legł w gruzach argument AVR-owców, jakoby ARM nie miały EEPROM i dlatego są gorsze od AVR. Po nadto EEPROM możemy mieć nawet 16kB o czym AVR-owcy mogą tylko pomarzyć (max 4kB). Wyszukiwać można też pod względem liczby timerów:



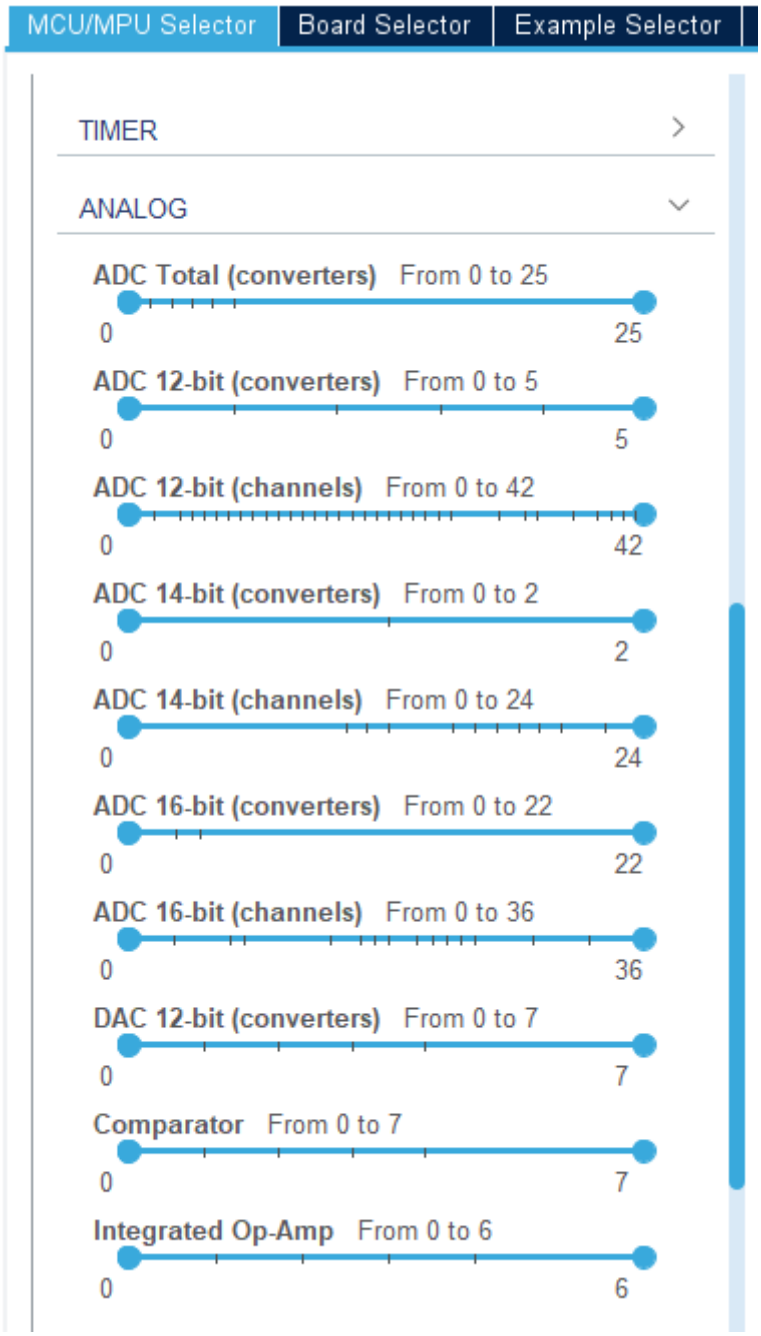
Rysunek 0007

a także innych peryferii jak CAN, SPI, I2C, UART, itd.:



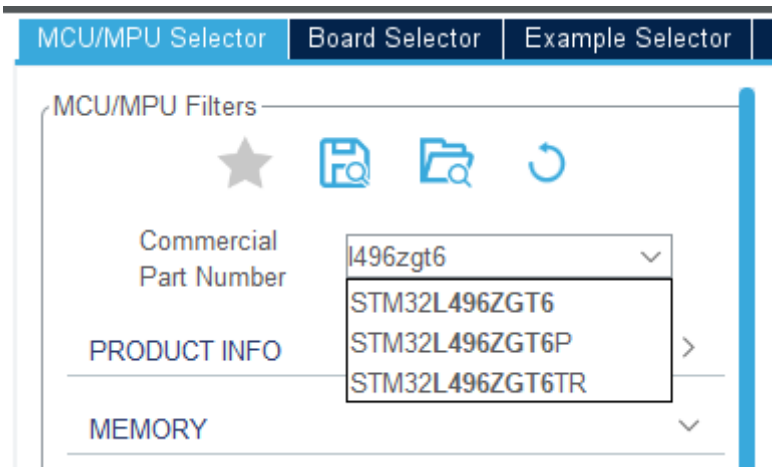
Rysunek 0008

Jeśli istnieje konieczność posiadania peryferii analogowych możemy zdecydować jakich i w jakiej konfiguracji:



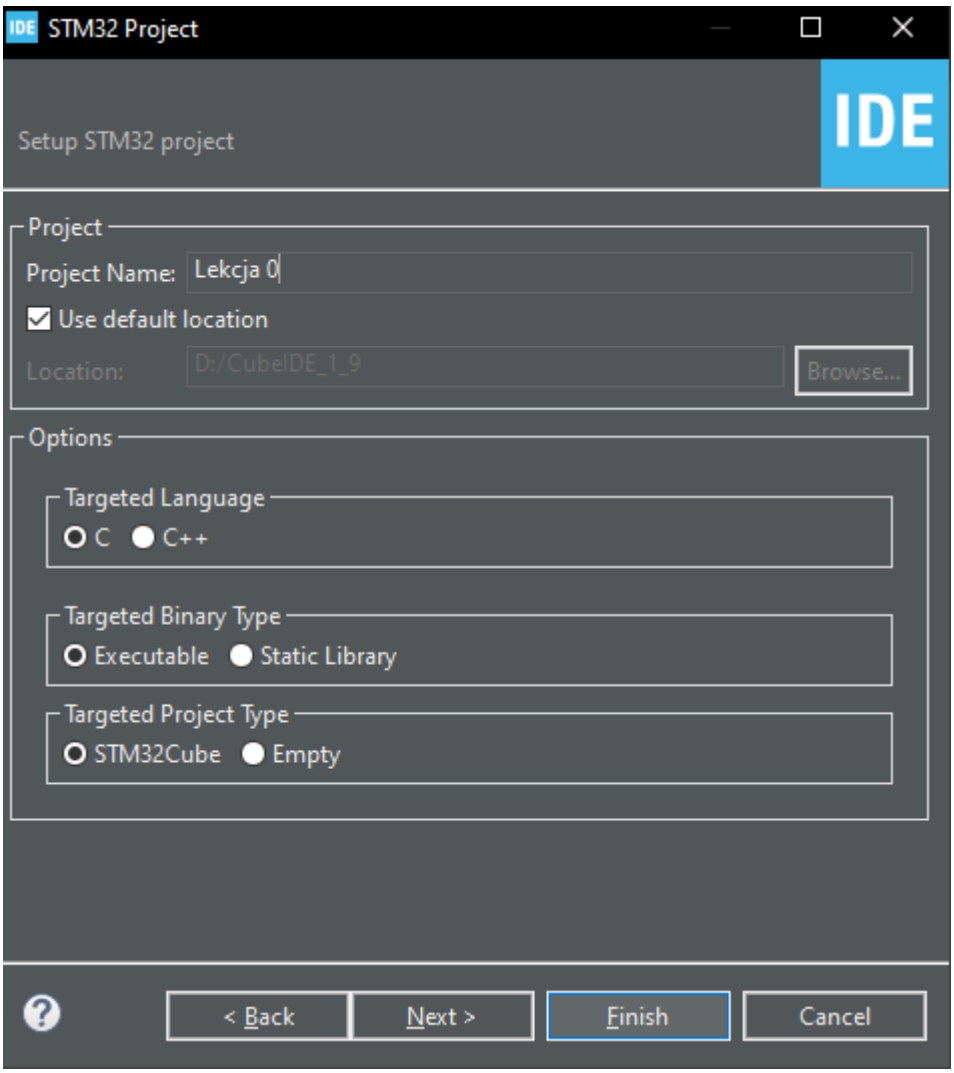
Rysunek 0009

Tworząc projekt można wybrać konkretny mikrokonmtroler:



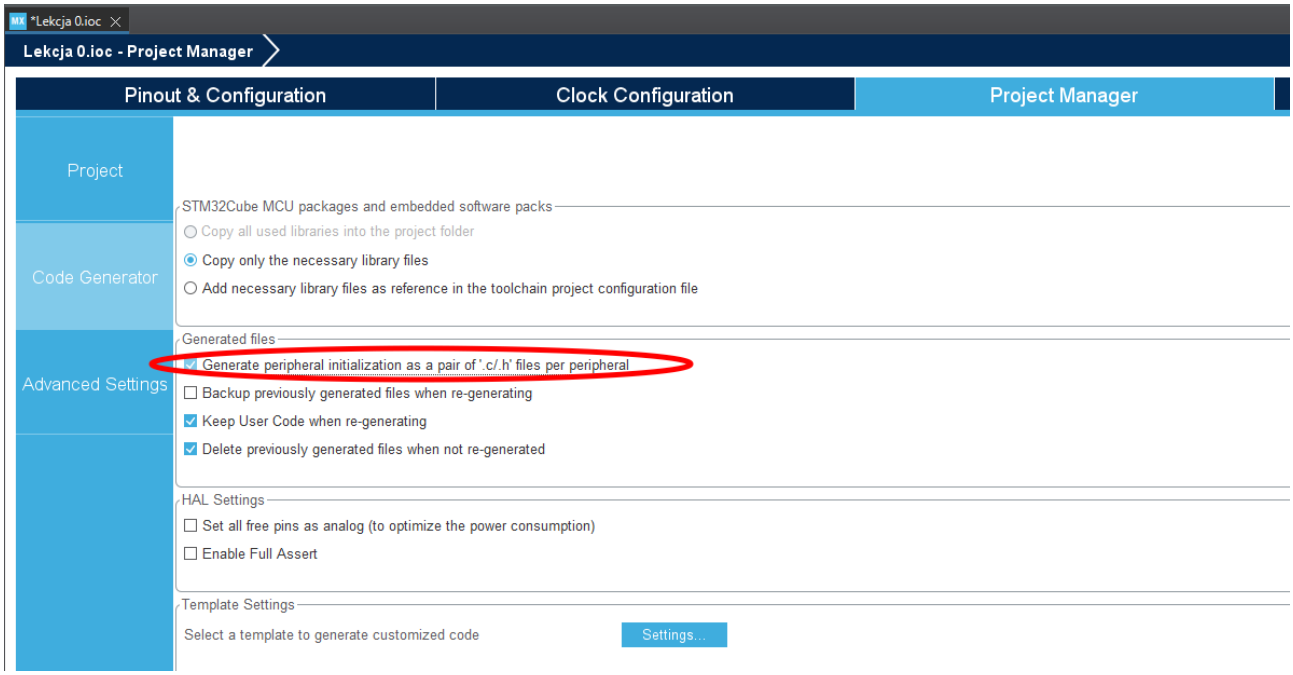
Rysunek 0010

Gdy wybraliśmy mikrokontolrt lub płytkę startową do projektu nazywamy projekt:



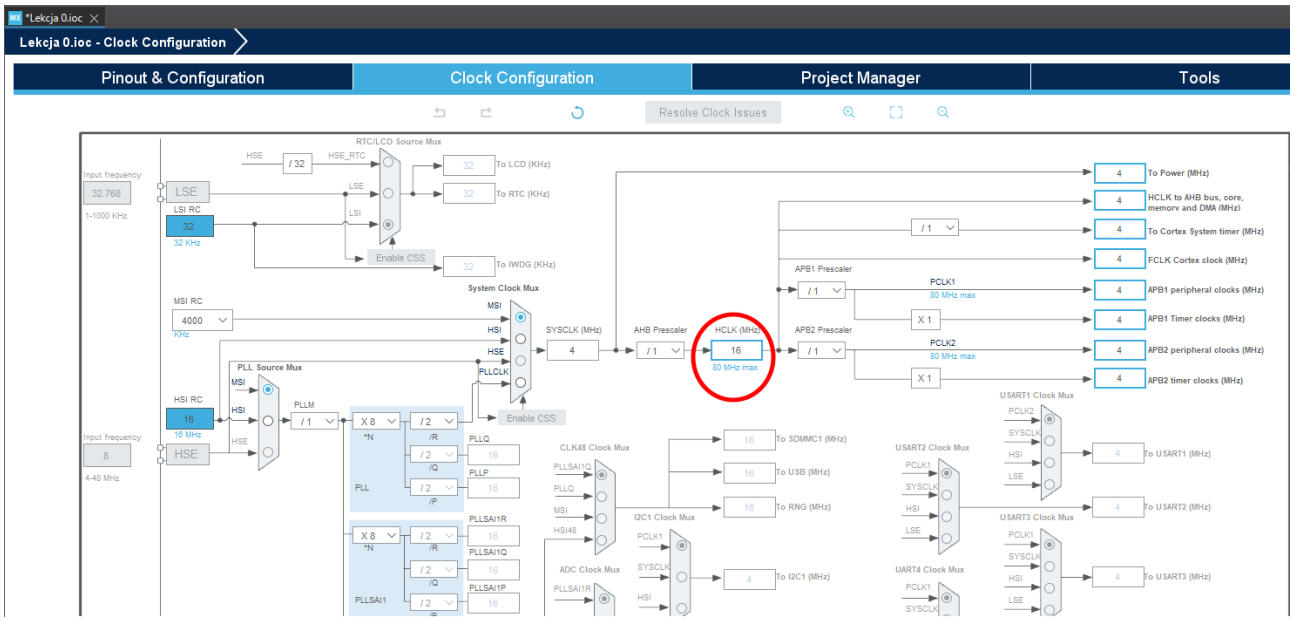
Rysunek 0011

W nazwie mogą być znaki spacji. W następnej kolejności polecam wybrać opcję „Generate peripheral initialization as a pair of .c/.h files per peripheral”



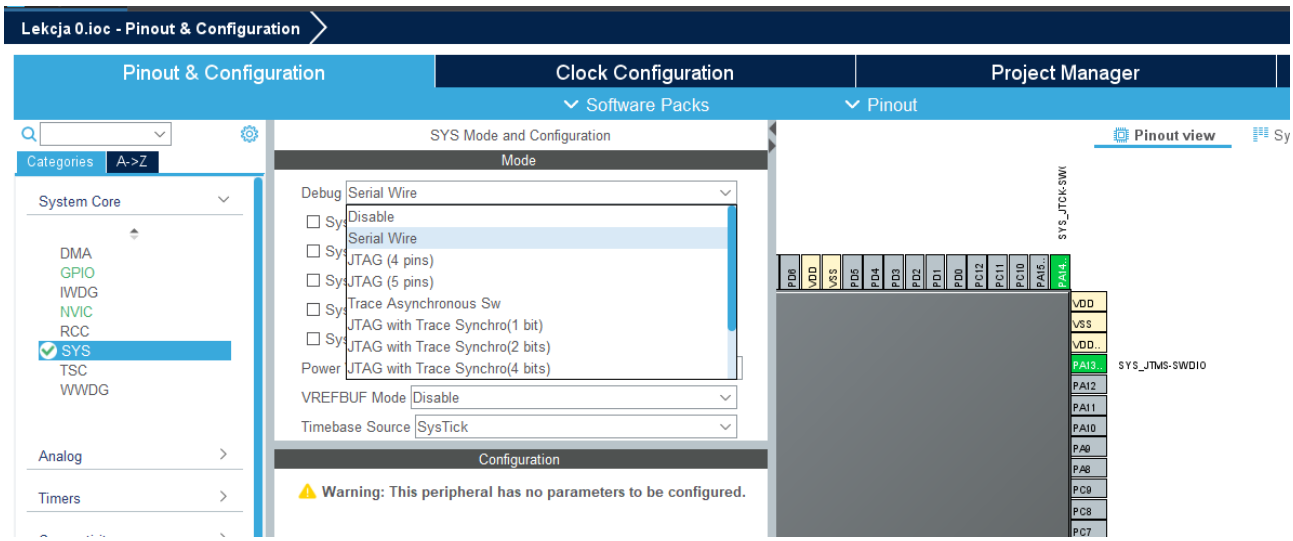
Rysunek 0012

W kolejnym kroku konfigurujemy zegary w tym najważniejszy taktujący CPU:



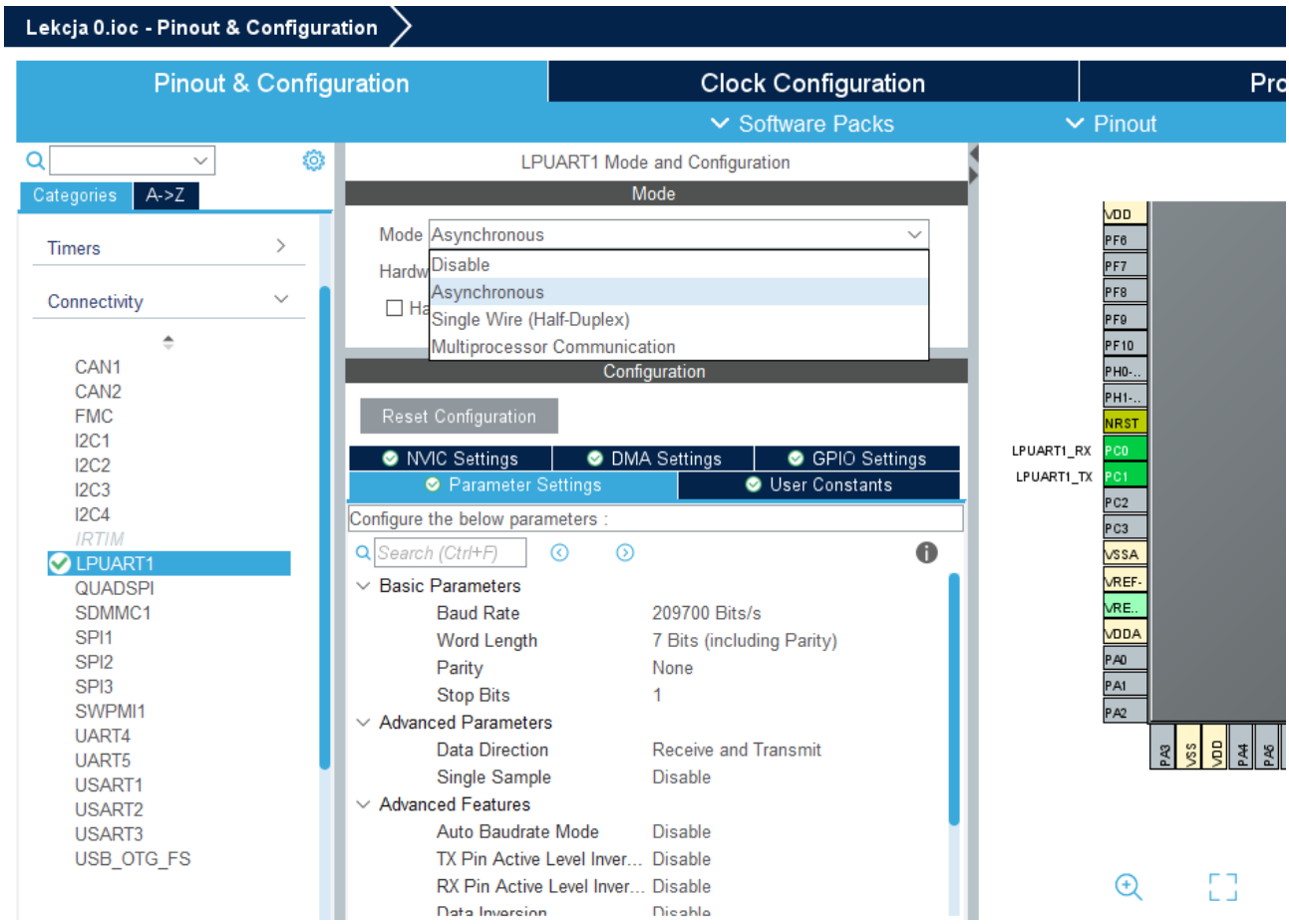
Rysunek 0013

Nie musimy „bawić się” w ustawianie podzielników i mnożników, wystarczy w okienku HCLK wpisać częstotliwość taktującą a CubeMX dobierze dla nas ustawienia. Jeżeli w selektorze wybraliśmy mikrokontroler a nie płytkę startową od STM, wskazane jest włączyć opcje debugowania:



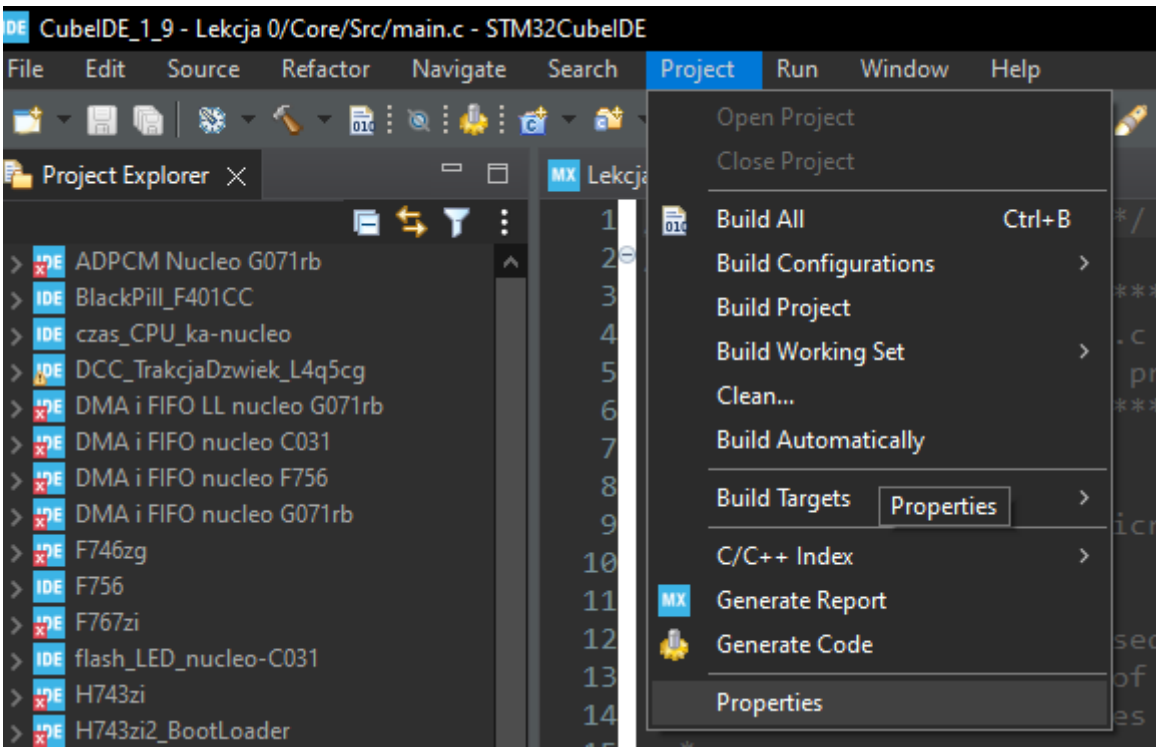
Rysunek 0014

Pozostaje skonfigurować używane peryferia dla przykładu UART:



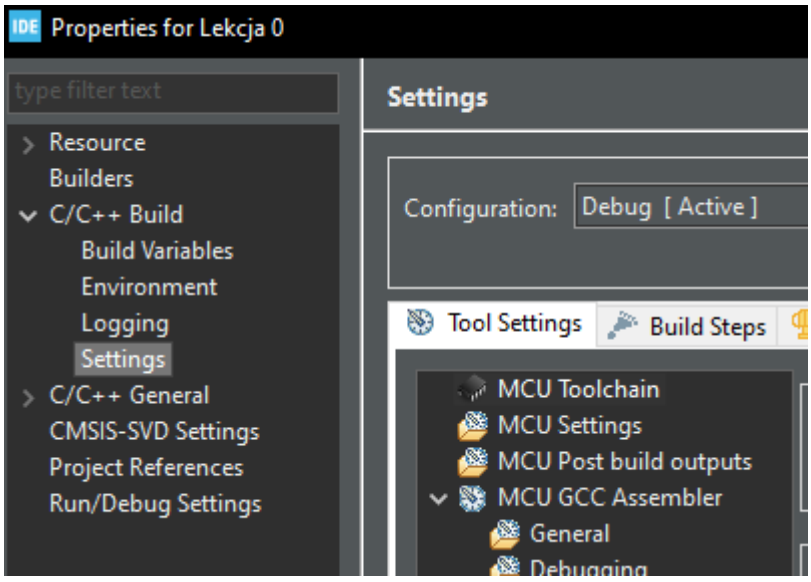
Rysunek 0015

Niestety czasami ustawienia domyślne nie są zbyt rozsądne. W UART prędkość i liczba bitów. W SPI liczba bitów. Po wygenerowaniu plików projektu warto zmienić konfigurację:

