KURS STM32

Wojciech Olech

CZĘŚĆ I: WSTĘP, ŚRODOWISKO, OBSŁUGA PŁYTKI

PLATFORMA STM32 VS AVR

Architektury

- ARM - Advanced RISC Machine, 32 i 64 bitowa architektura, szeroko stosowana w systemach wbudowanych i systemach o niskim poborze mocy.
- W porównaniu z AVR, znacznie bardziej zaawansowana, energooszczędna i wydajna.
- **ARMy typowo działają na logice 3.3V, AVRy na 3.3V lub 5V**

STM32 vs ATMega

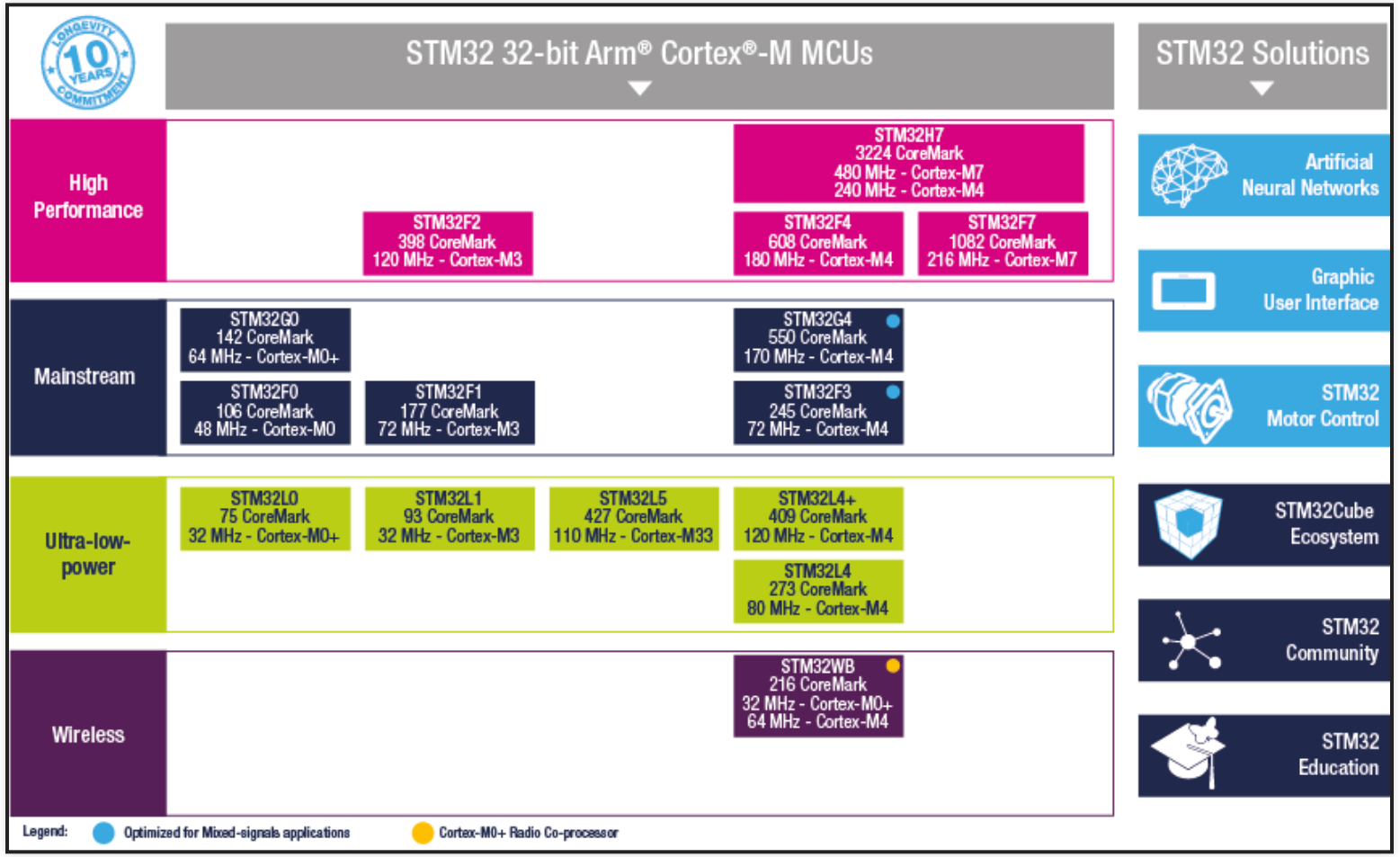
- STMy mają o wiele więcej wbudowanych peryferiów
- STMy mogą pracować na znacznie wyższych zegarach (ATMegi typowo 8/16MHz, STMy typowo 32-480MHz (w zależności od serii), plus lepsze IPC = znacznie większa wydajność)
- STMy mają kilka razy lepszy stosunek wydajności/możliwości do ceny
- Obie platformy są kompatybilne z Arduino*
- STMy mają bardzo dobre wsparcie wielu środowisk oraz frameworków (między innymi RTOS - Real Time Operating System)