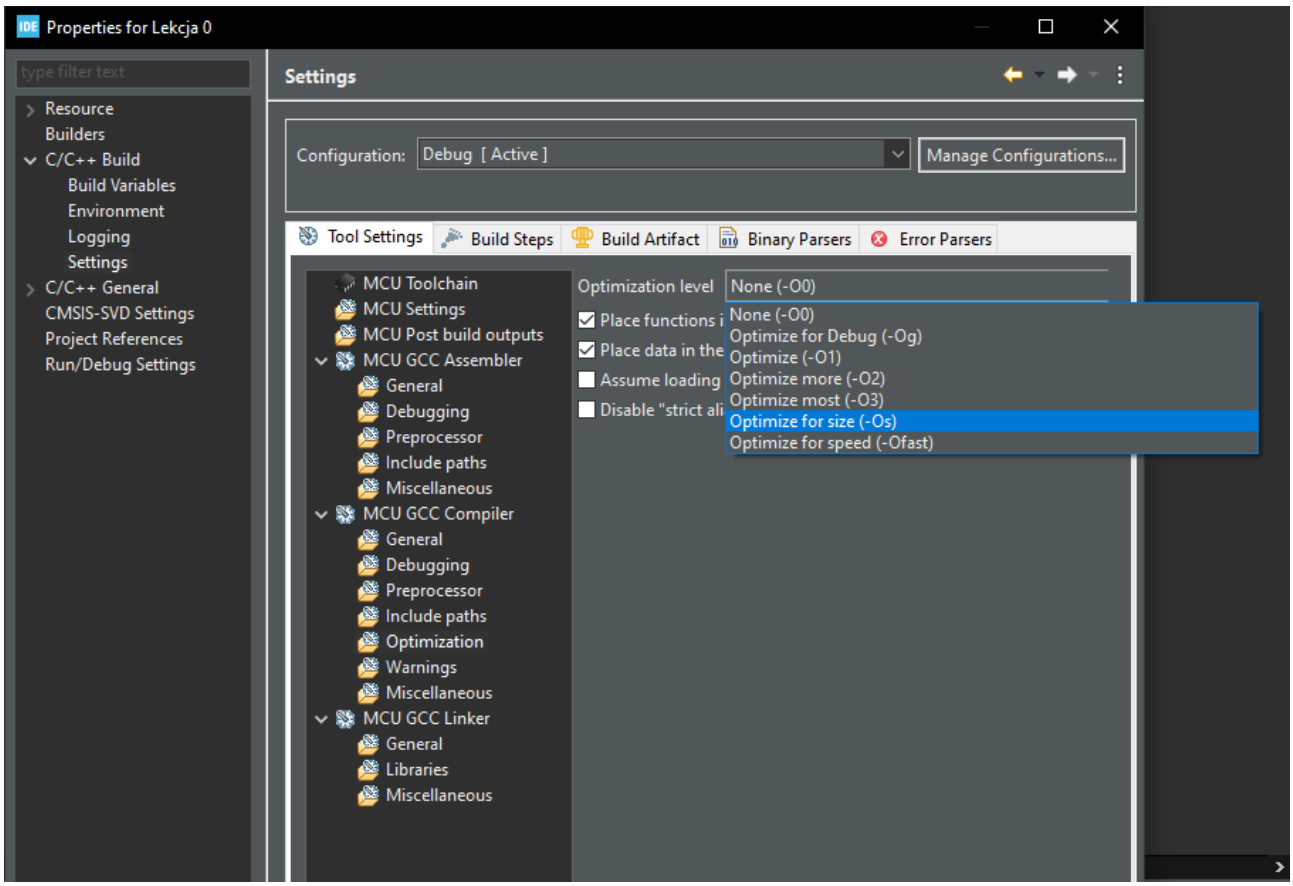


Rysunek 0016

bo niestety domyślnie optymalizacja jest wyłączona:



Rysunek 0017



Rysunek 0018

Film 0 o instalacji i konfiguracji CubeIDE dostępny jest na: https://youtu.be/72A69BSDGp0

Film 0 – NVIC o priorytetach i subpriorytetach przerwań: https://youtu.be/Vd9w-OHnOWE

Projekty, do których linki znajdują się w książce, należy rozpakować po czym zaimportować do CubeIDE. Jak to zrobić wyjaśniam w filmie: https://youtu.be/EJBJuJHd7_U

Wszystkie przykłady (op ile nie zaznaczono inaczej) zrealizowano w CubeIDE 1.6

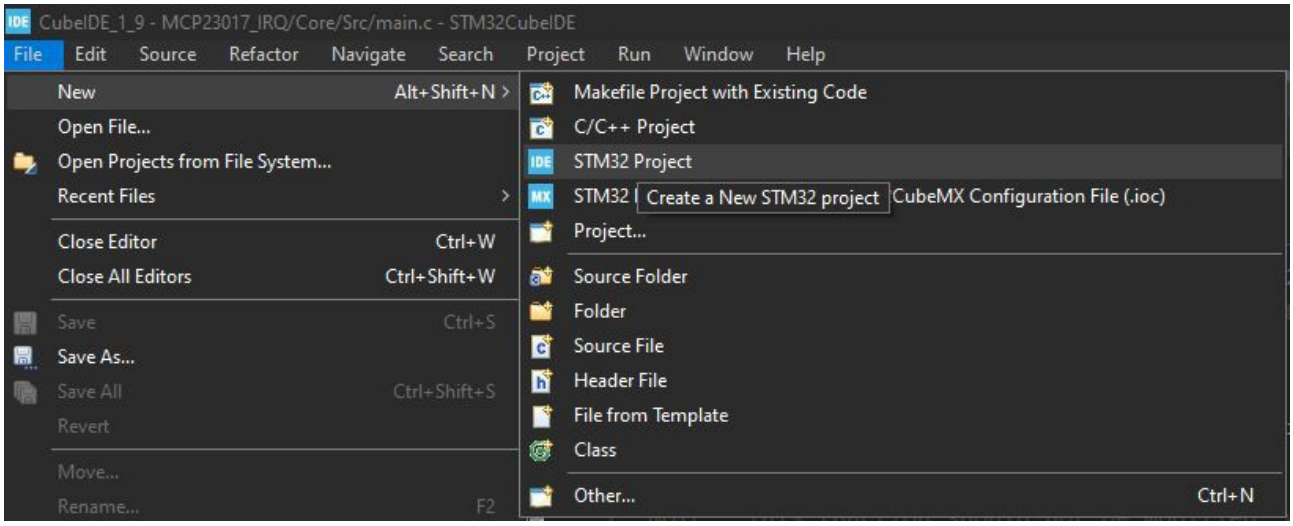
Programy dla Windows używane w książce: https://drive.google.com/Soft dla Win

Pliki danych (sample, grafiki): https://drive.google.com/Sample

Rozdział 1. Mikrokontrolerowe „Hello Word” czyli miganie LED'em na kilka sposobów

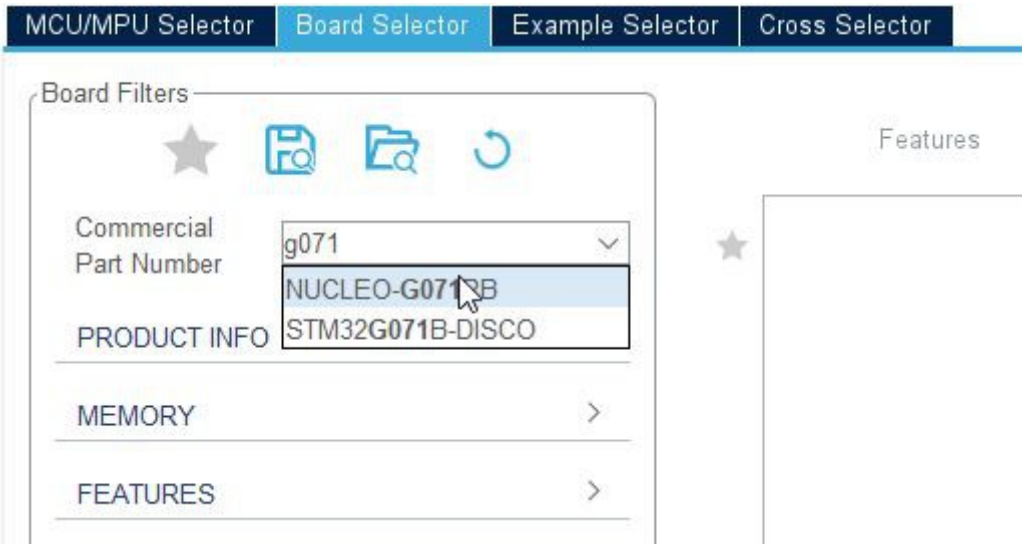
1.1 Miganie z Delay

Sposób najprostszy ale najgorszy. Po uruchomieniu CubeIDE wybieramy „File/New/STM32 Project” co ukazano na rysunku 0101.



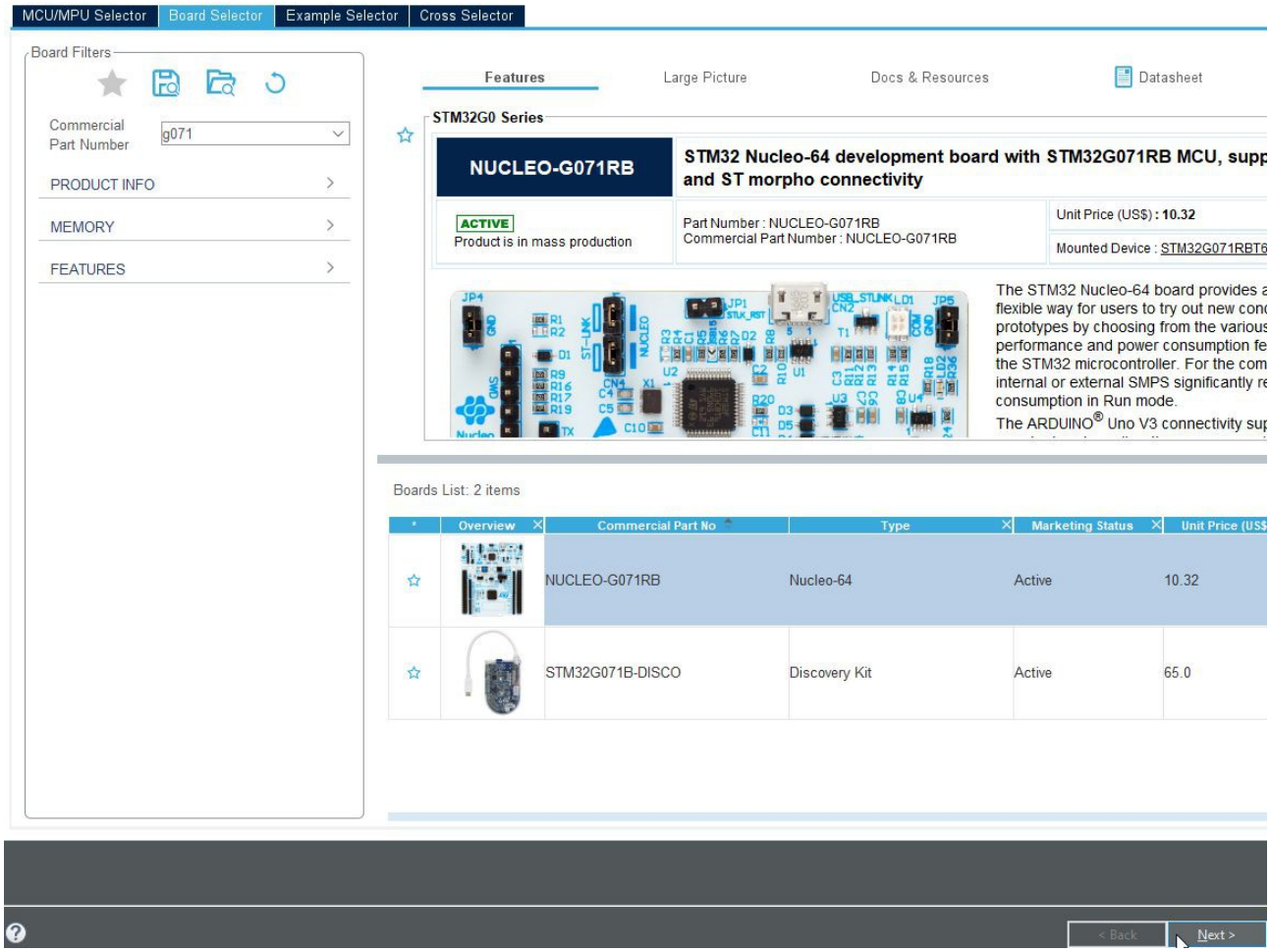
Rysunek 0101

CubeIDE może pobrać pliki z Internetu, po czym ukarze się się okno (rysunek 002), w którym w zakładce „Board Selectror” wybieramy nasza płytkę NUCLEO lub DISCOVERY, wpisując w „Commercial Part Number” cały lub część oznaczenia płytki (w przykładzie jest to „NUCLEO G071RB”.



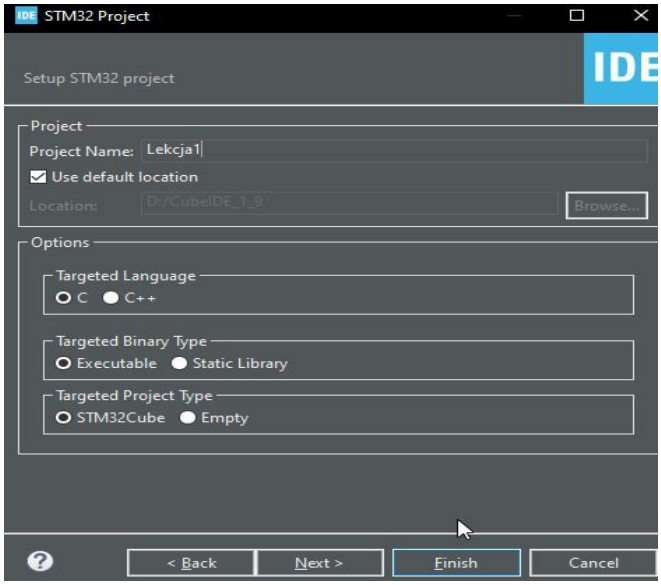
Rysunek 0102

Jeśli wpisaliśmy część symbolu może pokazać się wiele płytek (rysunek 003). Zaznaczamy naszą po czym naciskamy „Next”.



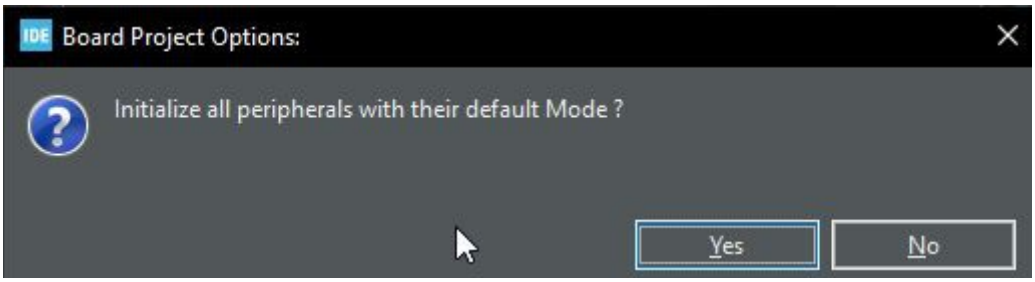
Rysunek 0103

Następnie pokaże się okno, w którym w”Project Name” wpisujemy nazwę naszego projektu.



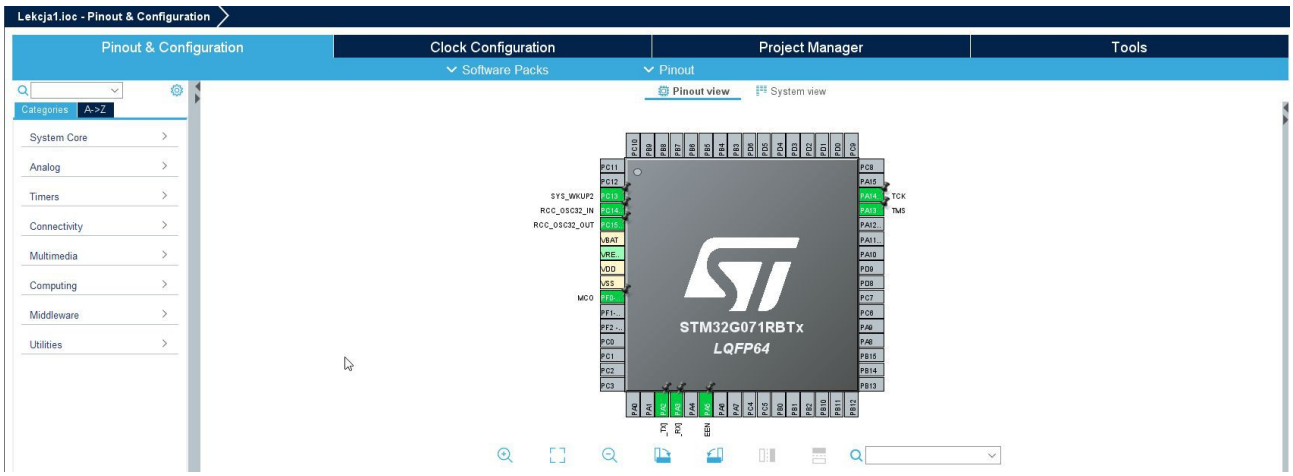
Rysunek 0104

Po naciśnięciu „Finish” pojawi się pytanie o konfigurację peryferii



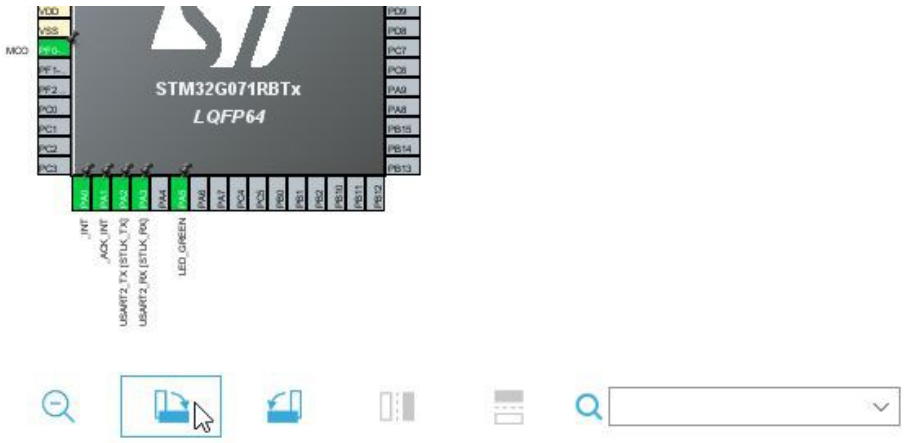
Rysunek 0105

Wybieramy „Yes”. Ukaże się okno



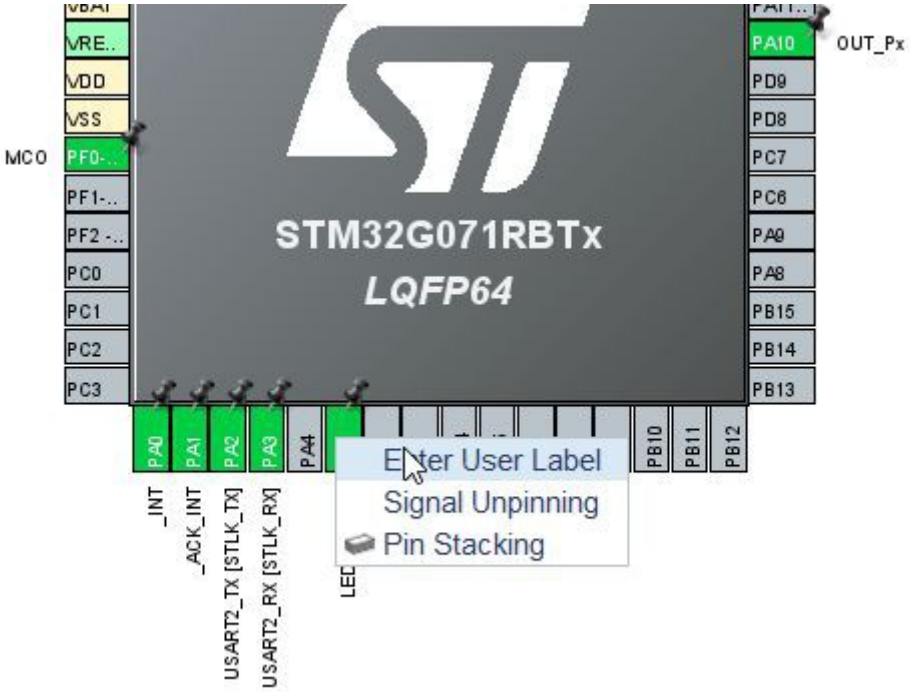
Rysunek 0106

, w którym można konfigurować peryferia ale do pierwszej lekcji nie jest to konieczne. Rysunek mikrokontrolera można powiększać rolką myszy trzymając wciśnięty klawisz CTRL i przesuwać myszą trzymając wciśnięty lewy klawisz myszy. W razie potrzeby ikonkami w dole okna



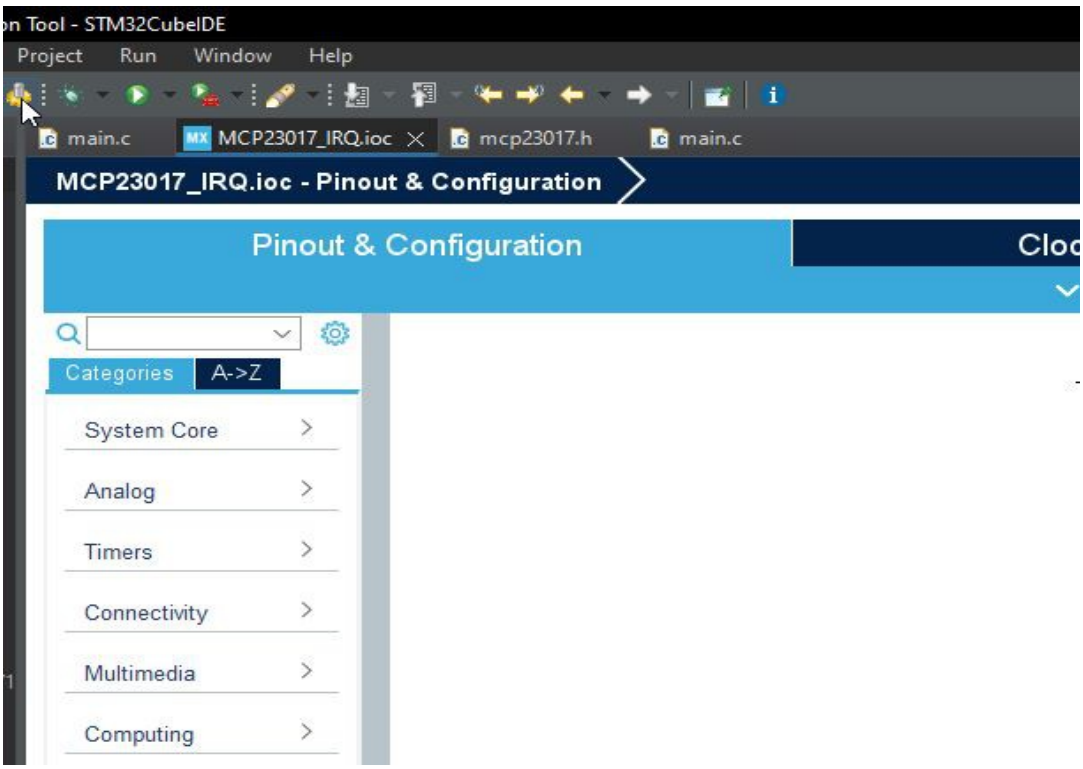
Rysunek 0107

Można rysunek obrócić. Okno z prawej strony z lupą umożliwia wyszukiwanie portów mikrokomntrolera po jego nazwie lub części. W przybliżeniu na rysunku 0107 widać, że zielona dioda jest podłączona do portu PA5 i opatrzona etykietą „LED_GREEN”. Nazwę etykiety można zmienić naciskając prawy klawisz myszy.



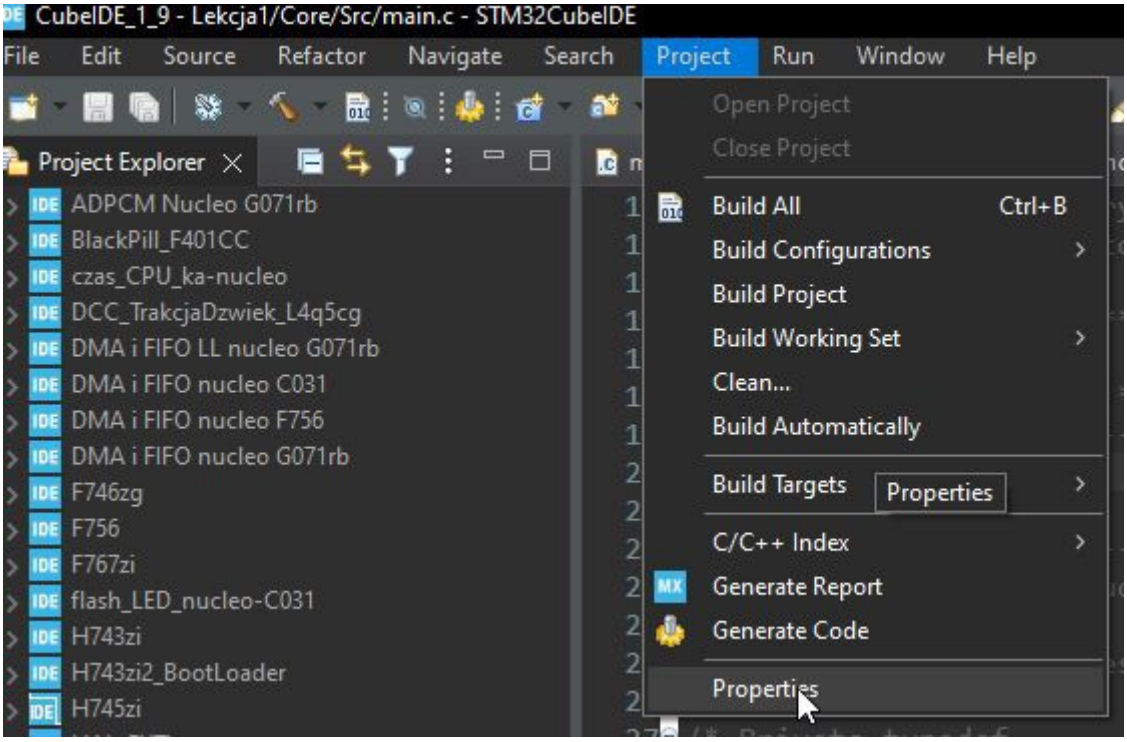
Rysunek 0108

Wygodne jest to, że w programie można posługiwać się zarówno nazwą poru (w tym wypadku PA5) jak i etykietą. Etykieta jest o tyle wygodna, że jeśli zmienimy fizycznie port, do którego podłączyliśmy LED to nie trzeba wprowadzać żadnych zmian w programie. Gdy już mamy skonfigurowane peryferia klikamy na „zębatkę”.



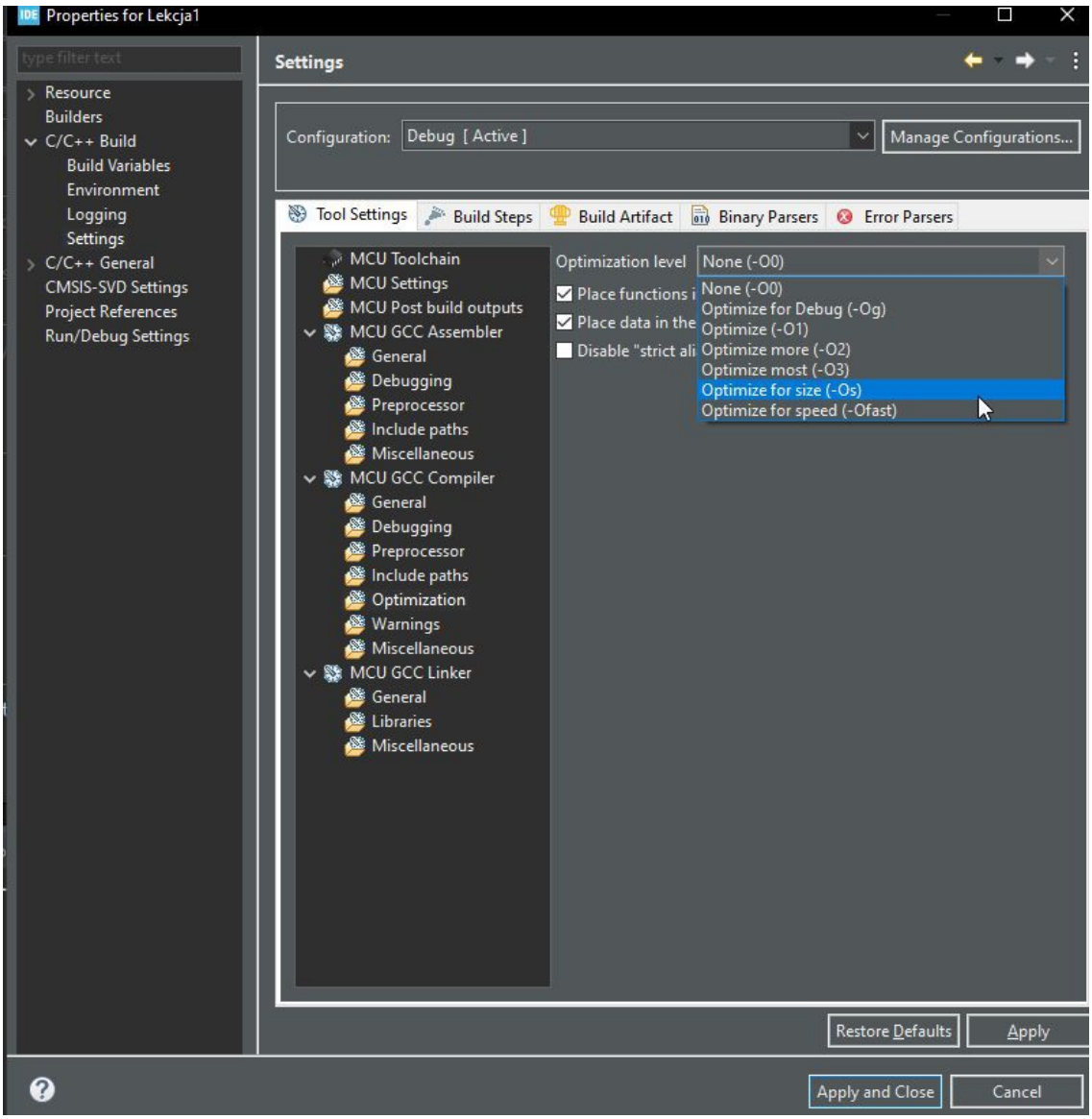
Rysunek 0109

Po ukazaniu się menu edytora wybieramy „Project/Properties”



Rysunek 0110

a w nowo otwartym oknie



Rysunek 0111

„Optimalization” po czym „Optymize for size (-os)”. Tysiąc słów nie zastąpi jednego filmu dlatego wszystkie czynności można zobaczyć na **Film1:** https://youtu.be/Ce70qvx2DU4

Film 1a – optymalizacja: https://youtu.be/RtgEDwLwqSE

Po tych czynnościach można rozpocząć pisanie programu. Jeśli nasz kod będzie zawarty pomiędzy komentarzami:

```
/* USER CODE BEGIN xxx */
a
/* USER CODE END xxx */
```

to ewentualne kolejne generowanie kodu przez konfigurator CubeMX (składowa CubeIDE) nie zniszczy tego co napisaliśmy. Kiedy może zajść konieczność ponownego generowania kodu? Gdy przykładowo zmienimy podłączenie peryferiów, będziemy chcieli dodać jakieś peryferia lub zmienić parametry ich pracy (np. szybkość UART). Dopiszmy więc po

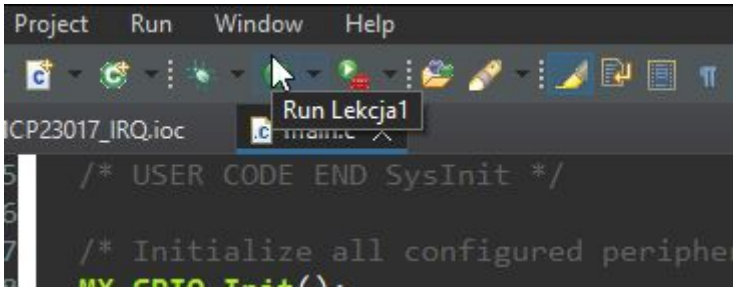
```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
```

Prosty kod migający diodą.

Listing 1a http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja01a.zip

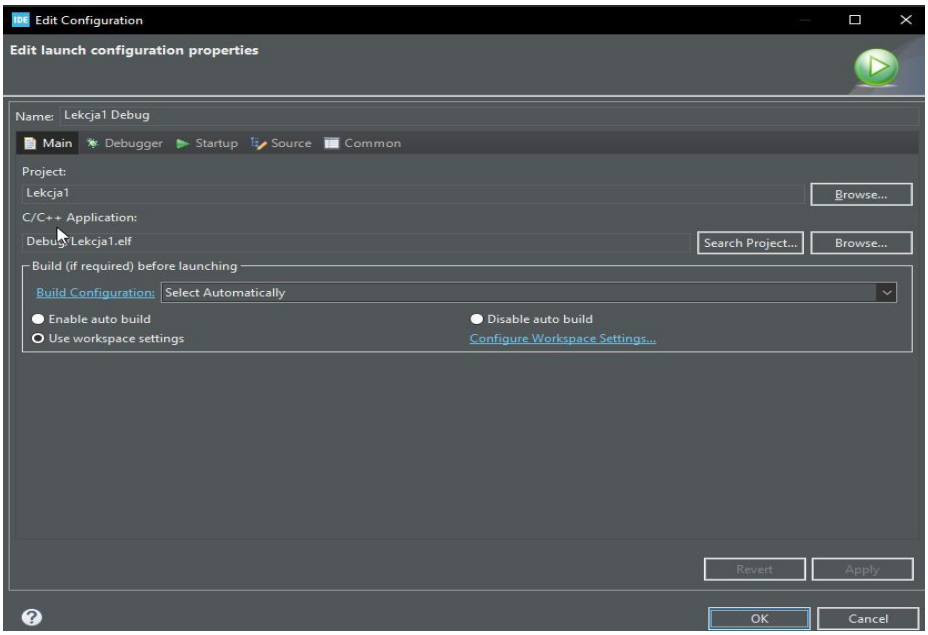
```
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

        HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
        HAL_Delay(500);
}
```

Kod jest „jakości” Arduino i pewnie stali widzowie mojego kanału na YouTube https://youtube.com/@eR-MIK zdziwią się, że pomijam użycie Watchdoga (WDG). Działanie nie jest przypadkowe, program ma być prosty a o WDG będzie później. Gdy kod jest gotowy klikamy na ikonkę „Run”



Rysunek 0112

Pierwsza kompilacja projektu po wygenerowaniu projektu przez CubeMX trwa dłużej ponieważ kompilowane są wszystkie pliki. Przy późniejszych zmianach trwa to krócej ponieważ kompilowane są tylko pliki, których dokonaliśmy zmian. Ponadto pierwsza kompilacja otworzy okno



Rysunek 0113

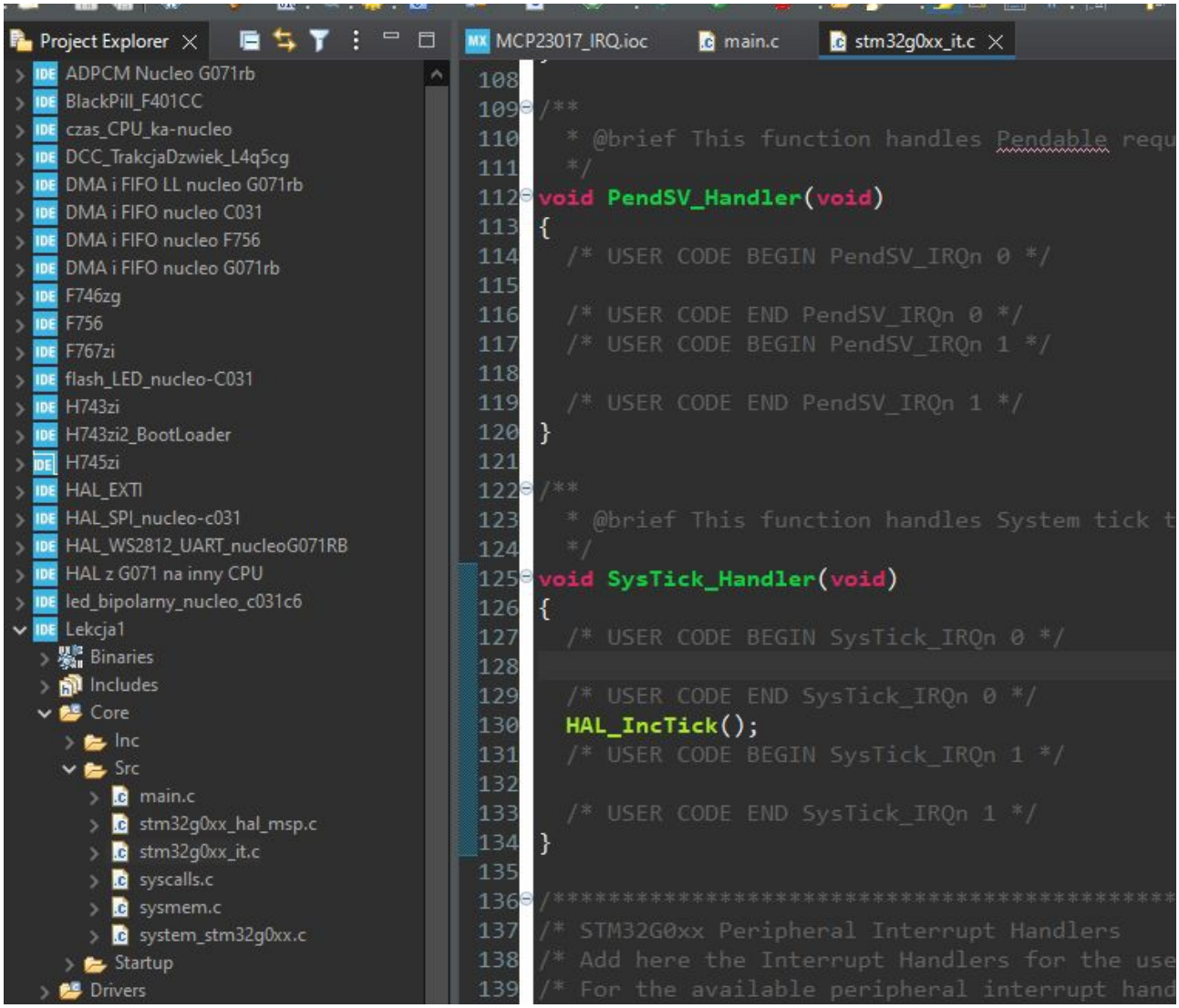
ustawień między innymi debugera. Jeśli do komputera mamy podłączony tylko jeden programator (płytkę NUCLEO/DISCOVERY) to nie trzeba nic zmienią, wystarczy kliknąć „OK”.

Po wgraniu programu dioda powinna migać. Można zmienić częstotliwość jej migania zmieniając parametr funkcji „HAL_Delay”. Jeśli mamy płytkę w większą ilością LED można zmienić diodę, którą steruje program zmieniając etykietę.

1.2 Miganie na przerwaniach systemowych

Pokazany wcześniej program jest prosty ale niedoskonały. W czasie migania Led'em nie można wykonywać innych czynności (poza tymi na przerwaniach) a CPU działa cały czas marnując energie i generując zakłócenia EMI. Jak go więc usprawnić? Można przenieść miganie LED'em na przerwania systemowe. W przeciwieństwie do AVR ARM mają w swoim rdzeniu timer (w AVR jest poza rdzeniem), jest prosty ale wystarcza do generowania przerwań systemowych. W przypadku AVR zaawansowany timer mogący pracować jako PWM, mierzący częstotliwość czy liczący czas sygnału zewnętrznego marnujemy do banalnego generowania przerwań co jedna milisekundę.

Jak „podpiąć” się pod przerwania w STM32? W Arduino jest to w zasadzie niemożliwe ale HAL STM32 przewidział taką opcję. Można to zrobić na kilka sposobów. Pierwszy



Rysunek 0114

to w „stm32g0xx.it.c” w funkcji „SysTick_Handler” w miejscu przeznaczonym na kod

użytkownika umieścić swój. Jest jednak sposób bardziej „systemowy”, który zadział na każdym STM32 bez konieczności „grzebania” w funkcjach przerwań. STM'owski HAL ma bardzo wiele funkcji z atrybutem „__weak” co umożliwia jej ponowne zadeklarowanie. W przypadku „HAL_IncTick” wygląda ona tak

Listing 1b **http:**//er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja01b.zip

```
__weak void HAL_IncTick(void)
{
  uwTick += (uint32_t)uwTickFreq;
}
```

Niewiele robi ale uwTick jest używane przez funkcje HAL dlatego tworząc swoją nie należy zapominać o tej zmiennej. Modyfikujemy więc nasz kod deklarując ponownie „HAL_IncTick()” za:

```
/* Private user code
---------------------------------------------------------*/

/* USER CODE BEGIN 0 */
```

Umieszczamy:

Listing 1b-2

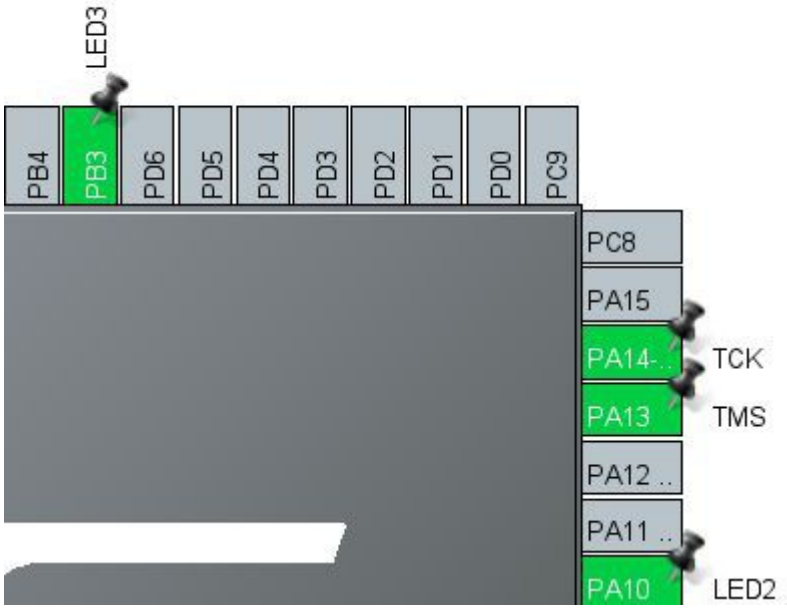
```
void HAL_IncTick(void){
	uwTick += (uint32_t)uwTickFreq;

	uint16_t static timLed;

	if( ++timLed >= 500 ){
		timLed = 0;
		HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
	}
}
```

Generalnie lepiej odliczać w dół ale o tym później. W pętli głównej usuwamy sterowanie diodą i delay. Kod oczywiście nie jest optymalny ale wiedza o przerwaniach systemowych będzie potrzebna w kolejnych programach. Zapraszam do obejrzenia filmu wyjaśniającego poruszaną tu tematykę. **Film 1b** https://youtu.be/yFkO5kqUMNc

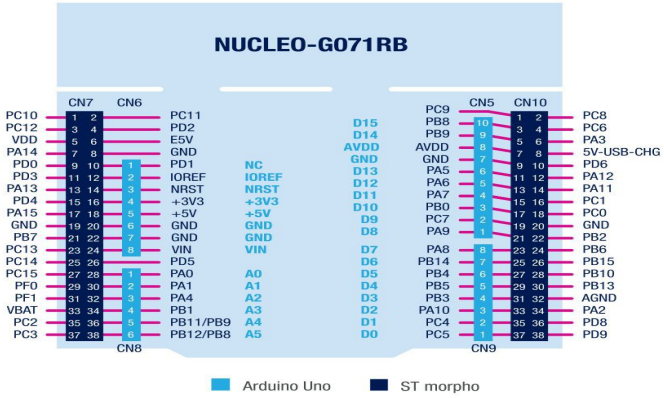
1.3 Miganie kilkoma LED z wykorzystaniem wirtualnych timerów

Używając delay nie ma prostego sposobu aby zamigać kilkoma diodami z różną częstotliwością. Znajdą się tacy, którzy zaproponują jakiś RTOS. Na 90% będą to osoby nie znające się na RTOS'ie i programowaniu mikrokontrolerów. Tu pozwolę sobie przytoczyć pewna konwersację. Otóż jeden z moich widzów chciał zrealizować prosty (dla mnie) projekt. Chwalił się jakie to ma osiągnięcia w programowaniu systemów (Linux i Windows). Próbowałem mu wytłumaczyć, że mikrokomntroler to coś innego ale nie słuchał. Po dwóch tygodniach”walki” przyznał, że nie daje rady mimo tego że ma „tony” papierów, które zdobył w szkole. Aby używać RTOS trzeba dobrze znać zagadnienia programowania, kompilatorów, przerwań, DMA i wiele wiele innych. Z pewnością nie jest to rozwiązanie dla początkujących w systemach wielozadaniowych. Na Arduino pozornie równolegle wykonywanie wielu zadań robi się używając „millis()” https://youtu.be/vSCuMQcjing

ale to nieoptymalny sposób zwłaszcza na AVR. W STM32 można by skorzystać z takiego rozwiązania zamieniając „millis()” na „HAL_GetTik()” ale nie jest to optymalne nawet na STM32 zwłaszcza tych z małą ilością pamięci RAM. Mało RAM na STM32 to 8kB na AVR 8kB to bardzo dużo - największy ma 16KB a STM32 1MB :-), mimo że AVR z tą samą ilością RAM i mniejszymi innymi zasobami jest droższy :-). Jak już wspominałem, lepiej liczyć w dół dlatego wypróbujemy kolejny kod z listingu 2.
Tworząc projekt postępujemy jak w poprzednim przykładzie ale w chwili ukazania się okna konfiguracji peryferiów ustawiam,y PA10 i PB3 w roli wyjście nadając im etykiety odpowiednio „LED2” i „LED3”.



Rysunek 2001
Szczegóły wyjaśnia **film 2** https://youtu.be/dIYk0PCbam8
zastępujący dziesiątki czy setki obrazów a tym bardziej ogromne ilości tekstu.
Dlaczego te a nie inne porty? Dlatego, że są dostępne na złączu „Arduino”! Skąd wiadomo, że akurat na tych pinach? Z noty katalogowej „UM2953” dostępnej na stronie producenta. Strona 19 wszystko wyjaśnia.



Rysunek 0202

Nie będę pokazywał co robić po wybraniu peryferiów, bo moim celem nie jest generowanie niepotrzebnych stron jak w wiodącym teraz wydawnictwie o tematyce elektronicznej. Wystarczy zapoznać się z wcześniejszą treścią książki i powtórzyć poprzednie czynności. Trochę „pogubiła" się chronologia ale myślę, że nie jest to problemem. Wracamy do programu. Dodajmy kilka wpisów:

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


            /* USER CODE BEGIN 3 */
            if( ! timLed1 ){
                  timLed1 = 500;
                  HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
            }
            if( ! timLed2 ){
                  timLed2 = 300;
                  HAL_GPIO_TogglePin(LED2_GPIO_Port, LED2_Pin);
            }
            if( ! timLed3 ){
                  timLed3 = 100;
                  HAL_GPIO_TogglePin(LED3_GPIO_Port, LED3_Pin);
            }


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

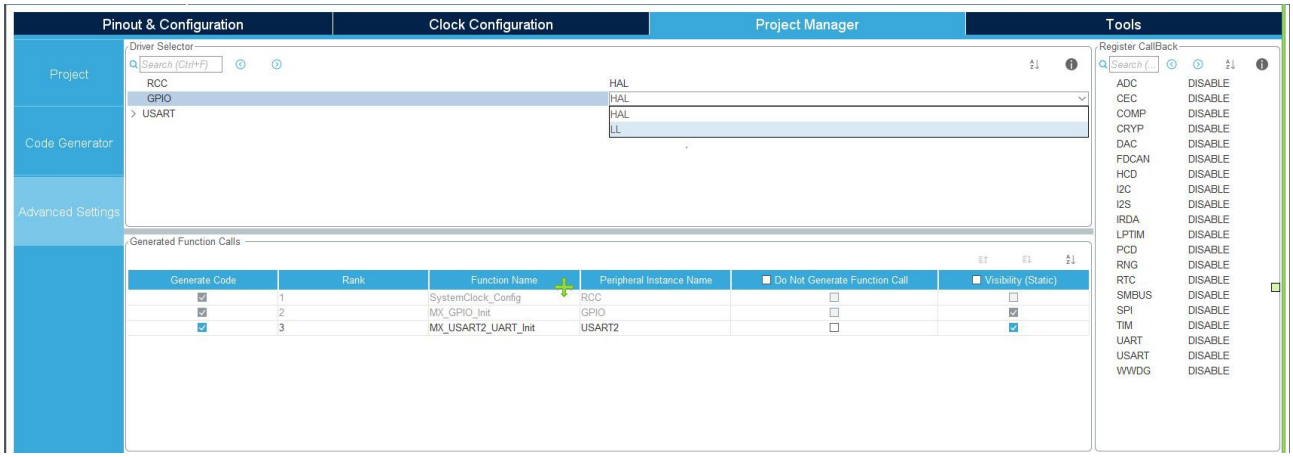
Listing 2 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja02.zip

Program działa podobnie jak ten dla Arduino z wykorzystaniem „millis()" ale używa wirtualnych timerów. Jaką to daje korzyści? Zamiast zużywać **ZAWSZE** czterech bajtów do zapamiętania czasu używamy, zależnie od potrzeby 1, 2 lub dla bardzo długich czasów (ponad 65 sekund) 4 bajty. To może dać dużą oszczędności pamięci RAM (w systemach 64-bit **nawet osiem bajtów** na jeden licznik) i oszczędność czasu CPU zwłaszcza w systemach 8-bit (AVR, 8051, Z80) oraz 16-bit (Intel 286, 80251, itp) ale w 32-bit i większych niekoniecznie. Tu należałoby wspomnieć o zmiennych z „fast' ale chyba jeszcze nie pora aby o tym pisać.

1.4 Miganie Led'em z wykorzystaniem bibliotek LL

Kodu pisany z wykorzystaniem bibliotek LL jest tak wydajny jak ten pisany „na rejestrach" i trochę go przypomina. Zaletą LL jest przenośność kodu pomiędzy rodzinami mikrokontrolerów czego nie można powiedzieć o pisaniu „na rejestrach".

Włączenie bibliotek LL w CubeMX dokonujemy w zakładce „Project Menager/Adwaced Setings" co pokazuje rysunek 0301.



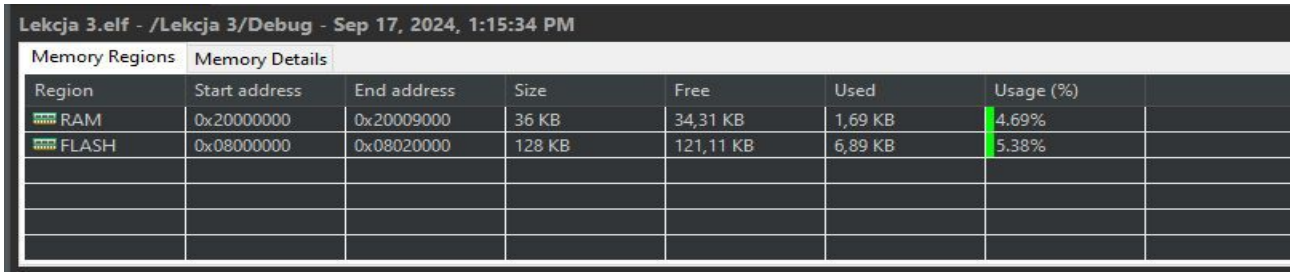
Rysunek 0301

Po wygenerowaniu kodu przystępujemy do pisania kodu

Listing 3a http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja03a.zip

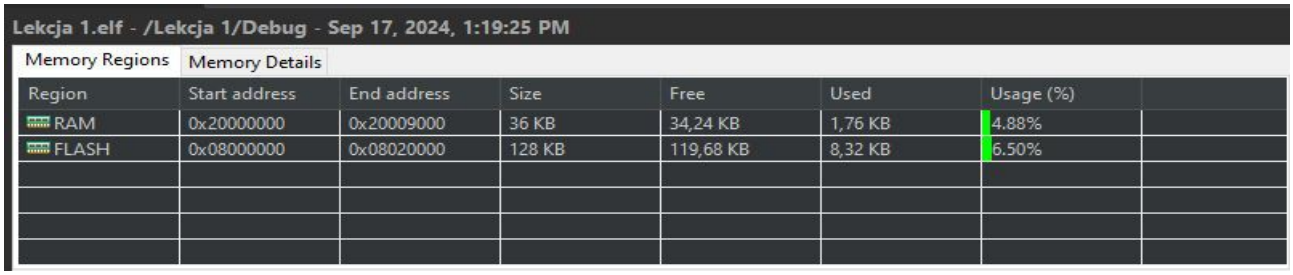
```
  /* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
      LL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
      LL_mDelay(500);
}
/* USER CODE END 3 */
```

kod po skompilowaniu zajmuje niecałe 7kB FLASH i 1,7kB RAM.