`* w dużej części, niekoniecznie w 100%`

WBUDOWANE PERYFERIA DOSTĘPNE W WIĘKSZOŚCI STMÓW

- DMA (Direct Memory Access) - kontroler pozwalający na wykonywanie operacji I/O bez udziału CPU
- GPIO - piny I/O na logice 3.3V (część pinów ma tolerancję 5V)
- Watchdog
- ADC (przetwornik analogowo-cyfrowy), 12-bitowy
- RTC (Real-Time Clock)
- Timery (proste i zaawansowane, z możliwością generowania sygnału PWM)
- Magistrale komunikacyjne:
  - CAN
  - I2C
  - SPI
  - USART
  - USB
  - Różne pochodne powyższych (I2S, SMBus, IRDA, SmartCard)
- Hardware'owe CRC

SERIE STM32


LONGEVITY 10 YEARS COMMITMENT
STM32 32-bit Arm® Cortex®-M MCUs
STM32 Solutions
High Performance
STM32H7
3224 CoreMark
480 MHz - Cortex-M7
240 MHz - Cortex-M4
STM32F2
398 CoreMark
120 MHz - Cortex-M3
STM32F4
608 CoreMark
180 MHz - Cortex-M4
STM32F7
1082 CoreMark
216 MHz - Cortex-M7
Mainstream
STM32G0
142 CoreMark
64 MHz - Cortex-M0+
STM32G4
550 CoreMark
170 MHz - Cortex-M4
STM32F0
106 CoreMark
48 MHz - Cortex-M0
STM32F1
177 CoreMark
72 MHz - Cortex-M3
STM32F3
245 CoreMark
72 MHz - Cortex-M4
Ultra-low-power
STM32L0
75 CoreMark
32 MHz - Cortex-M0+
STM32L1
93 CoreMark
32 MHz - Cortex-M3
STM32L5
427 CoreMark
110 MHz - Cortex-M33
STM32L4+
409 CoreMark
120 MHz - Cortex-M4
STM32L4
273 CoreMark
80 MHz - Cortex-M4
Wireless
STM32WB
216 CoreMark
32 MHz - Cortex-M0+
64 MHz - Cortex-M4
Artificial Neural Networks
Graphic User Interface
STM32 Motor Control
STM32Cube Ecosystem
STM32 Community
STM32 Education
Legend:
Optimized for Mixed-signals applications
Cortex-M0+ Radio Co-processor

OPROGRAMOWANIE

- **ZALECANE** - STM32CubeIDE - IDE zintegrowane z generatorem projektów na podstawie którego będzie prowadzony ten kurs.
  Link do pobrania: https://bit.ly/36Nfde3

ALTERNATYWY

- STM32CubeMX - generator projektów i graficzny konfigurator mikrokontrolera w postaci osobnego programu - może generować projekty dla różnych IDE
  Link do pobrania: https://bit.ly/35aVMvr
- Atollic TrueSTUDIO - środowisko programistyczne, na którym zostało oparte STM32CubeIDE. Nierozwijane.
  Link do pobrania: https://bit.ly/2NRH3Qa

POPULARNE BIBLIOTEKI/FRAMEWORKI DO OBSŁUGI STM32

- **CMSIS** - *Cortex Microcontroller Software Interface Standard*, biblioteka standardowa dla mikrokontrolerów opartych o ARM która pełni funkcję warstwy abstrakcji dla sprzętu, napisana w C. Jest bazą dla wszystkich bibliotek pisanych pod ARMy
- **STM32 HAL** - *Hardware Abstraction Layer*, standardowa biblioteka dla STM32 którą będziemy się posługiwać. Pozwala na łatwe używanie mikrokontrolera. Napisana w C.
- **Arduino** - Najpopularniejsze płytki i procesory STM32 mają wsparcie Arduino*
- **RTOS** - *Real Time Operating System*, framework umożliwiający tworzenie wielowątkowych programów na mikrokontrolery.
- **MBed** - Framework dla urządzeń embedded/IoT na architekturze ARM ze wsparciem RTOSa, wspiera C++a. Dobra alternatywa dla Arduino.