Rysunek 0302

Dla porównania w wykorzystaniem HAL 8,3kB i 1,8kB



Rysunek 0303

Kod w wykorzystaniem LL będzie jeszcze mniejszy jak włączymy LL dla UART. W takim przypadku potrzeba 2,14KB FLASH i 1,53kb RAM. W kolejnym kroku skorzystamy z przerwań systemowych.

Listing 3b http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja03b.zip

Jak napisać powyższy kod można zobaczyć w **filmie 3** https://youtu.be/SaK_oEQ2YcM

```
/* USER CODE BEGIN 0 */

void initIrqTimSys(){
	LL_SYSTICK_EnableIT();	// @paplociennik
	//LL_Init1msTick( LL_RCC_GetSystemClocksFreq() );

	//LL_Init1msTick( LL_RCC_GetSystemClocksFreq() );

	/*
	//SysTick_Config(SystemCoreClock/1000);
	//SysTick_Config(SysTick->LOAD);
	SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;
	NVIC_EnableIRQ(SysTick_IRQn);
	 */
}


void HAL_IncTick(void){
	uwTick += (uint32_t)uwTickFreq;
	uint16_t static timLed;


	if( ++timLed >= 500 ){
		timLed = 0;
		LL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
	}
}




	/* USER CODE BEGIN 2 */
	initIrqTimSys();

	/* USER CODE END 2 */




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

		/* USER CODE BEGIN 3 */
		//	LL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
		//	LL_mDelay(500);
	}
	/* USER CODE END 3 */
```

O bibliotekach LL ich zaletach i wadach mówiłem w kursie na YouTube:

https://www.youtube.com/playlist?list=PLdtkbzWTUVMkX_pBDn6tHqxY0u_d_kZ07

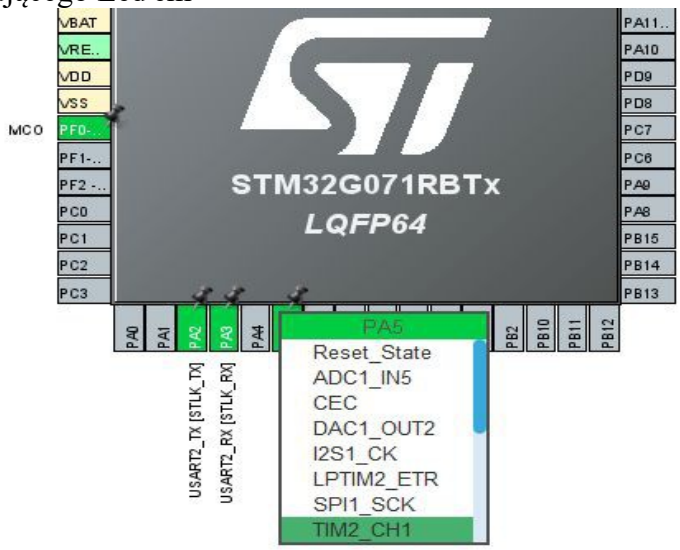
1.5 Miganie z wykorzystaniem sprzętowego TIMER'a

Temat timerów będzie omówiony później a tu tylko o tym wspomnę aby było zrozumiałe jak to działa z użyciem DMA.

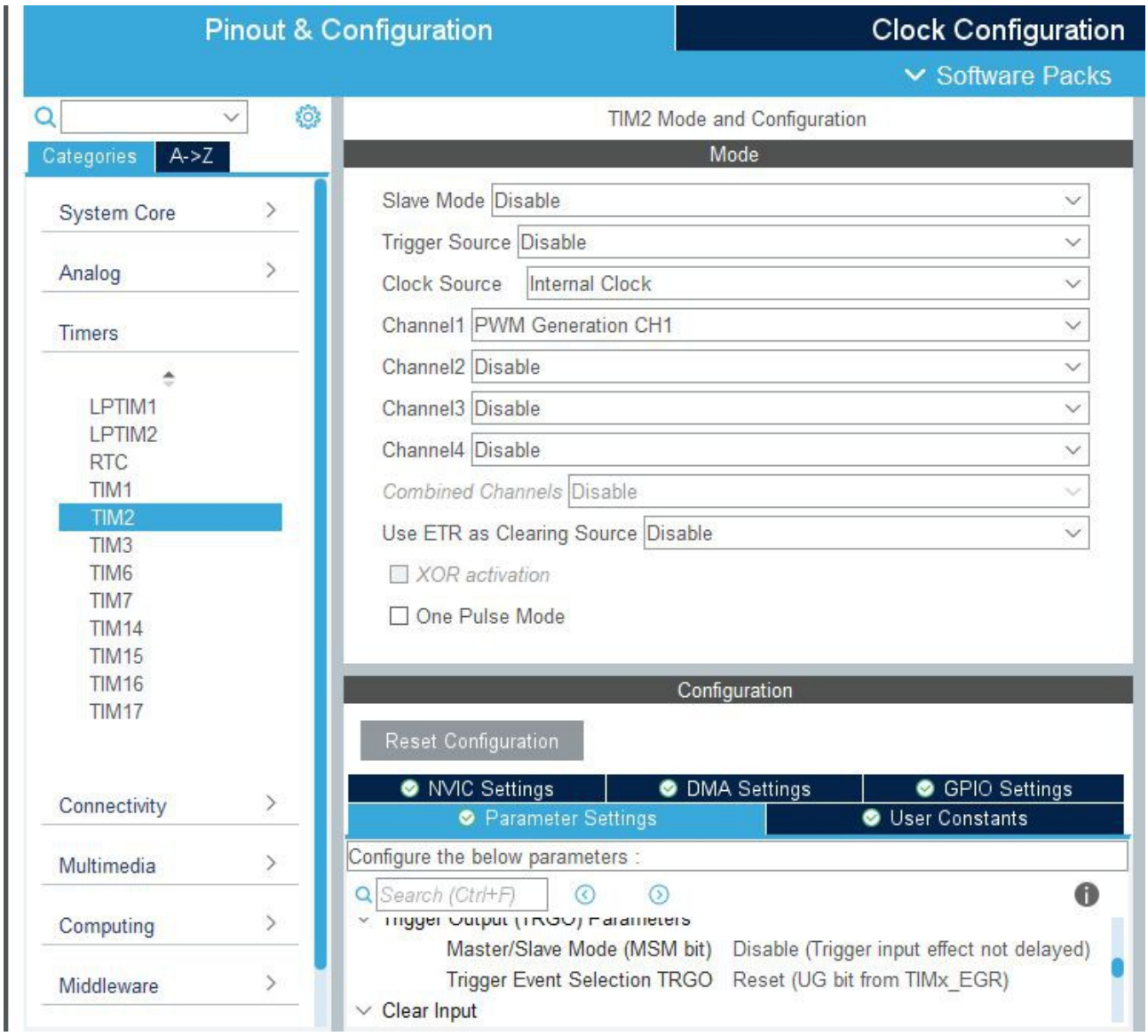
Kod programu jest banalny
Listing 4 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja04.zip

```
	HAL_TIM_Base_Start(&htim2);
	HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);

	/* USER CODE END 2 */

	/* Infinite loop */
	/* USER CODE BEGIN WHILE */
	while (1)
	{
		/* USER CODE END WHILE */

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

Startujemy TIMER oraz PWM i to wszystko. Konfiguracja CubeMX jest jednak bardziej skomplikowana. W pierwszym kroku sprawdzamy lewym klawiszem myszy czy i jaki timer jest podłączony do wyjścia sterującego Led'em

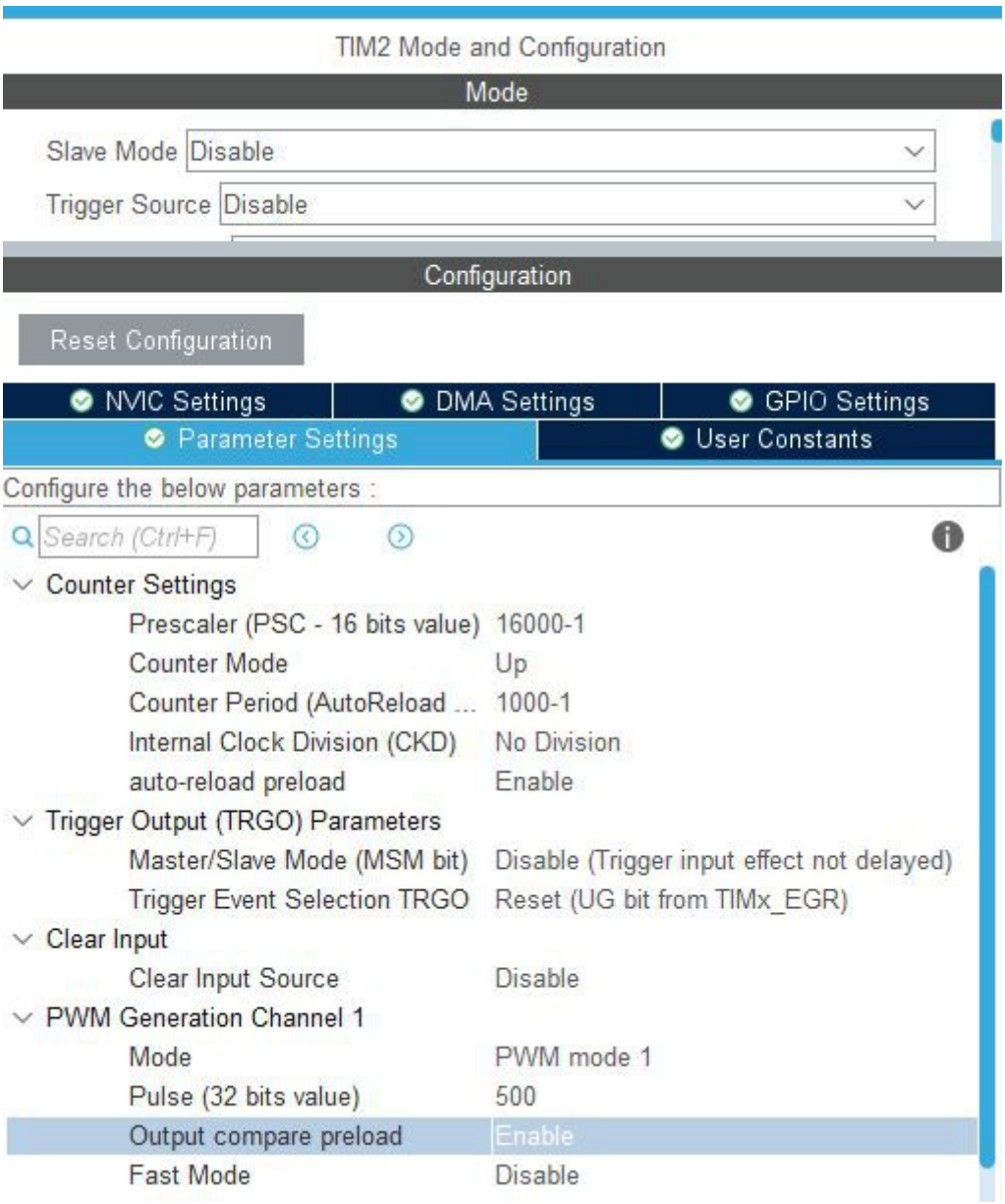


Rysunek 0401
Na pinie należy ustawić „TIM2_CH1” lub „Reset_State”, w przeciwnym wypadku nie będzie można poprawnie skonfigurować timera. Timer ustawiamy według rysunku



Rysunek 0402

W menu po lewej wybieramy „TIM2”. W oknie po prawej „Clock Source” na „Internal Clock” a w „Channel1” wskazujemy „PWM Generator Ch1”. W oknie konfiguracji



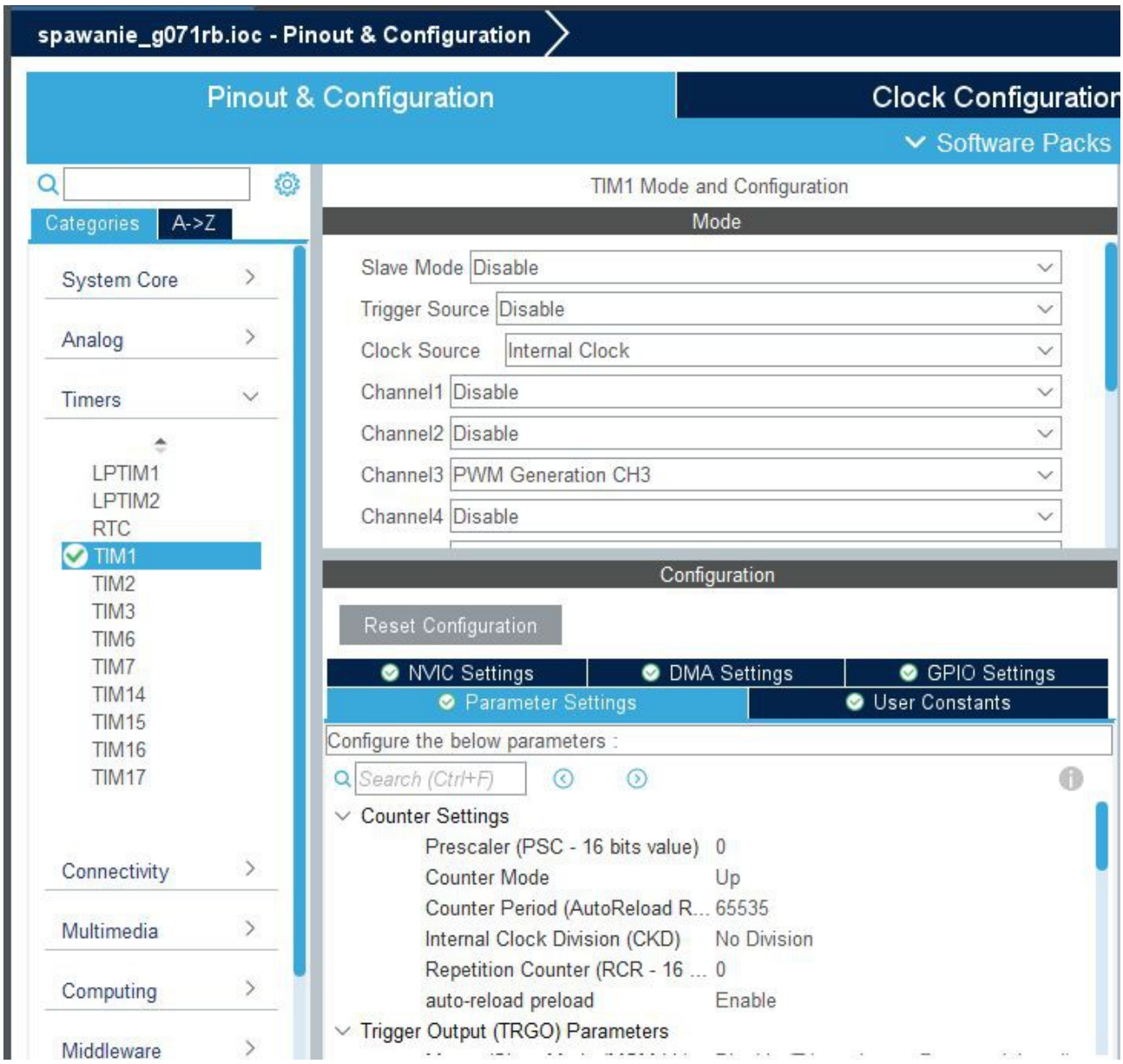
Rysunek 0403

Prescaler ustawiamy na 15999 (16000-1) co podzieli zegar CPU 16MHz przez 16000 co da 1kHz, „Couneter Periad" na 999 (1000-1) co wygeneruje sygnał 1Hz, „auto-reload" ustawiamy na „Enable". W sekcji „PWM Generation Cannel 1" „Pulse' ustawiamy na 500. Spowoduje to miganie diody 500/500ms. **Film 4** https://youtu.be/df_ZIJB07xM rozwieje wszelkie wątpliwości.

1.6 Miganie przez DMA

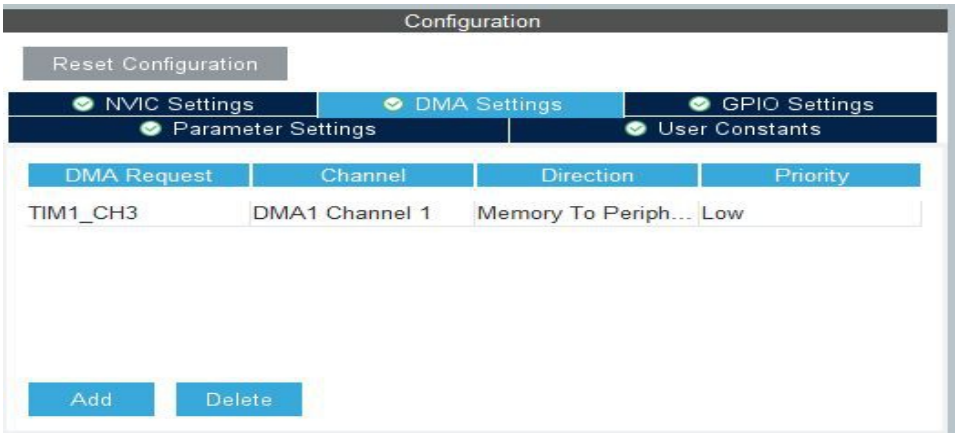
(Efekt spawania, burzy, płynne rozjaśnianie i wygaszanie LED)

W tej lekcji pokażę coś co jest niewykonalne na AVR czyli jak migać LED'em praktycznie bez angażowania CPU. W wielu sytuacjach można to zrobić całkowicie bez udziału CPU. W tym celu należy wykorzystać DMA. Do sterowania Led'em użyjemy portu PA10 powiązanego z Timerem1 kanał 3. Konfigurację pokazuje rysunek.



Rysunek 0501

W zakładce „DMA Setings” trzeba je włączyć naciskając przycisk „ADD”.



Rysunek 0502

Po wygenerowaniu kodu dopisujemy kod

Listing 5 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja05.zip

```
/* USER CODE BEGIN PV */

#define      SPAW_MIN           256
#define      SPAW_MAX           (1024-SPAW_MIN)     // Led Wspólna anda

#define      LEN_TABRNDPWM      512
uint16_t tabRndPwm[LEN_TABRNDPWM];


/* USER CODE END PV */



/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */


void SpawanieStart(){
      for(uint16_t x=0; x<LEN_TABRNDPWM; x++) tabRndPwm[x] = rand() % 0xFFFF;


      uint16_t len = rand() % SPAW_MAX + SPAW_MIN;
      if( len & 1 ){
            for(uint16_t x=0; x<LEN_TABRNDPWM; x++) tabRndPwm[x] = 0xFFFF;
      }
      HAL_TIM_PWM_Start_DMA(&htim1, TIM_CHANNEL_3, (uint32_t*)tabRndPwm, len);
}


void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim){
      if( htim == &htim1)SpawanieStart();
}


/* USER CODE END 0 */



      /* USER CODE BEGIN 2 */
      SpawanieStart();
      /* USER CODE END 2 */


      /* Infinite loop */
      /* USER CODE BEGIN WHILE */
      while (1)
      {

            /* USER CODE END WHILE */

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

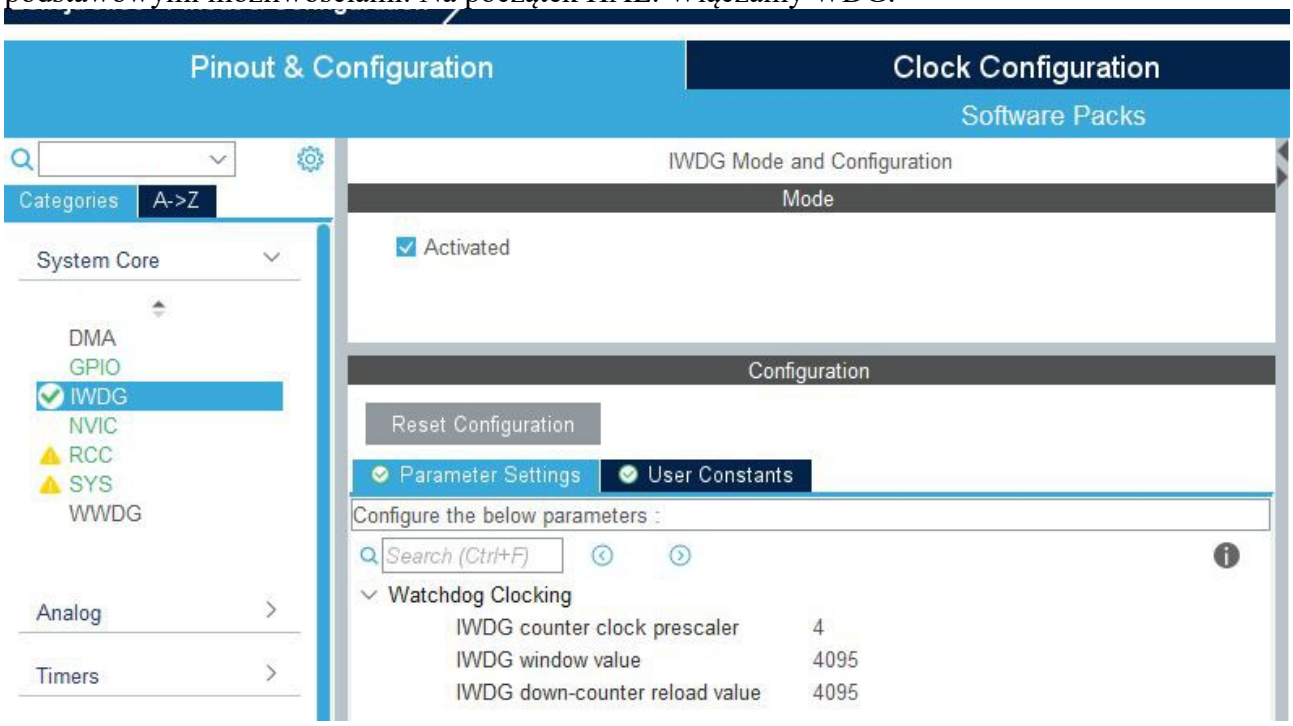
Przy miganiu LED'em przez DMA wielkiego zysku wydajności mikrokontrolera nie ma ale gdy DMA odtwarza sample dźwiękowe, zwłaszcza z dużą częstotliwością, lub wysyła dane do wyświetlacza graficznego kolorowego o dużej rozdzielczości zysk jest ogromny. Używając DMA można **generować obraz VGA** nie absorbują praktycznie CPU. Pokazałem to w filmie https://youtu.be/hsJBTR8MKO8

Pętla główna programu nie robi nic. Należałoby dodać tam obsługę WatchDog'a o czym w kolejnym rozdziale. Przed pętlą główną uruchamiany jest timer sterowany przez DMA funkcją „SpawanieStart()”. Funkcja ta generuje losową tabelę jasności świecenia LED. Po zakończeniu wyświetlania danych z tabeli generowane jest przerwanie, które ponownie uruchamia funkcję „SpawanieStart()”. Gdyby nie potrzeba ciągłego generowania nowych danych dla LED, DMA można by ustawić w tryb cykliczny. Takie rozwiązanie sprawdzi się gdy LED ma płynnie rozjaśniać się i wygaszać np. sterowanie światłem sygnalizatora kolejowego. Konfiguracja i efekt działania programu można zobaczyć w **Filmie 5** https://youtu.be/JNiXyKlvox0

Rozdział 2. Niezbędny a często pomijany Watchdog

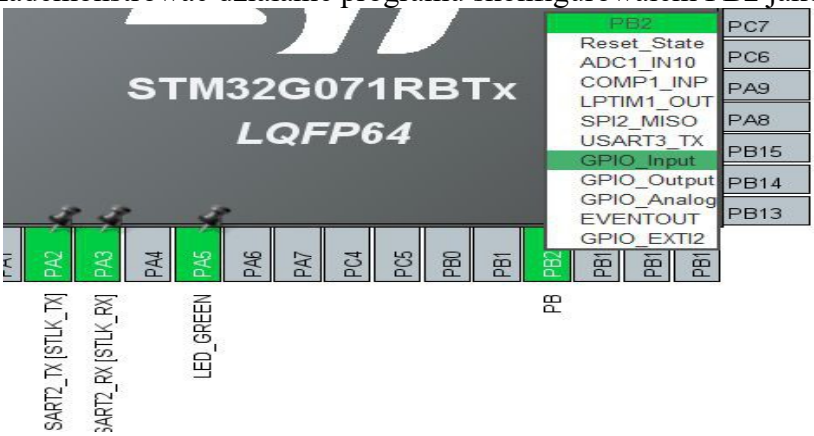
2.1 Prosty (IWDG)

W wielu STM32 IWDG ma możliwości podobne do WWDG w tym generowanie przerwań czy może być wyłączony po uśpieniu mikrokontrolera. Na razie jednak zajmiemy się jego podstawowymi możliwościami. Na początek HAL. Włączamy WDG.



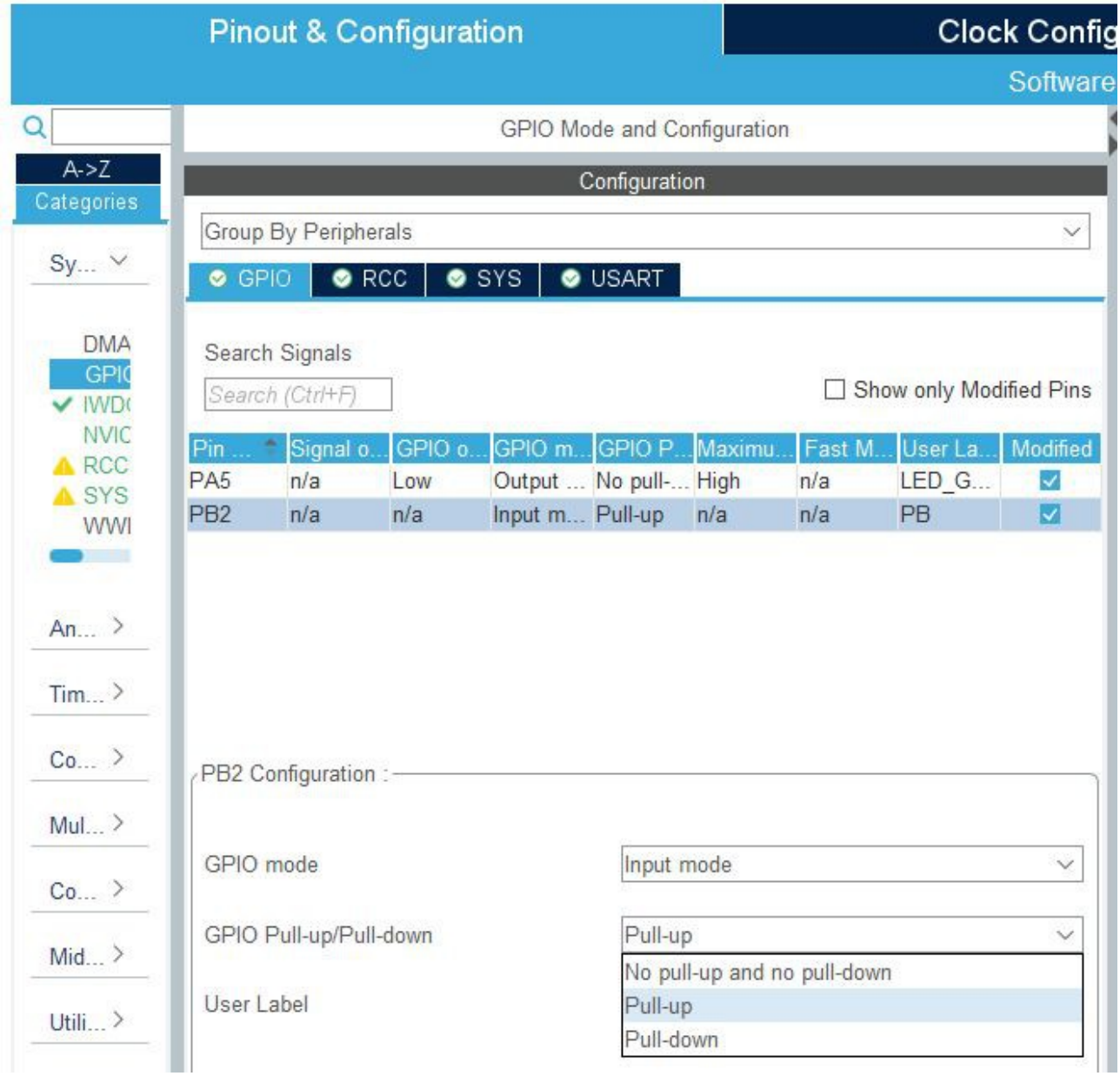
Rysunek 0601

Konfiguracji nie trzeba zmieniać ale aby wyjaśnić szczegóły. IWDG jest taktowany częstotliwością około 40kHz. Preskaler ustawiony jest na 4 a licznik liczy do 4095. To daje czas: 1/(40000/4/4095) =~ 400ms. Czas ten nie jest zbyt dokładny co wynika z generatora taktującego WDG i bezpiecznie jest założyć, że odchyłka może wynieść kilkanaście procent. Ja przyjmuję 10%. Jeśli w zadanym czasie WDG nie będzie zresetowany nastąpi reset mikrokontrolera. Aby zademonstrować działanie programu skonfigurowałem PB2 jako wejście



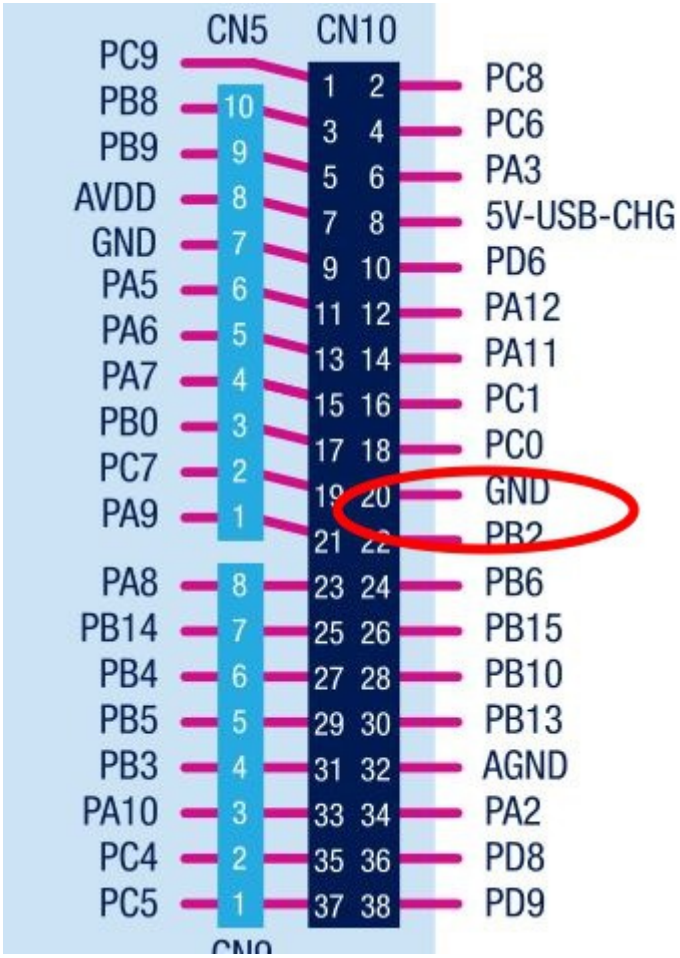
Rysunek 0602

z podciąganiem.



Rysunek 0603

Jak widać na rysunku w STM32 można podciągać do zasilania jak i masy czego droższy AVR nie potrafi. PB2 znajduje się obok pinu masy



Rysunek 0604

co pozwoli w prosty sposób zewrzeć wejście z masą uaktywniając je. Kod programu jest banalny

Listing 6 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja06.zip

```c
/* USER CODE BEGIN 0 */
void HAL_IncTick(void){
	uwTick += (uint32_t)uwTickFreq;

	uint16_t static timLed = 2000;


	if( --timLed == 0 ){
		timLed = 500;
		HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
	}
}

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)  {
	  while( ! HAL_GPIO_ReadPin(PB_GPIO_Port, PB_Pin) ){
		  HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
	  }
	  HAL_IWDG_Refresh(&hiwdg);
	  __WFI();

    /* USER CODE END WHILE */

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

Obsługę Led na przerwaniu już poznaliście. Tym razem jednak LED miga po 2 sekundach od resetu. W pętli głównej jest obsługa IWDG oraz usypianie CPU (komenda „__WFI();”). Dodatkowo odczytywany jest stan pinu wejściowego. Jeśli jest zwarty z masą CPU „kręci” się w wiecznej pętli co po 400ms spowoduje reset mikrokomntrolera. Jak to wygląda prezentuje **film 6** https://youtu.be/Ita1j_5BoS8

Używając bibliotek LL można zrobić to samo angażując mniej pamięci (niecałe 5kB FLASH zamiast prawie 11).

Listing 7 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja07.zip

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void initIrqTimSys(){
	LL_SYSTICK_EnableIT();		//  @paplociennik
	//LL_Init1msTick( LL_RCC_GetSystemClocksFreq() );

	/*
	//SysTick_Config(SystemCoreClock/1000);
	//SysTick_Config(SysTick->LOAD);
	SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;
	NVIC_EnableIRQ(SysTick_IRQn);
	 */
}


void IncTick(void){
//	uwTick += (uint32_t)uwTickFreq;
	uint16_t static timLed;


	if( ++timLed >= 500 ){
		timLed = 0;
		LL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
	}
}
/* USER CODE END 0 */



	/* Infinite loop */
	/* USER CODE BEGIN WHILE */
	while (1)  {
		LL_IWDG_ReloadCounter(IWDG);
		__WFI();
		/* USER CODE END WHILE */

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

Tym razem nie bawiłem się na obsługę przycisku. **Film 7**, znajduje się pod linkiem https://youtu.be/8NXcx9vKAkc

2.2 Zaawansowany Watchdog czyli WWDG

Okienkowy WDG jest pewniejszy od prostego bo zresetuje mikrokontroler nie tylko gdy na czas nie zostanie zresetowany ale też gdy resetowany jest zbyt często. Niestety WWDG nie pracuje po uśpieniu mikrokontrolera a co za tym idzie nie może wybudzić uśpionego mikrokontrolera. Cenną zaletą WWDG może być generowania przerwań (podobnie jak w AVR) gdy zadziała. W AVR przerwania od WDG wykorzystywałem do określenia miejsca w programie, które powodowało jego zawieszenie. Zadanie było trochę skomplikowane bo po odczytaniu ze stosu adresu powrotu do programu należało sprawdzić w pliku listingu (*lss) generowanym przez kompilator gdzie nastąpiło zawieszenie programu.

Listing 8 film 8

2.3 Zatrzymanie WDG i innych peryferii w czasie debugowania

Zatrzymanie programu debugerem zwykle nie zatrzymuje Watchdoga. W konsekwencji tracimy połączenie z mikrokontrolerem a on zostaje zresetowany i podejmuje pracę. Aby tego uniknąć należy zaraz za „main()" wywołać funkcję:

```
__HAL_DBGMCU_FREEZE_IWDG();
```

W praktyce to jest makro w większości wypadków odpowiada operacji:

```
DBGMCU -> APB1FZ |= DBGMCU_APB1_FZ_DBG_IWDG_STOP;
```

W przypadku STM32G07x należy posłużyć się operacją:

```
SET_BIT(DBG->APBFZ1, DBG_APB_FZ1_DBG_IWDG_STOP); // G07x
```

Czasem konieczne (zależy od CubeMX) jest zablokowanie rozłączenia debugera po uśpieniu lub zatrzymaniu CPU. Wykonujemy to ustawiając rejestr DGBMCU:

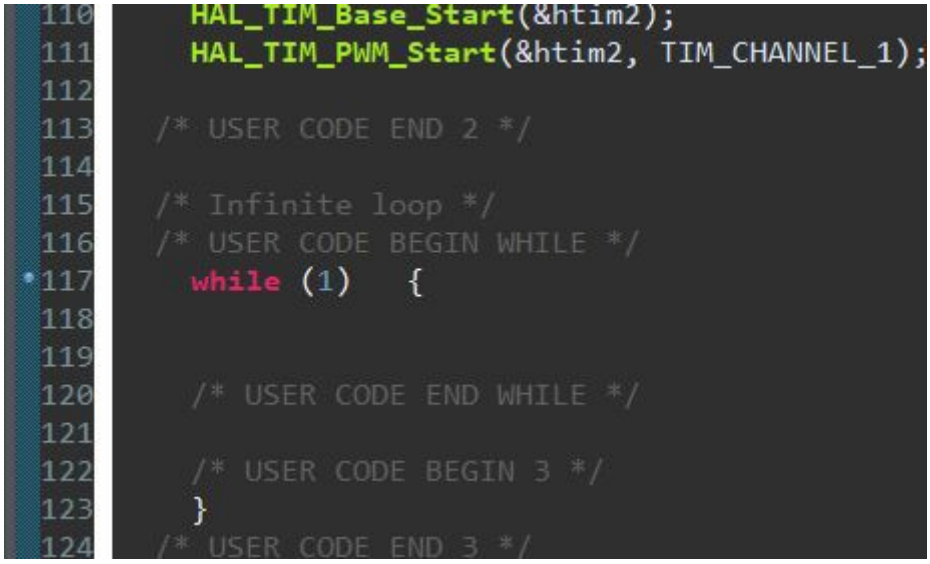
```
DBGMCU->CR |= DBGMCU_CR_DBG_SLEEP | DBGMCU_CR_DBG_STOP | DBGMCU_CR_DBG_STANDBY;
```

Jeśli konieczne jest zatrzymanie timerów po wejściu w debugowanie należy wywołać makra:

```
__HAL_DBGMCU_FREEZE_TIM1();
__HAL_DBGMCU_FREEZE_TIM2();
__HAL_DBGMCU_FREEZE_TIM3();
__HAL_DBGMCU_FREEZE_TIM4();
```

i kolejne jeśli używamy więcej timerów. Do zaprezentowanie możliwości zatrzymywania timerów posłużymy się przykładem z lekcji 4. Na razie nie wstawiamy funkcji „`__HAL_DBGMCU_FREEZE_TIM2();`"

tylko ustawiamy pułapkę przy „while(1)" (CYTL+SHIFT+B, lub klikając myszką)



Rysunek 0901

Po skompilowaniu i wgraniu programu przez „F11” uruchamiamy go „F8”. Program zatrzyma się na pułapce ale Led ciągle miga. Po dodaniu w kodzie funkcji „*__HAL_DBGMCU_FREEZE_TIM2();*”

zaraz za main():

Listing 9 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja09.zip

```
int main(void)
{
  /* USER CODE BEGIN 1 */

      //----- Zatrzymuj IWDG podczas debugowania -----//
      __HAL_DBGMCU_FREEZE_IWDG(); // DBGMCU -> APB1FZ |=
DBGMCU_APB1_FZ_DBG_IWDG_STOP;
      //----- wersja dla LL
      //SET_BIT(DBG->APBFZ1, DBG_APB_FZ1_DBG_IWDG_STOP);  // G07x


      //----- Nie rozlaczaj z debugerem po usypieniu, stop, itp
      //       DBGMCU->CR |= DBGMCU_CR_DBG_SLEEP | DBGMCU_CR_DBG_STOP |
DBGMCU_CR_DBG_STANDBY;   //

//       __HAL_DBGMCU_FREEZE_TIM1();
        __HAL_DBGMCU_FREEZE_TIM2();
```

Po zatrzymaniu programu timer także się zatrzyma i LED nie będzie migać. Podobnie można zatrzymać inne peryferia jak SMBUS czy LPTIM. W **filmie 9** https://youtu.be/QB6jqDT8TWk prezentuję jak to działa.

Rozdział 3. Określenie przyczyny resetu CPU

Do czego może być potrzebne określenie przyczyny resetu? Zawsze warto w jakiś sposób logować informacje o pracy systemu. Najprostszym sposobem będzie wysyłanie informacji na UART, w bardziej rozbudowanych mogą to być logi na nośniku (wewnętrzna lub zewnętrzna pamięć FLASH, karta SD, PenDrive, Dysk twardy) lub wysyłane przez Ethernet lub GSM. Dzięki temu w razie nieprawidłowej pracy można z logów wywnioskować co się dzieje z naszym systemie. Ze względu na to, ze będziemy mieli do czynienia z coraz bardziej rozbudowanymi programami, szczegóły konfiguracji będą na filmach. W programie pokazowym włączamy IWDG, UART jest domyślnie skonfigurowany tak jak trzeba. Uzupełniamy kod wygenerowany przez CubeMX następującymi funkcjami:
Listing 10 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja10.zip

```c
/* USER CODE BEGIN PV */
struct RstFlags{
        uint8_t power: 1;
        uint8_t pin: 1;
        uint8_t soft: 1;
        uint8_t bod: 1;
        uint8_t wdg: 1;
        uint8_t iwdg: 1;
        uint8_t wwdg: 1;
}RstFlags;
/* USER CODE END PV */

/* USER CODE BEGIN 0 */
uint16_t volatile timLED, timUART = 2000;
void HAL_IncTick(void){
        uwTick += (uint32_t)uwTickFreq;

        if ( timLED ) timLED--;
        if ( timUART ) timUART--;
}
/* USER CODE END 0 */

        /* USER CODE BEGIN 2 */
                if ( __HAL_RCC_GET_FLAG(RCC_FLAG_PWRRST) ) RstFlags.power = 1;
                if ( __HAL_RCC_GET_FLAG(RCC_FLAG_IWDGRST) ) RstFlags.iwdg = RstFlags.wdg = 1;
                if ( __HAL_RCC_GET_FLAG(RCC_FLAG_IWDGRST) ) RstFlags.wwdg = RstFlags.wdg = 1;
                if ( __HAL_RCC_GET_FLAG(RCC_FLAG_PINRST) ) RstFlags.pin = 1;
                __HAL_RCC_CLEAR_RESET_FLAGS();  // RCC->CSR |= RCC_CSR_RMVF;    // Kasuje flagi

                char txt[100];
                strcpy(txt, "\r\n***** Reset *** ");
                if( RstFlags.power ) strcat(txt, "Pwr ");
                if( RstFlags.bod ) strcat(txt, "Bod ");
                if( RstFlags.wdg ) strcat(txt, "Wdg ");
                if( RstFlags.pin ) strcat(txt, "Pin ");
                if( RstFlags.soft ) strcat(txt, "Soft ");
                strcat(txt, "\r\n");

                HAL_UART_Transmit(&huart2, (uint8_t *)txt, sizeof(txt), 10);
        /* USER CODE END 2 */

        /* Infinite loop */
        /* USER CODE BEGIN WHILE */
        while (1)  {
                HAL_IWDG_Refresh(&hiwdg);
                __WFI();

                /* USER CODE END WHILE */

                /* USER CODE BEGIN 3 */

                while( ! HAL_GPIO_ReadPin(PB_GPIO_Port, PB_Pin) ) ;


                if( ! timLED ){
                        timLED = 500;

                        HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
                }


                if( ! timUART ){
                        timUART = 2000;

                        const char txt[] = "Test. ";
                        HAL_UART_Transmit(&huart2, (uint8_t *)txt, sizeof(txt), 10);
                }
        }
        /* USER CODE END 3 */
```

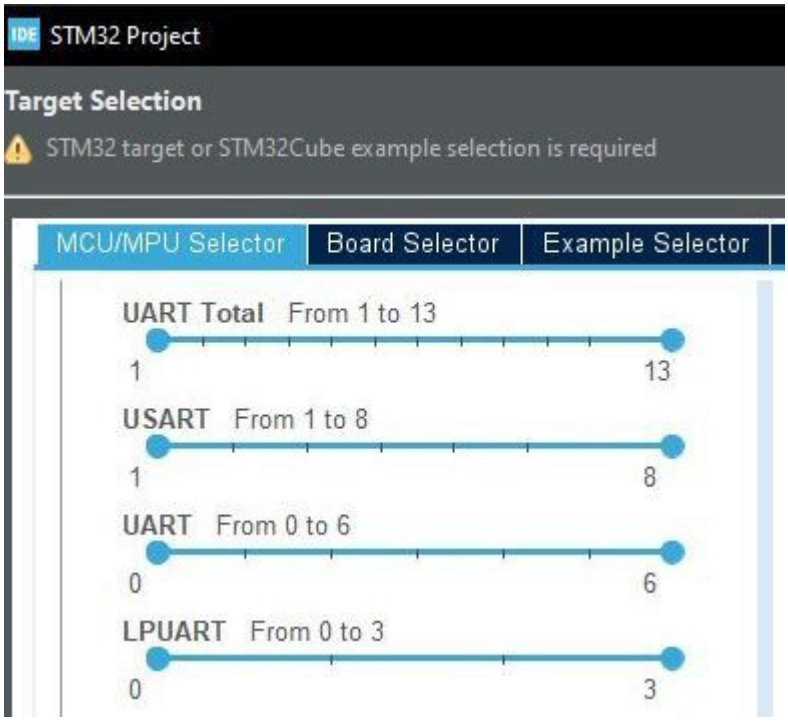
Najistotniejszy fragment to odczytywanie flag CPU „__HAL_RCC_GET_FLAG()” i ich kasowanie „__HAL_RCC_CLEAR_RESET_FLAGS()”. Wyniki działania programu obserwujemy w oknie terminala

Rysunek 1001

co widać w **Filmie 10** https://youtu.be/e2Jca1EHLnc

Rozdział 4. UART

UART jest jednym z najczęściej używanych interfejsach w mikrokontrolerach. Już 8051 posiadał jeden UART co często było zbyt małą ilością. Nieliczni producenci jak „Dallas" wyposażali swe 8051 w dwa UART. Problem małej ilości UART zauważyli to twórcy AVR i wiele z nich jest wyposażonych w dwa takie interfejsy a niektóre nawet w cztery. W STM32 liczba UART sięga 13.

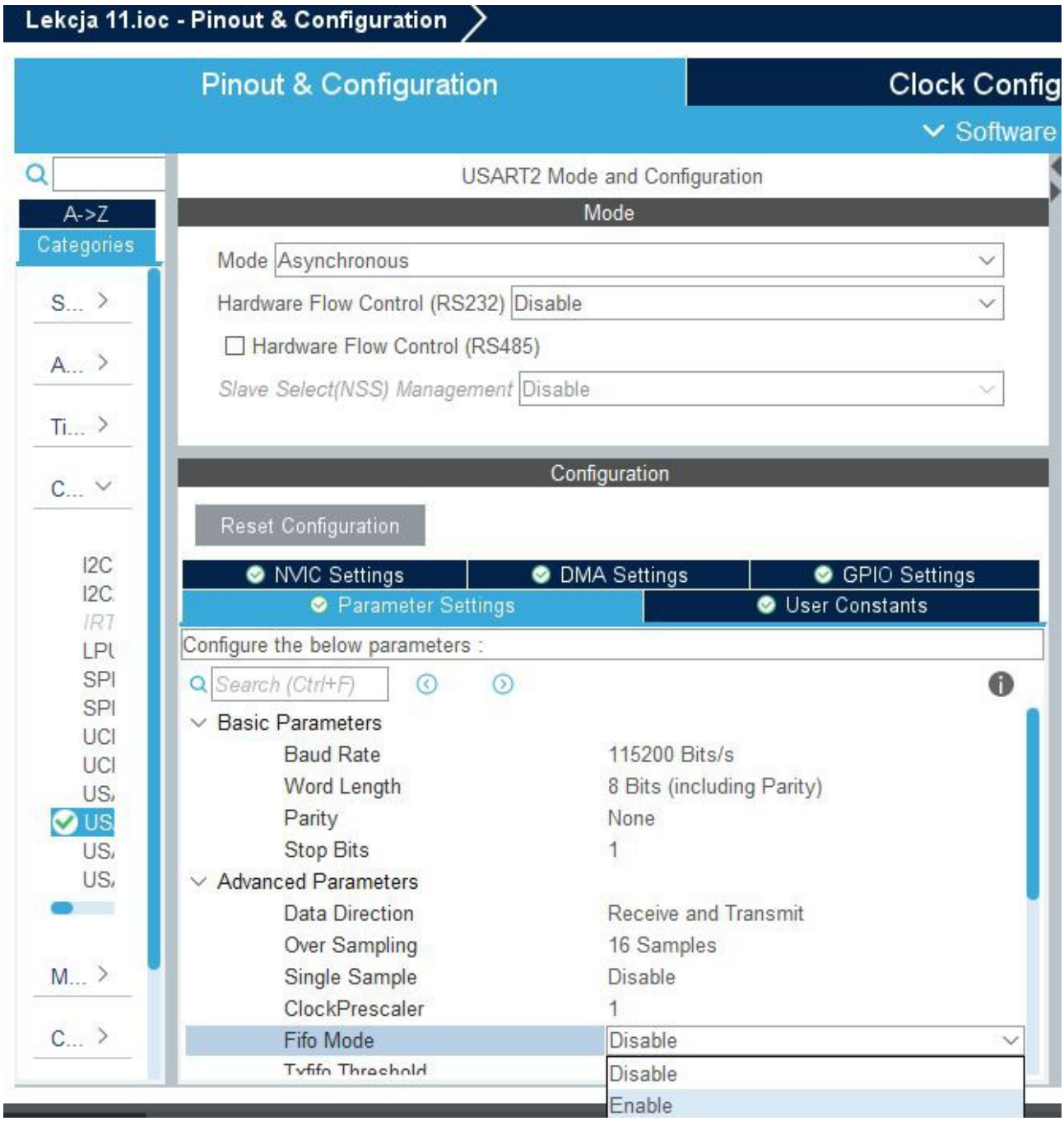


Rysunek 1101

W razie potrzeby liczbę interfejsów można zwiększyć przez I2c/SPI układami SC16IS7xx co pisano w: https://er-mik.prv.pl/projekty_ze/2023-11_73_ZE_Dodatkowe_porty_UART_w_Arduino.pdf

4.1 Polling

Na początek opiszę najprostszą ale i najgorszą metodę komunikacji, która używa funkcji blokujących, których należy unikać tak samo jak DELAY. W pierwszej kolejności należy skonfigurować UART. Domyślne ustawienia są prawie dobre, jedyne co należy zrobić to włączyć FIFO



Rysunek 1102

O włączeniu IWDG nie pisze bo to, jak powiedział klasyk, „oczywista oczywistość”. Po wygenerowani kodu przechodzimy no próby nadawania. O konfiguracji programu terminala wspominałem wcześniej więc daruję sobie powtarzanie tego. Włączenie optymalizacji „-Os” mam nadzieję, że też weszło w krew.

4.1.1 Nadawanie

Jest proste ale trzeba zwrócić uwagę na czas transmisji, bo jeśli źle ustawimy TIMEOUT to zostanie ona przerwana. Ponadto na czas nadawania program „wisi” i im transmisja będzie wolniejsza a danych do wysłania więcej tym dłużej CPU jest „zablokowany”. Może mieć to istotne

znaczenie gdy musimy szybko reagować na zdarzenia lub WatchDog musi mieć ustawione częste odświeżanie. Dane wysyłamy w banalny sposób:

Listing 11 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja11.zip

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
uint16_t volatile timLED, timUART = 2000;
void HAL_IncTick(void){
	uwTick += (uint32_t)uwTickFreq;

	if ( timLED ) timLED--;
	if ( timUART ) timUART--;
}
/* USER CODE END 0 */

	/* Infinite loop */
	/* USER CODE BEGIN WHILE */
	while (1)  {
		HAL_IWDG_Refresh(&hiwdg);
		__WFI();

		/* USER CODE END WHILE */

		/* USER CODE BEGIN 3 */
		if( ! timLED ){
			timLED = 500;

			HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
		}


		if( ! timUART ){
			timUART = 2000;

			uint16_t static cnt;

			char txt[20];
			sprintf(txt, "Test %d\r\n", ++cnt);
			HAL_UART_Transmit(&huart2, (uint8_t *)txt, sizeof(txt), 10);
		}

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

Tym razem pokazałem wszystkie zmiany jakie należy wprowadzić w kodzie ale w kolejnych przekładach ograniczę się tylko do istotnych elementów pomijając obsługę IWDG czy migania LED'em.

Film 11 https://youtu.be/py_F87s5egE

Jak obliczyć TIM EOUT (ostatni parametr funkcji „HAL_USART_Transmit”)? Prosto, czas

transmisji bajtu * liczba bajtów. Wysyłając teksty trzeba mieć na uwadze fakt, że często wysyłamy też bajt o wartości 0 oznaczający koniec teksu. Przykładowe obliczenie dla 9600 8N1. Łączna liczba bitów 10 (bit startu, 8 danych, bit stopu). Czas transmisji bajtu = 1/(9600/10) =~ 0,001 sekundy co daje 1ms.

4.1.2 Odbiór

Dość ciężko zrealizować skuteczny odbiór przez UART bez przerwań czy DMA używając HAL. Spróbuję to jednak zrobić tak aby CPU realizował też inne zadania. Wskazane włączyć FIFO (rysunek 1102) jeśli mikrokontroler je posiada.
Do poprzedniego programu, w pętli „while(1)", dodajemy kod:

```
        uint8_t pData;
        if( HAL_UART_Receive(&huart2, &pData, 1, 10) == HAL_OK ){
              HAL_UART_Transmit(&huart2, &pData, 1, 10);
        }


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

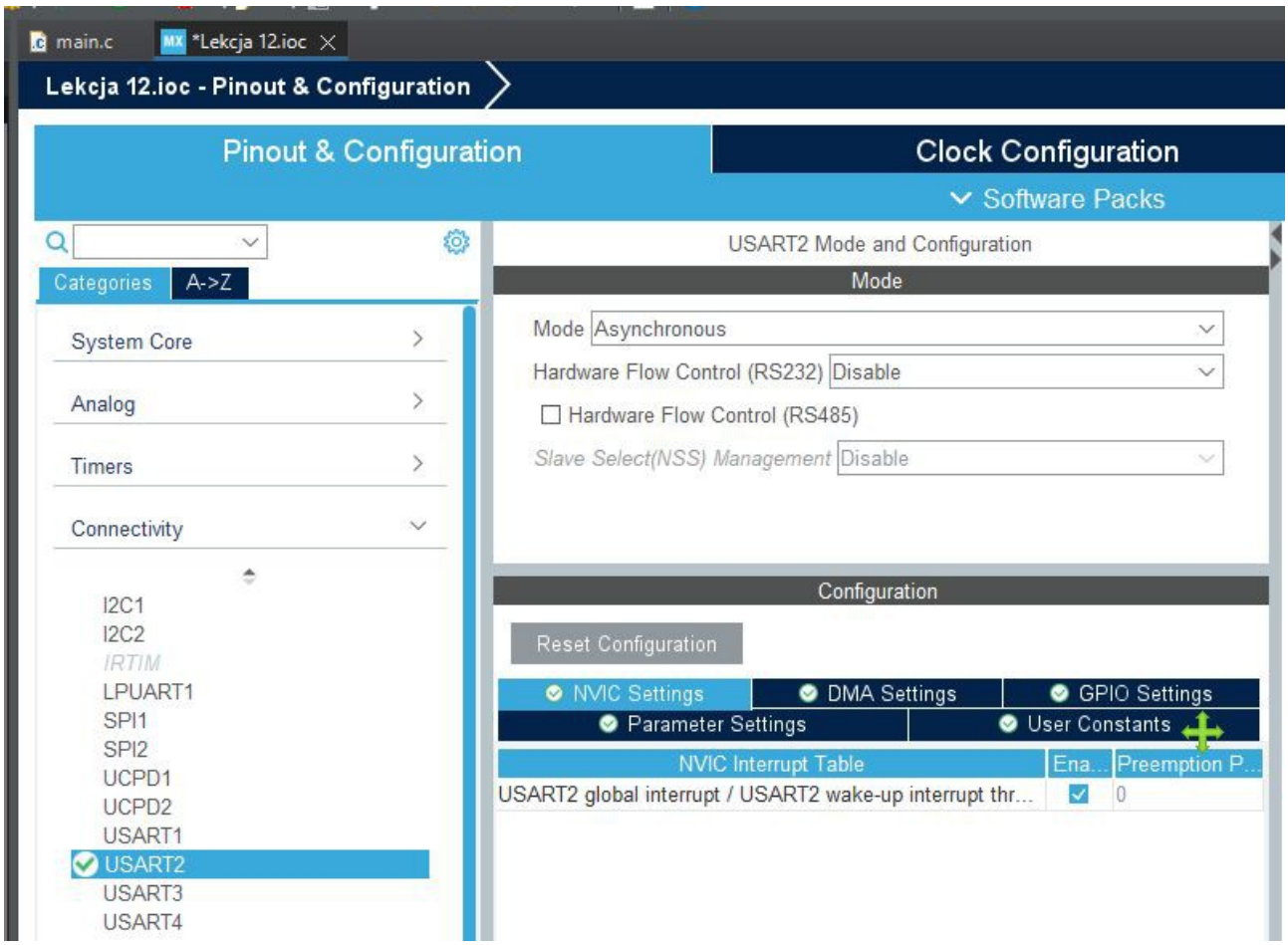
Ten dodatek zawiera już kod z listingu 11. Efekt działania programu możemy zobaczyć na filmie 11 . Pokazany kod obioru nie jest zalecany bo jest narażony na gubienie znaków. Aby tego uniknąć należy skorzystać z przerwań lub lepiej DMA i przerwania od końca ramki.

4.2 Przerwania

Trzeba być świadomym, że wykorzystanie przerwań nadawczych może nieco wydłużyć czas transmisji. Wyjaśniam to w filmie https://youtu.be/Iq1ziSCAFHI

4.2.1 Nadawanie

Nadawanie jest prostsze niż bez przerwań bo nie zawracamy sobie głowy czasem nadawania. Konfigurując UART musimy włączyć przerwania od niego



Rysunek 1201

Program kopiujemy z lekcji 11 z tym, że funkcję

```
HAL_UART_Transmit(&huart2, (uint8_t *)txt, sizeof(txt), 10 );
```

zamieniamy na:

```
HAL_UART_Transmit_IT(&huart2, (uint8_t *)txt, sizeof(txt) );
```

4.2.2 Odbiór

Tu będzie trochę więcej roboty. Tworzymy funkcje:

Listing 12 http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja12.zip

W pętli głównej funkcję odbiorczą zamieniamy na następującą:

```
int16_t pData;
if( (pData=gerByteUart() ) != -1 ){
      timUART = 5000;
      HAL_UART_Transmit(&huart2, (uint8_t *)&pData, 1, 10);
}
```

Pozostaje jeszcze wywołać InitUARTrx();

```
	/* USER CODE BEGIN 2 */
	InitUARTrx();
```

Niestety opisany powyżej sposób odbioru nie jest optymalny i będzie sprawiał kłopoty przy większych prędkościach transmisji a to dlatego, że HAL ma duży narzut kodu. Można by przejąć

```
#define		LEN_BUF_RX_UART			16
uint8_t pDataUARTrx, ptrBufUartWr, ptrBufUartRd, DataUARTrx[LEN_BUF_RX_UART];
void InitUARTrx(){
	HAL_UART_Receive_IT(&huart2, &pDataUARTrx, 1);
}


void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
	if( huart == &huart2 ){
		DataUARTrx[ptrBufUartWr] = pDataUARTrx;
		if( ++ptrBufUartWr >= LEN_BUF_RX_UART ) ptrBufUartWr = 0;
		HAL_UART_Receive_IT(&huart2, &pDataUARTrx, 1);
	}
}


int16_t gerByteUart(){
	if( ptrBufUartWr != ptrBufUartRd ){
		uint8_t znak;
		znak = DataUARTrx[ptrBufUartRd];
		if( ++ptrBufUartRd >= LEN_BUF_RX_UART ) ptrBufUartRd = 0;
		return znak;
	}
	return -1;
}
```

przerwanie odbiorcze dzięki czemu nike trzeba ciągle wywoływać „HA_UART_Recive_IT” ale dalej pokażę lepszy sposób.

4.3 O czym się nie mówi czyli obsługa błędów Odbioru

Bez niej odbiór przestaje działać o czym w innych książkach czy kursach nie piszą. Dlaczego tak jest? Otóż w realnych warunkach mogą pojawić się błędy spowodowane zakłóceniami. W takiej sytuacji HAL wyłącza odbiór. Dlatego musimy przejąć kontrolę nad obsługą błędów.

```
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart){
	if( huart == &huart2 ) InitUARTrx();
}
```

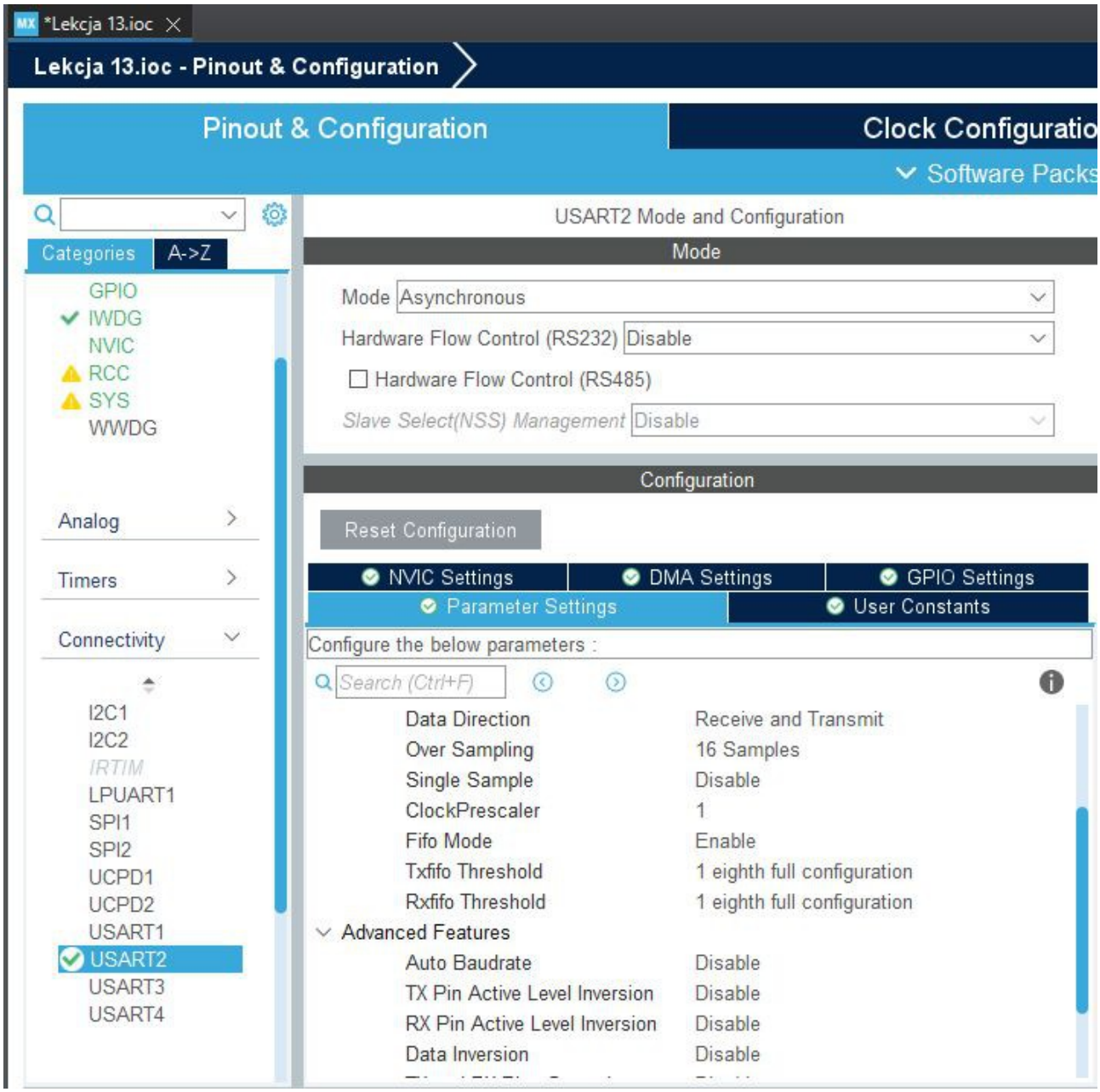
W obsłudze błędów inicjalizujemy odbiór. Cały kod obsługi UART na przerwaniach znajdziemy na serwerze w **Listingu 12** http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja12.zip oraz w **filmie 12** https://youtu.be/5GJFET6cTQ8

4.4 DMA

DMA pozwala odciążyć CPU dzięki czemu można obsłużyć naprawdę duże szybkości transmisji. W AVR jest to niemożliwe i realne prędkości odbiegają od deklarowanych przez producenta co opisałem w artykule https://er-mik.prv.pl/projekty_ze/2023-10_85_ZE_Jak_jest_najwieksza_predkosc_SPI_w_Arduino_z_AVR.pdf

4.4.1 Nadawanie

Przy konfiguracji UART należy włączyć DMA



Rysunek 1301 Trzeba też włączyć globalne przerwania od UART

Rysunek 1302

w przeciwnym wypadku po jednym wysłaniu danych HAL będzie zwracał status „HAL_BUSY”.

Trzeba też zamienić:

```
HAL_UART_Transmit_IT(&huart2, (uint8_t *)txt, sizeof(txt) );
```

na:

```
HAL_UART_Transmit_DMA(&huart2, (uint8_t *)txt, sizeof(txt) );
```

4.4.1 Odbiór

Nie widzę większego sensu aby odbierać pojedyncze znaki przez DMA. Jeśli jednak ktoś się uprze to jak w przypadku nadawania zmieniamy _IT na _DMA. W odbiorze danych przez DMA będzie w następnej lekcji.

4.5 Odciążenie CPU czyli przerwanie od końca ramki (IDLE)

(wydajne zwłaszcza z wykorzystaniem DMA)
UART w STM32 ma ciekawą cechę, pozwala na generowanie przerwania gdy pomiędzy znakami odstęp będzie dłuższy niż 1,5 znaku co zwykle oznacza koniec ramki danych.
Bardzo rozczarował mnie Linux konkretnie Debian, w którym odstępy pomiędzy znakami przy większych prędkościach (od 460kb/s wzwyż) wynosiły nawet kilka znaków. Winę ewidentnie ponosił system a nie programy bo wypróbowałem kilka terminali. Co ciekawe na Windows (ten sam komputer) problemu nie było.
Aby odbierać dane w opisany sposób należy wywołać funkcje:

```
void InitUARTrx(){
	HAL_UARTEx_ReceiveToIdle_DMA(&huart2, DataUARTrx, LEN_BUF_RX_UART);
}
```

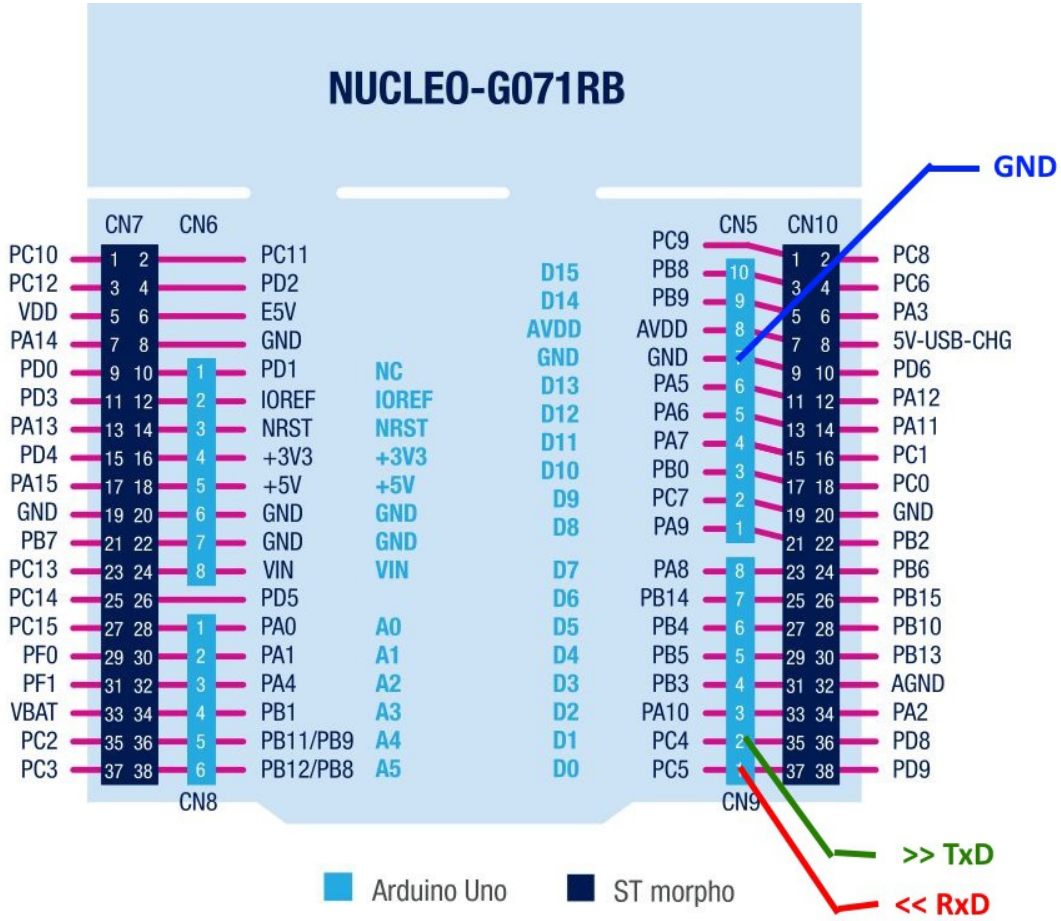
i czekać na przerwanie

```
void HAL_UARTEx_RxEventCallback(UART_HandleTypeDef *huart, uint16_t Size){
	if( huart == &huart2 ){
		FluartRxFull = Size;
		InitUARTrx();
	}
}
```

z którego dowiemy się ile znaków zostało odebranych. Funkcje nadawania i odbioru z **Listingu 13** http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja13.zip prezentuje **Film 13** https://youtu.be/vCXnZwXHuHE

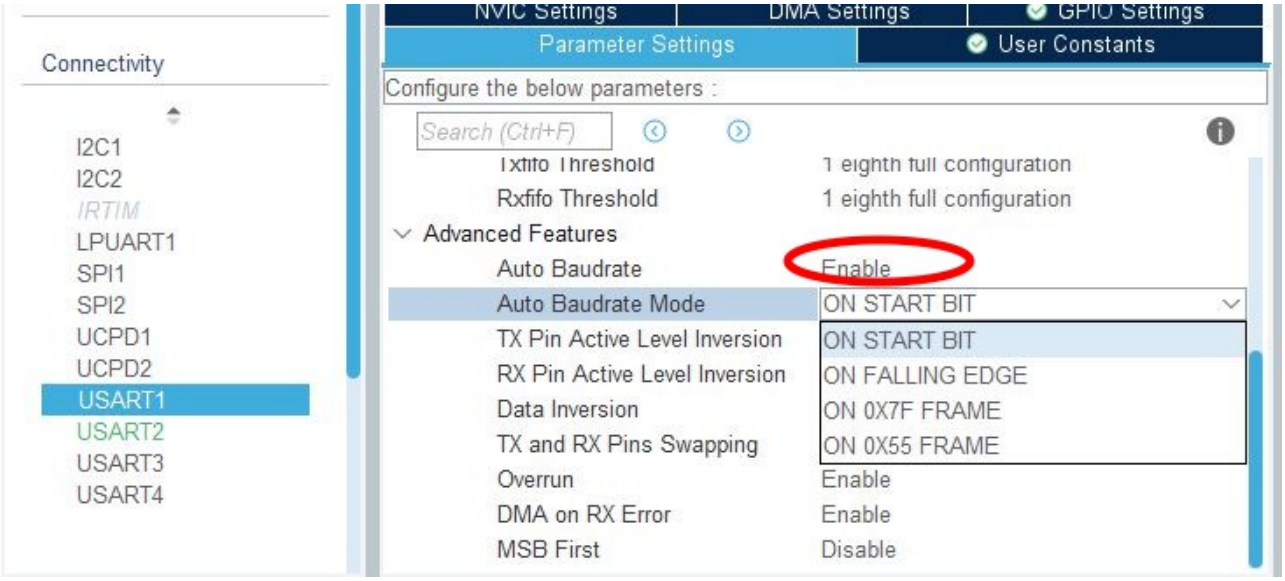
4.6 Marzenie „AVR'owców" czyli AutoBaud

Niestety nie wszystkie zaawansowane funkcje są dostępne dla wszystkich UART. W niektórych mikrokontrolerach nie do każdego UART można podłączyć DMA. W przypadku STM32G071 tylko UART1 posiada funkcje AutoBaud czego nie widać w CubeMX. Dlatego do lekcji 14 wykorzystamy UART1. Umiejscowienie wyprowadzeń UART przedstawia rysunek



Rysunek 1401
Do NUCLEO należy podłączyć konwerter USB-UART.

Rysunek 1402
Należy pamiętać o zmianie portu vCOM w programie terminala. Pozostało jeszcze skonfigurować UART1 w CubeMX



Rysunek 1403

Sposób wykrywania prędkości transmisji wybieramy w „Auto Baudrate Mode". Domyślne ustawienie jest całkiem dobre. Pomiędzy pierwsza a drugą opcją nie widzę praktycznych różnic. Opcja 3 „ON 0x7F" sprawdzi się gdy budujemy własne systemy i decydujemy od czego ma zaczynać się ramka danych. Będziemy korzystali z przerwań dlatego należy je włączyć podobnie jak w poprzednich lekcjach.

Film 14 dostępny na: https://youtu.be/a_gCdOWFiRI

W filmie wspominam o problemie Auto Baud przy szybszych transmisjach. Rozwiązaniem jest taktowanie UART'a z większą częstotliwością niż CPU. Co może dać tego typu działanie można przeczytać w artykule: https://er-mik.prv.pl/projekty_ze/2024-03_95_ZE_Od_czego_zalecy_wydajnosci_CPU.pdf

Kody źródłowe (projekt 14 i **listingi**) znajdują się na:

http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja14.zip

4.7 Kolejka nadawcza

Temat często pomijany w książkach i kursach Internetowych. Gdy wysyłamy dane przez UART (i nie tylko) na przerwaniach czy DMA nie można wysłać kilku komunikatów jeden po drugim np.:

```
HAL_UART_Transmit_DMA(&huart2, &tekst1, len1 );
HAL_UART_Transmit_DMA(&huart2, &tekst2, len2 );
HAL_UART_Transmit_DMA(&huart2, &tekst3, len3 );
```

Wynika to z tego, że zaraz po uruchomieniu wysyłania tekstu1 funkcja „HAL_USART_Transmit" się zakończy mimo, że transmisja trwa nadal. Z tego powodu, kolejna transmisja się nie uda (funkcja zwróci HAL_BUSY). Problem jest tym większy im wolniejsza jest transmisja. Można by dodać delay:

```
HAL_UART_Transmit_DMA(&huart2, &tekst1, len1 );
HAL_Delay(1);
HAL_UART_Transmit_DMA(&huart2, &tekst2, len2 );
HAL_Delay(1);
HAL_UART_Transmit_DMA(&huart2, &tekst3, len3 );
```

Dobierając czas opóźnienia do szybkości transmisji i liczby wysyłanych danych. Tyle, że nic nie

zyskujemy w stosunku do funkcji blokujących bo program „wisi” w delay. Jak rozwiązać ten problem? Trzeba zrealizować mechanizm kolejki. Już nagrałem film na ten temat: https://youtu.be/3g2qlDEVK94 warto go obejrzeć aby kod programu był zrozumiały. Praktycznie ten sam projekt można pobrać z serwera: **listing 15** http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja15.zip
Jak działa mechanizm kolejki? Jeśli nie trwa transmisja (USART jest „wolny”) wysyłamy dane przez DMA czy na przerwaniach. Gdy trwa transmisja wskaźnik na dane jest kopiowany do kolejki. Tablica kolejki nie jest duża bo dla każdej transmisji trzeba zapamiętać wskaźnik (4 bajty) i i ich długość (2 bajty). Ze względu na to, że kopiujemy wskaźnik na dane, gdy znajdują się one w Ram MUSZĄ być zadeklarowane jako statyczne!
W książce omówię tylko najistotniejsze fragmenty kodu:

```
void printUART(char*txt){
	__NOP();
#if LEN_KOLEJKA_UART > 1
	if ( packedFifoUart >= LEN_KOLEJKA_UART ){		// jak wskaźnik zapisu "dogonił" odczyt to przepełniony bufor
		packedFifoUart = ptrKolejkaUartWr = ptrKolejkaUartRd = 0;
		//mamy problem
		return;
	}else{
		//----- Zapis do FIFO
		uint8_t *adr = (uint8_t*)txt;
		uint16_t len = strlen(txt);


		if( ++packedFifoUart == 1 ){	// Gdy FIFO puste
			HAL_UART_Transmit_DMA(&huart2, adr, len );
		}else{
			kolejkaTxUART[ptrKolejkaUartWr].adr = adr;
			kolejkaTxUART[ptrKolejkaUartWr].len = len;
			if( ++ptrKolejkaUartWr >= LEN_KOLEJKA_UART ) ptrKolejkaUartWr = 0;
		// Bufor kołolwy
		}
	}
	if ( packedFifoUart > packedFifoUartMAX ) packedFifoUartMAX = packedFifoUart;
	__NOP();
}
```

Dla próby można zmienić LEN_KOLEJKA_UART na zero i sprawdzić jak zachowuje się program. Zakończenie transmisji wywołuje przerwanie „***HAL_UART_TxCpltCallback***” te z kolei funkcję „***USART_TransmitNEXT***”, która, jeśli w kolejce czekają dane, wysyła je z wykorzystaniem DMA:

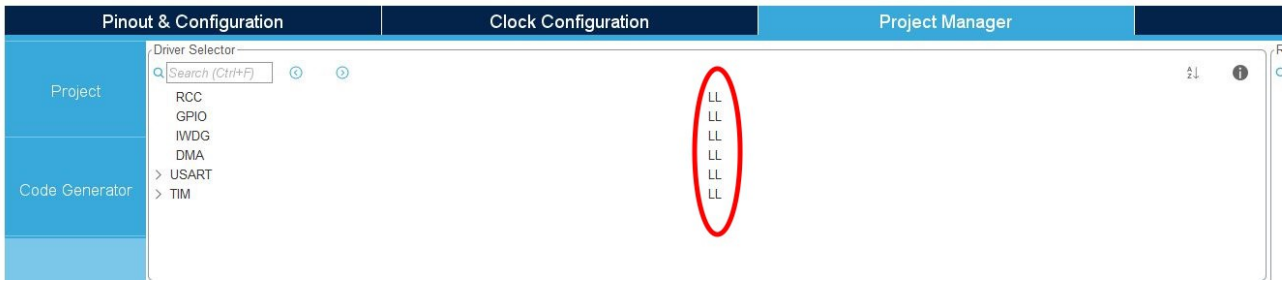
```
void UART_transmitNext(bool irq){
	UNUSED( irq );

	if ( --packedFifoUart > 0 ){			// Jak jest jeszcze coś w buforze
		//------ Odczyt FIFO
		uint8_t *adr = kolejkaTxUART[ptrKolejkaUartRd].adr;
		uint16_t len = kolejkaTxUART[ptrKolejkaUartRd].len;
		if( ++ptrKolejkaUartRd >= LEN_KOLEJKA_UART ) ptrKolejkaUartRd = 0;
	// Bufor kołowy

		HAL_UART_Transmit_DMA(&huart2, adr, len );
	}
}


void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart){
	if( huart == &huart2 ){
		UART_transmitNext(true);
	}
}
```

Kolejkowanie transmisji gwarantuje, że wysyłane komunikaty nie będą gubione chyba, że źle dobierzemy wielkość kolejki. Tablica kolejki zajmuje po 6 bajtów na komunikat ale to chyba nie problem biorąc pod uwagę, że niewielki STM32G071Rb ma 36kB RAM. Gdyby AVR miał tyle Ram to programiści byliby szczęśliwi (najwięcej RAM z AVR'ów ma Mega1284 „aż” 16kB). **Film 15** związany z powyższym kodem jest dostępny na: https://youtu.be/ZUCAYCnVxro

4.8 Biblioteki LL

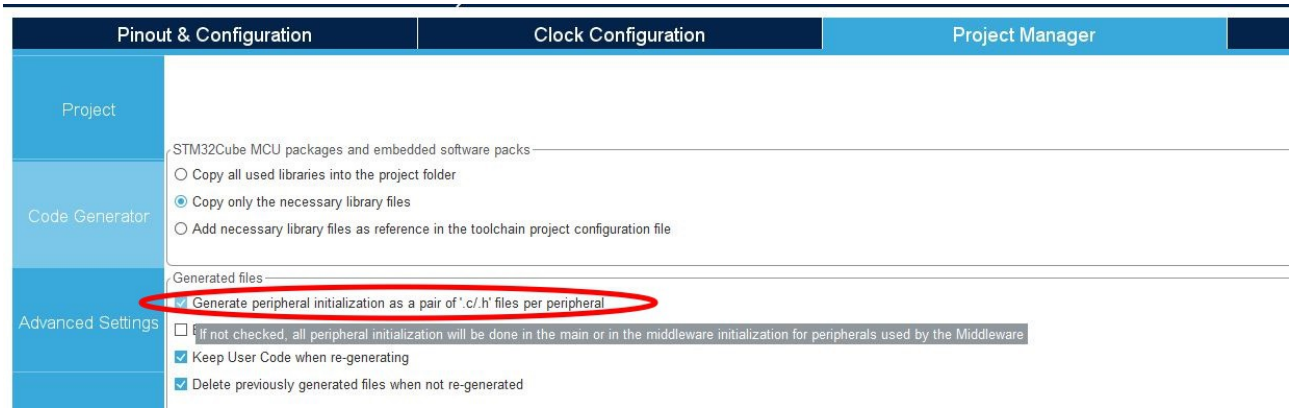
4.8.1 Polling

Ze względu na to, że odbiór nie ma korzyści w stosunku do wykorzystania HAL tematem nie będę się zajmował. Pokażę tylko jak nadawać. Trzeba jednak skonfigurować STM'a. W pierwszej kolejności trzeba wybrać w projekcie biblioteki LL a nie HAL:



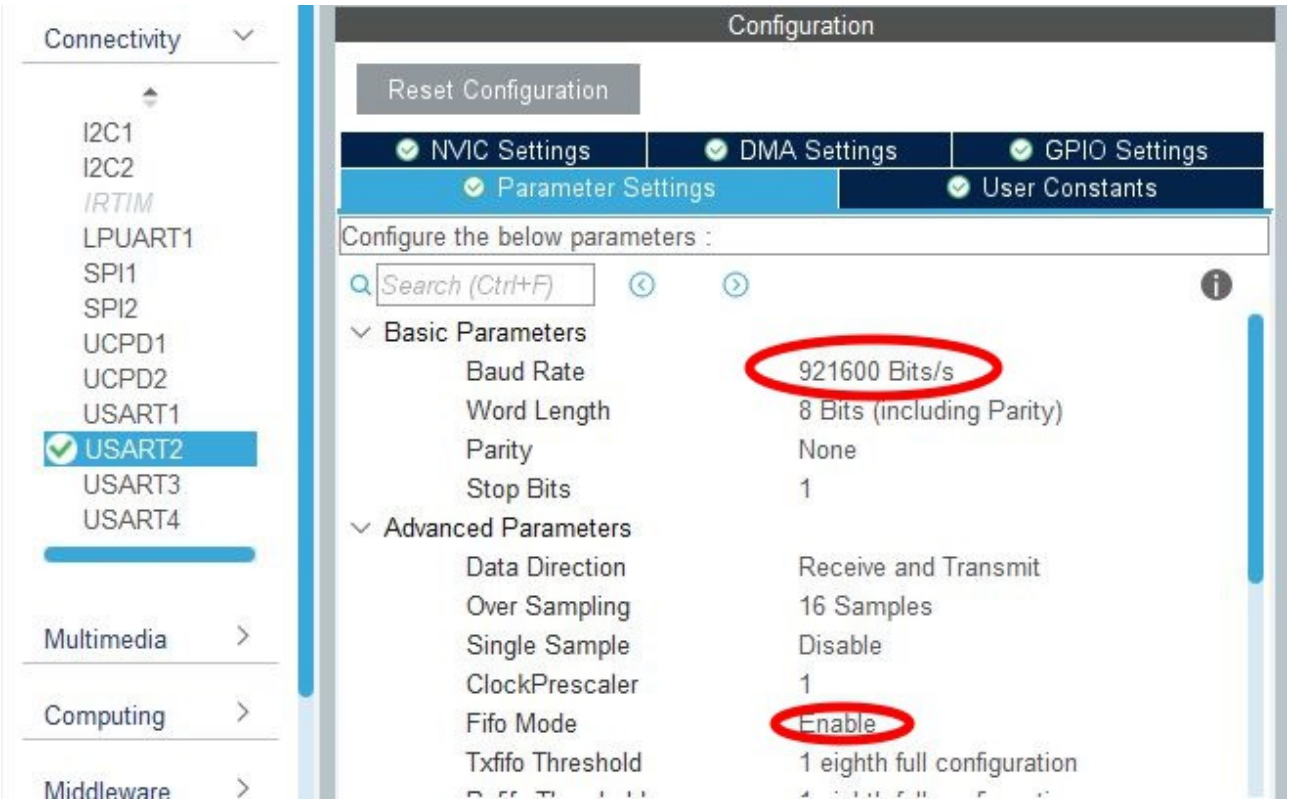
Rysunek 1601

Warto tez zaznaczyć „Gererate periypherial initialization as a pair 'c'/h' files per peripherial” nawet gdy korzystamy z HAL:



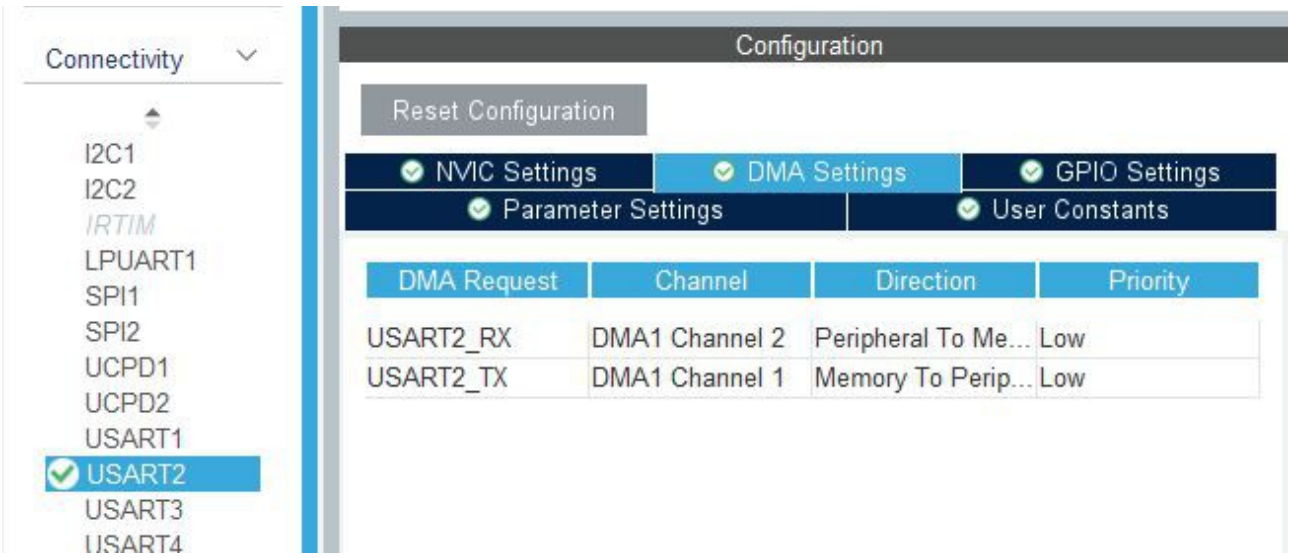
Rysunek 1602

Pozostaje skonfigurować UART:



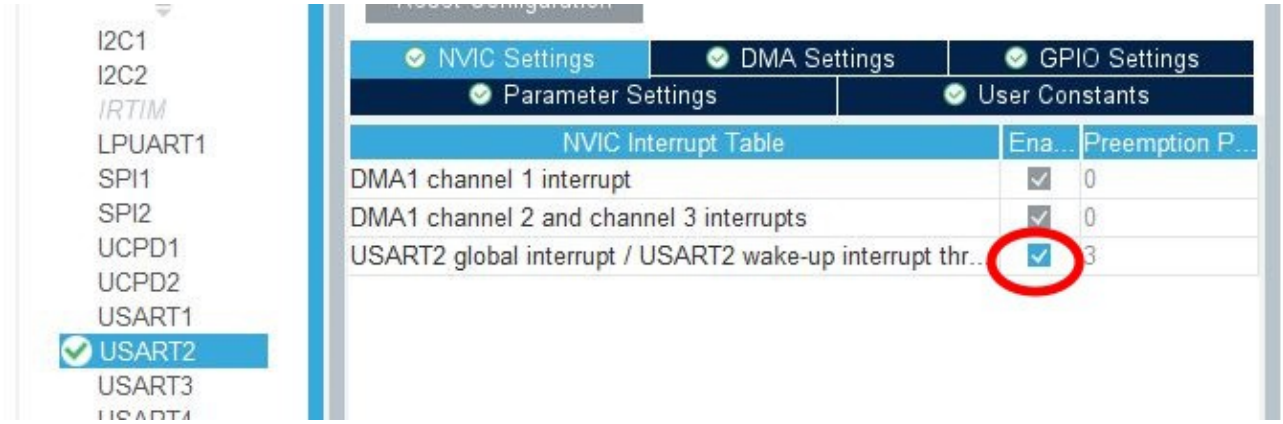
Rysunek 1603

Jak zaznaczono warto włączyć FIFO. Ze względu na to, ze będziemy w późniejszych przykładach używali DMA włączamy je:



Rysunek 1604

Przerwania też będą potrzebne:



Rysunek 1605

Pliki **projektu 16** można pobrać z:

http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja16.zip .

Film 16 https://youtu.be/6-3IvbFVXJI prezentuje kod programu. Przypominam, że książka i filmy nawzajem się uzupełniają. Najbardziej istotnym fragmentem jest funkcja wysyłająca dane:

```
uint8_t sendUARTtxt(char *txt ){
	LL_GPIO_SetOutputPin(UART2_SYNC_GPIO_Port, UART2_SYNC_Pin);
	for(uint8_t x=0; x<strlen(txt)+1; x++) {
		// Czekanie aż bufor nadawczy pusty
		while( ! LL_USART_IsActiveFlag_TXE(USART2) ) {
			/* obsluga timeout */
		}
		// Wpisane znaku do UART
		LL_USART_TransmitData8(USART2, txt[x] );
	}
	LL_GPIO_ResetOutputPin(UART2_SYNC_GPIO_Port, UART2_SYNC_Pin);
	return true;
}
```

Nie ma potrzeby inicjalizować UART'a bo robią to funkcje tworzone przez CubeMX (szczegóły w filie 16 https://youtu.be/6-3IvbFVXJI)

4.8.2 Przerwania

Większego sensu używania odbioru przez UART bez wykorzystania przerwań nie ma więc omówię to zagadnienie teraz razem z nadawaniem na przerwaniach. Oczywiście dużo wyjaśnień jest w filmie 16. Na początek w pliku „stm32g0xx_it.c" w handlerze obsługi przerwania „USART2_IRQHandler" dodajemy funkcje „irqUart2()":

```
void USART2_IRQHandler(void)
{
  /* USER CODE BEGIN USART2_IRQn 0 */

	extern void irqUar2();
	irqUar2();
```

Przy okazji możemy tez dodać funkcje obsługujące przerwania od DMA:

```
void DMA1_Channel1_IRQHandler(void)
{
  /* USER CODE BEGIN DMA1_Channel1_IRQn 0 */

	 extern void irqDmaUartTX();
	 irqDmaUartTX();

  /* USER CODE END DMA1_Channel1_IRQn 0 */
}

/**
  * @brief This function handles DMA1 channel 2 and channel 3 interrupts.
  */
void DMA1_Channel2_3_IRQHandler(void)
{
  /* USER CODE BEGIN DMA1_Channel2_3_IRQn 0 */

	 extern void irqDmaUartRX();
	 irqDmaUartRX();
```

W „main.c” dopisujemy funkcje obsługi przerwania od UART:

```
bool volatile FLtransitTx, FLuartOVR;
uint8_t *buf_uart_tx_prt;
uint16_t uart_tx_len, DMArxLen;

#define RX_UART_LEN             48
char rx_buffer[RX_UART_LEN];
uint8_t volatile rx_bufferCntWR;
uint8_t rx_bufferCntRD;

void irqUar2(){
        //------------ nadawanie --------------//
                if( LL_USART_IsActiveFlag_TXE(USART2) )
                        if( --uart_tx_len == ZIOBRO ){
                                LL_USART_DisableIT_TXE(USART2);
                                FLtransitTx = false;
                                return;
                        }
                        LL_USART_TransmitData8(USART2, *buf_uart_tx_prt++ );
                }
        //------------ obdiór --------------//
        //----- Sprawdzamy błędy i reagujemy na nie (ustawienie flagi błędu itp)
        //      errUART2 = LL_USART_IsActiveFlag_ORE(USART2);
        //----- kasujemy flagi błędów, bez tego wieczne IRQ
        LL_USART_ClearFlag_ORE(USART2); // przepełnie bufora (zgubienie znaku) - można wyłączyć w USART_CR3
        LL_USART_ClearFlag_FE(USART2);  // format ramki (w miejscu stop wartość 0 itp)
        LL_USART_ClearFlag_NE(USART2);  // szumy (w obrębie bitu zmiany sygnału) - można wyłączyć
        // LL_USART_ClearFlag_PE(USART2);// parzystości nie używamy

        //      if( LL_USART_IsActiveFlag_RXNE(USART2) ) {              // Bez FIFO może być IF
        while( LL_USART_IsActiveFlag_RXNE(USART2) ) {   // Z FIFO lepiej WHILE co nie przeszkadza gdy nie
ma FIFO
                rx_buffer[rx_bufferCntWR] = LL_USART_ReceiveData8(USART2);
                if ( ++rx_bufferCntWR >= RX_UART_LEN ) rx_bufferCntWR = 0;
                if( rx_bufferCntWR == rx_bufferCntRD ) FLuartOVR = true;
        }
}

int16_t GetUart(){
        uint16_t z;
        if( rx_bufferCntRD != rx_bufferCntWR ){
                z = rx_buffer[rx_bufferCntRD];
                if( ++rx_bufferCntRD == RX_UART_LEN ) rx_bufferCntRD = 0;
                return z;
        }
        return -1;
}

uint8_t sendUARTtxt(char *txt ){
        if( ! FLtransitTx )     {
                LL_GPIO_SetOutputPin(UART2_SYNC_GPIO_Port, UART2_SYNC_Pin);
                FLtransitTx = true;
                buf_uart_tx_prt = (uint8_t*)txt;
                uart_tx_len = strlen(txt) + 1;
                LL_USART_EnableIT_TXE(USART2);
                return true;
        }
        return false;
}
```

Dzięki fladze „FltransmitTx” dopóki nie skończy się nadawanie poprzedniego komunikatu nie zostanie uruchomione wysyłanie kolejnego. Pozostaje jeszcze w pętli głównej dopisać:

```
		/* USER CODE BEGIN 3 */

		int16_t c;
		while( (c=GetUart()) != -1 ){
			timUARTtx = DEF_TIM_UART_TX << 3;
			char static txtEcho[TXT_ECHO_LEN];
			snprintf(txtEcho, TXT_ECHO_LEN, ANSI_PEN_CYAN"%c"ANSI_PEN_WHITE, c );
			sendUARTtxt(txtEcho);
		}

		if( ! timUARTtx ) {
			timUARTtx = DEF_TIM_UART_TX;
			sendUARTtxt("Test ");
		}
```

4.8.3 DMA

Z nadawaniem związane są funkcje:

```
void irqDmaUartTX(){
	//------------ nadawanie --------------//
	//	if ( LL_DMA_IsActiveFlag_TC1(DMA1) ) {  // TC1 - IRQ od kanału 1
	// Kanał nr 1 nie dzieli z innymi wektora IRQ więc nie trzeba sprawdzać źródła IRQ
	// Nie ma też włączonych IRQ od połowy transfetu więc źródłem może być tylko i wyłąscznie
	//	koniec transferu
	{
		LL_DMA_ClearFlag_TC1(DMA1);
		//LL_DMA_ClearFlag_HT1(DMA1);   // Nie używamy przerwań od połowy transfetu więc kasowanie
nie jest wymagane

		LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_1);
		FLtransitTx = false;
	}
}

uint8_t sendUARTtxt(char *txt ){
	if( ! FLtransitTx ){
		LL_GPIO_SetOutputPin(UART2_SYNC_GPIO_Port, UART2_SYNC_Pin);
		FLtransitTx = true;
		buf_uart_tx_prt = (uint8_t*)txt;
		uart_tx_len = strlen(txt);

		NVIC_EnableIRQ(DMA1_Channel1_IRQn);                    // nr DMA
		LL_DMA_EnableIT_TC(DMA1, LL_DMA_CHANNEL_1);     // i kanał poznamy w CubeMX

		LL_DMA_ConfigAddresses(DMA1, LL_DMA_CHANNEL_1, (uint32_t)buf_uart_tx_prt,
				LL_USART_DMA_GetRegAddr(USART2, LL_USART_DMA_REG_DATA_TRANSMIT),
				LL_DMA_DIRECTION_MEMORY_TO_PERIPH);

		LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_1, uart_tx_len);

		LL_USART_EnableDMAReq_TX(USART2);
		LL_DMA_EnableChannel(DMA1, LL_DMA_CHANNEL_1);
		return true;
	}
	return false;
}
```

Za odbiór odpowiadają:

```
void irqDmaUartRX(){
	//------------ odbior --------------//
	//if (LL_DMA_IsActiveFlag_TC2(DMA1) )           // TC2 - IRQ od kanału 2
	// Mamy aktywny tylko kanał nr 2 więc nie trzeba sprawdzać źródła IRQ
	//  ale gdy aktywujemy kanał 3 bedzie to koneczne
	// Nie ma też włączonych IRQ od połowy transfetu więc źródłem może być tylko i wyłąscznie
	//       koniec transferu
	{
		LL_DMA_ClearFlag_TC2(DMA1);
		//LL_DMA_ClearFlag_HT2(DMA1);    // Nie używamy przerwań od połowy transfetu więc
kasowanie nie jest wymagane
		LL_GPIO_TogglePin(UART2_SYNC_GPIO_Port, UART2_SYNC_Pin);

		LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_2);
		//wyłączamy kanał DMA, aby ponownie ustawić ilość danych do
		// (licznik w rejestrze DMA->CNDTR nie może być zmieniony, gdy kanał DMA jest włączony).
		enableDMArx();

		char static txtEcho[100];
		snprintf(txtEcho, 100, ANSI_PEN_YELLOW""CRLF"Dma: "ANSI_PEN_CYAN"'%s' "ANSI_PEN_WHITE,
rx_buffer );
		sendUARTtxt(txtEcho);
		timUARTtx = DEF_TIM_UART_TX << 3;
	}
}

//--------- inicjalizacja odbioru --------------//
void initUARTrx(){
	NVIC_EnableIRQ(DMA1_Channel2_3_IRQn);                    // nr DMA
	LL_DMA_EnableIT_TC(DMA1, LL_DMA_CHANNEL_2);      // i kanał poznamy w CubeMX

	LL_DMA_ConfigAddresses(DMA1, LL_DMA_CHANNEL_2,
			LL_USART_DMA_GetRegAddr(USART2, LL_USART_DMA_REG_DATA_RECEIVE),
			(uint32_t)rx_buffer,
			LL_DMA_DIRECTION_PERIPH_TO_MEMORY );
	LL_USART_EnableDMAReq_RX(USART2);
	enableDMArx();
}


void enableDMArx(){
	uint16_t len = UARTRX_DMALEN;
	DMArxLen = len;
	LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_2, len );
	LL_DMA_EnableChannel(DMA1, LL_DMA_CHANNEL_2);
}
```

W pętli głównej nic nie trzeba zmieniać. Ze względu na to, że odbiór danych przez DMA z UART nie jest zbyt wygodny najlepiej go połączyć z przerwaniem od końca ramki o czym w następnym punkcje.

4.8.4 Przerwanie od IDLE (końca ramki danych)

Aby zaprezentować jak działa przerwanie od końca ramki w przerwaniu od UART umieszczamy kod:

```
        if( LL_USART_IsActiveFlag_IDLE(USART2) ){
                LL_USART_ClearFlag_IDLE(USART2);
                LL_GPIO_TogglePin(UART2_SYNC_GPIO_Port, UART2_SYNC_Pin);


                LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_2);
                uint16_t len = DMArxLen - LL_DMA_GetDataLength(DMA1, LL_DMA_CHANNEL_2);
                //wyłączamy kanał DMA, aby ponownie ustawić ilość danych do
                // (licznik w rejestrze DMA->CNDTR nie może być zmieniony, gdy kanał DMA jest
włączony).
                enableDMArx();


                if( len ){
                        rx_buffer[len] = 0;
                        char static txtEcho[100];
                        snprintf(txtEcho, 100, ANSI_PEN_YELLOW""CRLF"DI: "ANSI_PEN_CYAN"'%s'
"ANSI_PEN_WHITE, rx_buffer );
                        sendUARTtxt(txtEcho);
                        timUARTtx = DEF_TIM_UART_TX << 3;
                }
        }
}
```

natomiast w „initUARTrx()” dopisujemy:

```
    LL_USART_ClearFlag_IDLE(USART2);
    LL_USART_EnableIT_IDLE(USART2);
```

Rozdział 5. Przerwania zewnętrzne EXTI

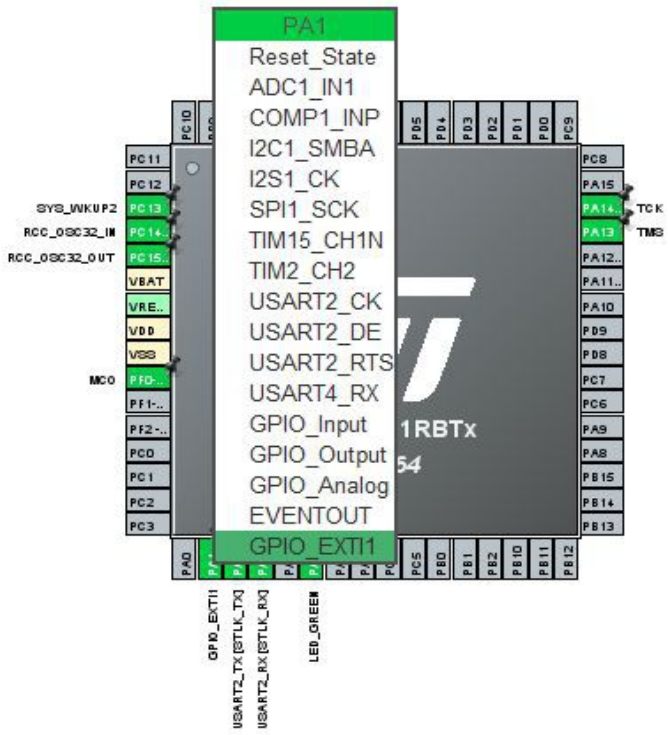
Wyzwalanie zboczem narastającym, opadającym lub obydwoma. Nie ma jak w AVR czy 8051 albo Z80, 6502 itp. poziomem co czasem może być problemem.

5.1 Zliczanie impulsów (prosty pomiar małych częstotliwości)

Zaprezentowany sposób wykorzystania przerwań nie jest doskonały. Większą rozdzielczość uzyskamy gdy naszym licznikiem nie będzie timer systemowy lecz inny taktowany wyższą częstotliwością a najlepiej jeśli będzie on sprzętowo zliczał impulsy ale o tym będzie w rozdziale poświęconym timerom gdzie zbudujemy najprawdziwszy licznik częstotliwości i czasu. Maksymalna częstotliwość zliczania zależy od taktowania CPU gdy korzystamy ze sprzętowego liczenia przez timery od ich taktowania.

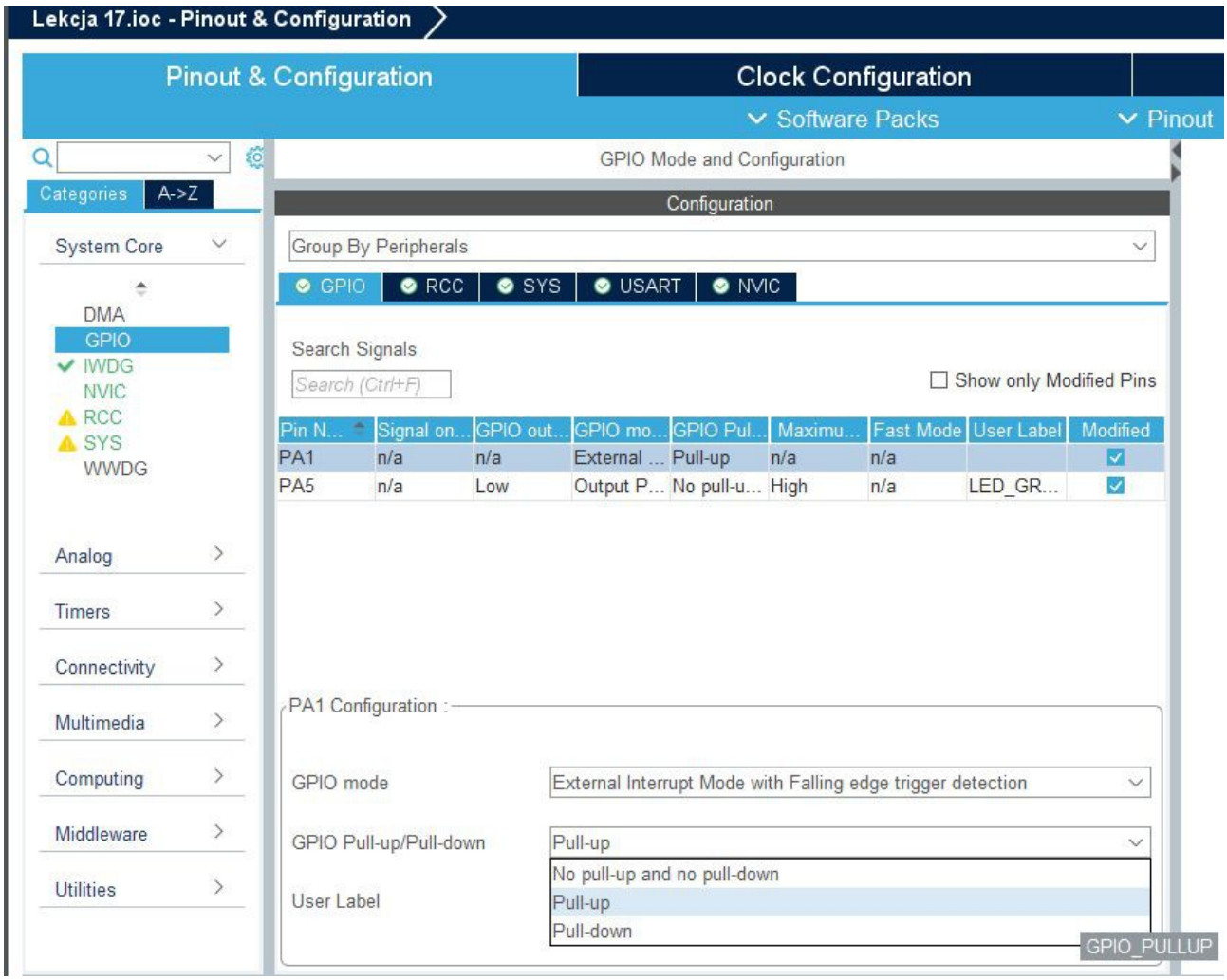
Jeśli obawiamy się, że złe ustawienia generatora mogą uszkodzić wejście STM32 to można szeregowo z wejściem wstawić rezystor 1...10k.

Konfigurujemy wyprowadzenie PA1 w roli wejście przerwań:



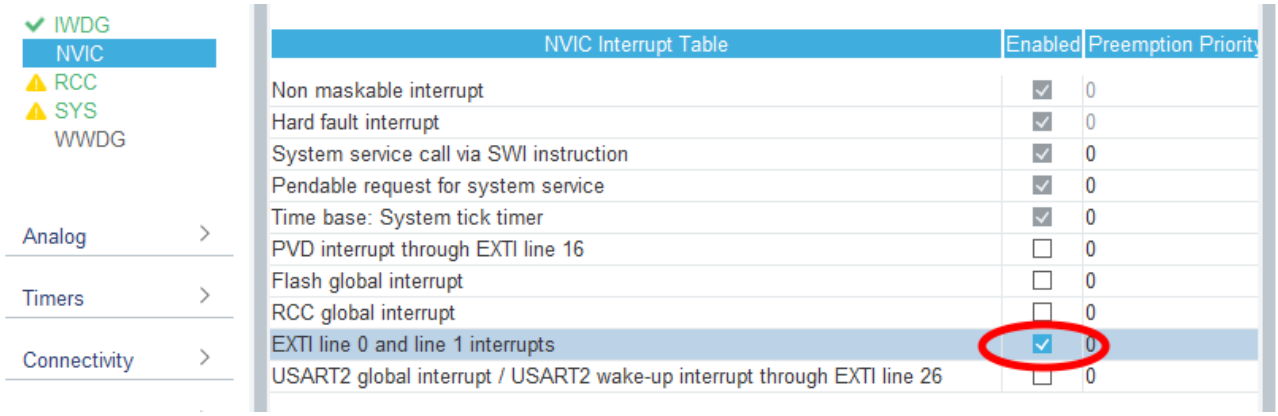
Rysunek 1701

Warto włączyć podciąganie można też wybrać „GPIO mode” czy ma reagować na zbocze narastające czy opadające:



Rysunek 1702

i coś o czym łatwo zapomnieć, trzeba włączyć przerwania w NVIC:



Rysunek 1703

W przypadku peryferiów przerwania włączane są automatycznie ale dla wejść EXTI nie. **Program 17** http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja17.zip jest dosyć prosty, dodajemy funkcje:

```c
/* USER CODE BEGIN 0 */

//void HAL_GPIO_EXTI_Rising_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_Rising_Callback could be implemented in the user file
//   */
//}

uint32_t volatile cntExti1, fExti;
void HAL_GPIO_EXTI_Falling_Callback(uint16_t GPIO_Pin){
      if( GPIO_Pin == GPIO_PIN_1 ) cntExti1++;
}


uint16_t volatile timUartTx, timLed;
void HAL_IncTick(void){
      uwTick += (uint32_t)uwTickFreq;

      if( timUartTx ) timUartTx--;
      if( timLed ) timLed--;

      uint16_t static tim;
      if( ++tim >= 1000 ){
            fExti = cntExti1;

            tim = cntExti1 = 0;
      }
}
```

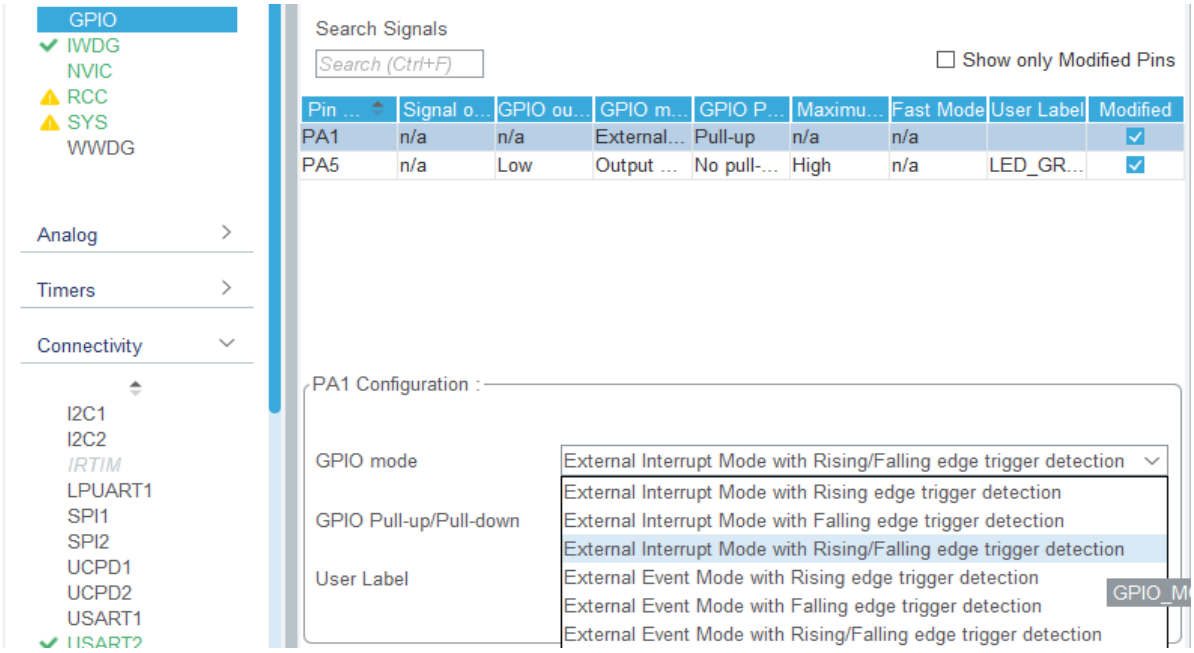
Natomiast w pętli głównej:

```c
  /* USER CODE BEGIN 3 */
            if( ! timLed ){
                  timLed = 500;
                  HAL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
            }

            if( ! timUartTx ){
                  timUartTx = 1000;
                  char static cnt, txt[50];
                  sprintf(txt, "%ld f=%ld,%03d kHz\r\n", ++cnt, fExti/1000, fExti % 1000 );
                  HAL_UART_Transmit_IT(&huart2, (uint8_t*)txt, strlen(txt) );
            }
```

Film 17 https://youtu.be/PDr-aT74p-0 prezentuje pomiar, na STM32 16MHz udało się uzyskać pomiar 200 kHz, 64MHz ponad 600kHz. Używając bibliotek LL wynik będzie lepszy ale metoda pomiaru jaką pokazałem jest zła bo przy pomiarze dużych częstotliwości CPU jest bardzo obciążony i przerwania mogą „zawiesić” program główny.

5.2 Prosty pomiar czasu impulsu

Jak w poprzednim przypadku najlepiej wykorzystać sprzętowy timer. Konfiguracja CubeMX jest prawie taka sama jak w poprzednim przypadku, jedyna różnica to reakcja PA1 na oba zbocza:



Rysunek 1801

W programie pokażę istotne zmiany, w funkcjach:

```
/* USER CODE BEGIN 0 */

uint32_t volatile cntExti1, fExti, cH, cL, cntH, cntL;
void HAL_GPIO_EXTI_Rising_Callback(uint16_t GPIO_Pin){
	if( GPIO_Pin == GPIO_PIN_1 ){
		cntL = cL;
		cH = 0;
	}
}


void HAL_GPIO_EXTI_Falling_Callback(uint16_t GPIO_Pin){
	if( GPIO_Pin == GPIO_PIN_1 ){
		cntH = cH;
		cL = 0;
		cntExti1++;
	}
}


uint16_t volatile timUartTx, timLed;
void HAL_IncTick(void){
	uwTick += (uint32_t)uwTickFreq;

	cH++; cL++;
```

oraz pętli głównej:

```
sprintf(txt, "%d f=%ld,%03ld kHz H=%ld ms L=%ld ms\r\n", ++cnt, fExti/1000, fExti % 1000, cntH, cntL );
HAL_UART_Transmit_DMA(&huart2, (uint8_t*)txt, strlen(txt) );
```

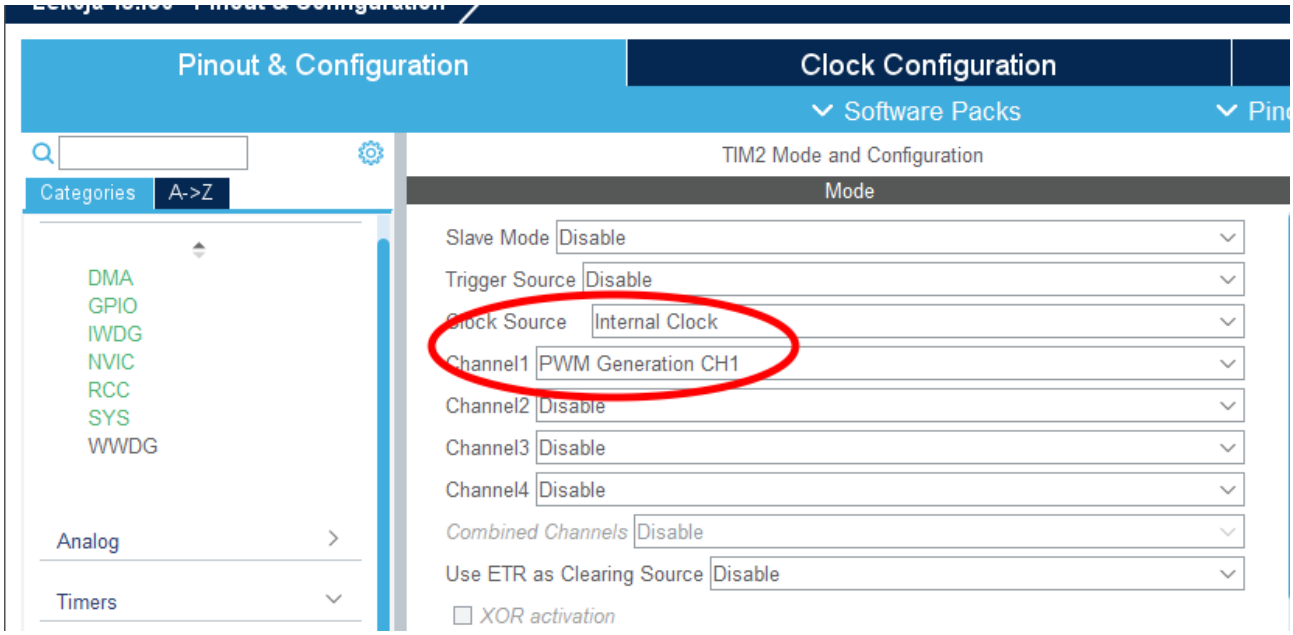
Kod **projektu 18a** jest do pobrania z:

http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja18a.zip

Efekt działania programu można obejrzeć w **filmie 18a**: https://youtu.be/MfCuW_s8rDM

5.3 Generowanie sygnału do pomiaru częstotliwości i czasu

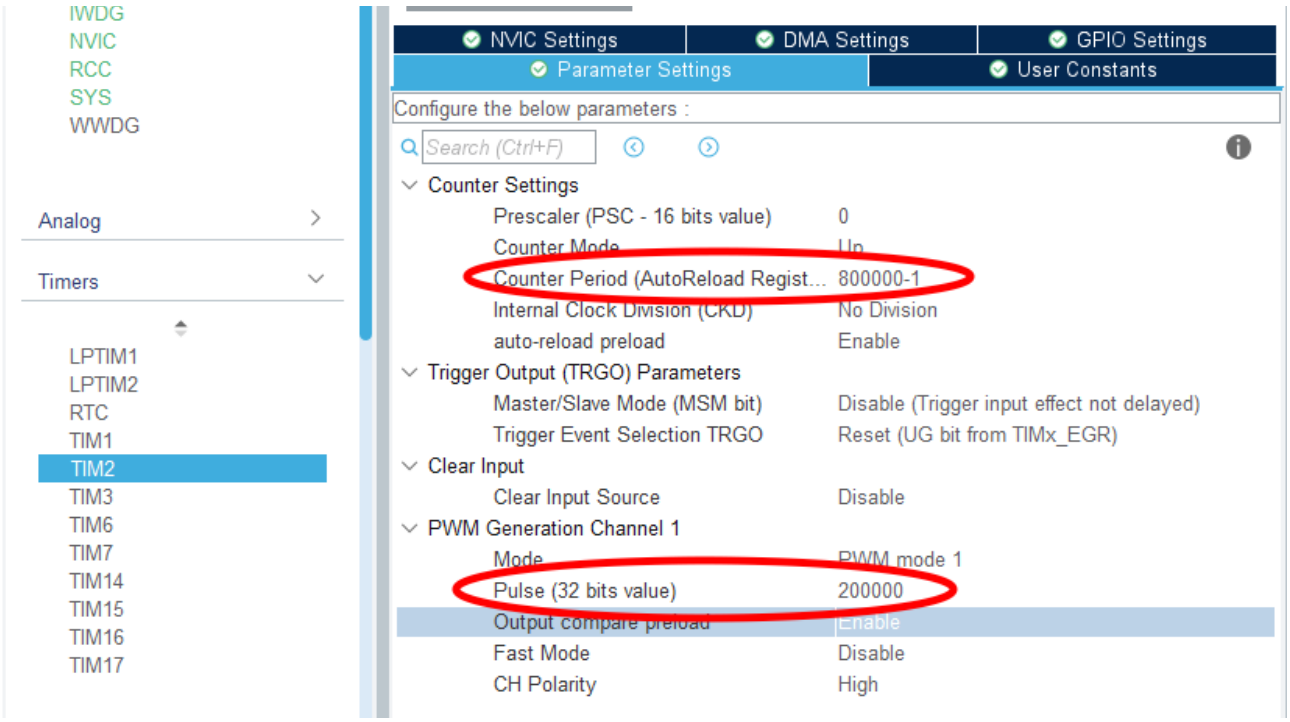
Jeśli nie posiadamy generatora możemy go zrobić z naszego STM32. Co prawda dotyczy to timerów ale bez wdawania się w szczegóły zrealizujemy prosty generator sygnału prostokątnego o regulowanym wypełnieniu i częstotliwości. Aby zrealizować cel należy uaktywnić timer. Wybrałem TIM2 wyjście CH1 bo jest to pin PA0 obok PA1, którym mierzymy sygnał.



Rysunek 1802

Ustawimy jeszcze częstotliwość 20Hz. Ustawienie rejestru ARR wynika z:

16000000(Fcpu) / 20(Hz) = 800000. Wpisujemy wartość o jeden mniejszą bo timer liczy od zera:



Rysunek 1803

Wypełnienie jest ustawione na 1/4 (200000). W kodzie dodajemy tylko jedną linię:

```
/* USER CODE BEGIN 2 */
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
```

Która to uaktywnia timer. Teraz wystarczy zewrzeć PA0 z PA1 zworką i można testować program.

Kod **projektu 18b** jest do pobrania z:

http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja18b.zip

Efekt działania programu można obejrzeć w **filmie 18b**: https://youtu.be/6S2Dy4MG2W4

5.4 Przerwanie EXTI od układów peryferyjnych

Zagadnienie będzie rozwinięte w kolejnych rozdziałach, na razie zachęcam do obejrzenia filmów:

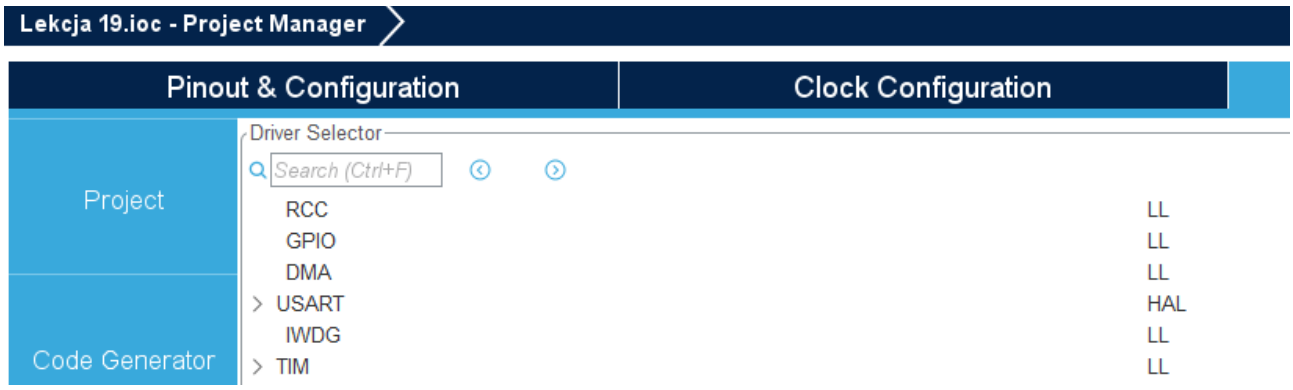
https://youtu.be/cK4KFeLFCI4

https://youtu.be/0bsMJ_D79K4

https://youtu.be/c6yiF7CjZzc

5.5 Zliczanie impulsów z wykorzystaniem LL

Dla miłośników zwięzłych i przewidywalnych kodów obsługa EXTI z wykorzystaniem LL. Projekt tworzymy jak w poprzedniej lekcji (IWDG, EXTI, TIM2, UART) ale trzeba przełączyć się z HAL na LL poza UART.



Rysunek 1901

W pliku „stm32g0xx_it” w handlerze EXTI dopisujemy:

```
void EXTI0_1_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI0_1_IRQn 0 */

      extern void irqEXTI();
      irqEXTI();
```

w „main.c”:

```c
/* USER CODE BEGIN 0 */

uint32_t volatile cntExti1, fExti;
void irqEXTI(){
	if (LL_EXTI_IsActiveFallingFlag_0_31(LL_EXTI_LINE_1) != RESET) cntExti1++;
}

uint16_t volatile timLed, timUartTx;
void HAL_IncTick(void){
	uwTick += (uint32_t)uwTickFreq;


	if( timLed ) timLed--;
	if( timUartTx ) timUartTx--;

	uint16_t static tim;
	if( ++tim >= 1000 ){
		fExti = cntExti1;

		tim = cntExti1 = 0;
	}
}

void initIrqTimSys(){
	LL_SYSTICK_EnableIT();  //  @paplociennik
	//LL_Init1msTick( LL_RCC_GetSystemClocksFreq() );

	//LL_Init1msTick( LL_RCC_GetSystemClocksFreq() );

	/*
	//SysTick_Config(SystemCoreClock/1000);
	//SysTick_Config(SysTick->LOAD);
	SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;
	NVIC_EnableIRQ(SysTick_IRQn);
	 */
}
/* USER CODE END 0 */


/* USER CODE BEGIN 2 */
initIrqTimSys();

LL_TIM_CC_EnableChannel(TIM2, LL_TIM_CHANNEL_CH1);
LL_TIM_EnableCounter(TIM2);

{
	char const txt[] = "***** Start ***\r\n";
	HAL_UART_Transmit(&huart2, (uint8_t*)txt, strlen(txt), 10 );
}
```

Pętla główna wygląda tak:

```
	/* USER CODE BEGIN WHILE */
	while (1)  {
		LL_IWDG_ReloadCounter(IWDG);
		__WFI();

		/* USER CODE END WHILE */

		/* USER CODE BEGIN 3 */
		if(  ! timLed ){
			timLed = 500;
			LL_GPIO_TogglePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin);
		}


		if( ! timUartTx ){
			timUartTx = 1000;
			char static cnt, txt[50];
			sprintf(txt, "%d f=%ld,%03ld kHz\r\n", ++cnt, fExti/1000, fExti % 1000 );
                        HAL_UART_Transmit_IT(&huart2, (uint8_t*)txt, strlen(txt) );
			//HAL_UART_Transmit(&huart2, (uint8_t*)txt, strlen(txt), 10 );
		}

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

Kod **projektu 19** jest do pobrana z: http://er-mik.prv.pl/Ksiazka_Programowanie_STM32/Lekcja19.zip

Film 19 dostępny na: https://youtu.be/koqzYlHVkJM

Wspierający powstanie książki

Podmioty gospodarcze

https://kamami.pl

https://msalamon.pl

Osoby prywatne

Artur Furmanek, Paweł, Paweł Woźniak, Pitaszek, Grzegorz Wójcik, Marcin Wójcik, Tomasz Bednarek, Grzegorz Sosnowski, Maciej Fiedorowicz, Daniel Kukuła, Karol Daniszewski, Cezary Zawadzki, Andrzej Brzezina, Paweł Rawel, Marcin Robaczewski, Robert Loboda, Łukasz Pytel, Artur Skoneczny, John Doe, Mikołaj Ograbek, Janusz Sosnowski, Mikołaj Majewski, Paweł Jax, Mikołaj Ograbek, Piotr Grzonka, Sebastian Horodecki, Paweł Matuszewski, Aleksander K, Arkadiusz Bagiński, Marcin Woźnica, Jakub R, Emil Wach, Krzysztof Szwaba, Robert Płoucha, Tomasz Grabiec, Mariusz K

Okładka 2

O autorze:

Od 2002 roku zajmuje się zawodowo projektowaniem urządzeń i pisaniem programów na nie. Urządzenia wykorzystują głównie mikrokontrolery czasem mikroprocesory i/lub układy programowalne GAL, CPLD, FPGA. Od 1997 do 2022 roku współpracował z wydawnictwem AVT publikując blisko 250 artykułów. Publikował także w „Magazynie Amiga", „Praktycznym Elektroniku", „Hobby Elektroniku". Od 1991 pisał programy i budował urządzenia dla Commodore 64, od 1993 dla Amigi i MON. Pierwsze recenzje pojawiły się w 1993 roku w „Commodore & Amiga" artykuł w 1991 roku w „Praktycznym Elektroniku", kiedy to miał 17 lat w chwili publikacji (projekt z 1989 roku). Hobbystycznie zajmuje się modelarstwem (głównie kolejowym) z naciskiem na elektronikę. Efektem hobby jest tani dekoder trakcji i dźwięku sterowany przez DCC http://kolejki.prv.pl/dekoderySTM32 oraz urządzenia SRK (sterowania ruchem kolejowym). Od 2019 roku prowadzi kanał na YouTube https://youtube.com/@eR-MIK o tematyce elektronicznej poruszający głównie zagadnienie związane z mikrokontrolerami, mikroprocesorami i układami programowalnymi.

Kontakt z autorem: sas.ze@vp.pl

Kanał na YouTube: https://youtube.com/@eR-MIK

Strony internetowe autora:
http://er-mik.prv.pl/ - strona PHP/HTML bez użycia kreatorów (tylko edytor tekstu ASCII strona stworzona jeszcze na Amidze)
http://es2.noip.pl - strona na serwerze autora
http://kolejki.prv.pl/ - tematyka modelarstwa kolejowego

Portfolio autora:
http://er-mik.prv.pl/projekty_avt.php - projekty dla AVT, MA, PE, HE, BIW, FET od 1991 roku
http://er-mik.prv.pl/projekty_edw.php - projekty dla AVT po 2017 roku
http://er-mik.prv.pl/projekty_ze.php - projekty dla „Zrozumieć elektronikę"