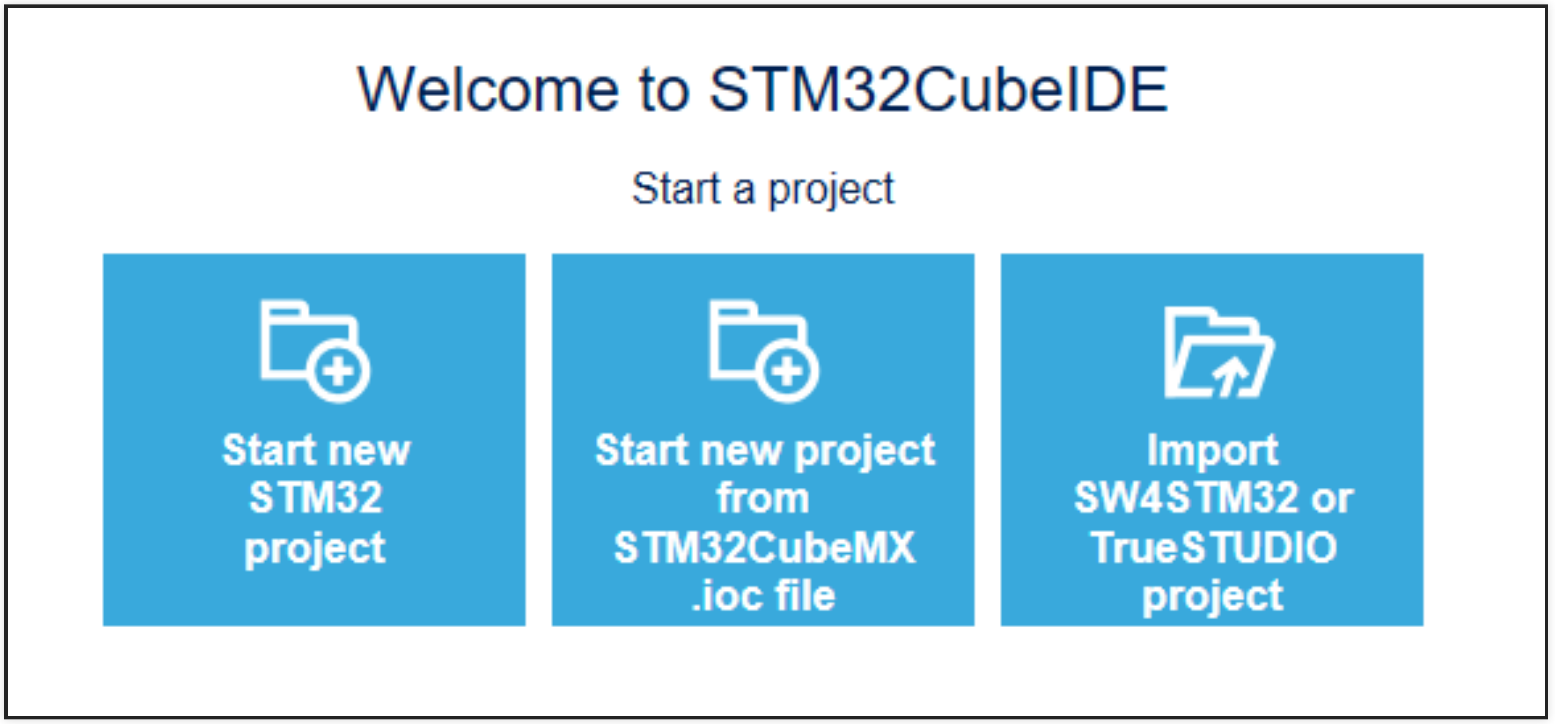
`* Nie wszystkie biblioteki Arduinowe będą działać na STM32, ponieważ niektóre mają wstawki dla AVRów lub korzystają z peryferiów niedostępnych na STMach (EEPROM)`

STM32CUBEMX

- Narzędzie do konfigurowania mikrokontrolera oraz generowania projektu dla dowolnego wspieranego IDE
- Umożliwia bardzo łatwe tworzenie projektów dla STMów i płytek Nucleo/Discovery
- Zintegrowane z STM32CubeIDE

TWORZENIE PROJEKTU

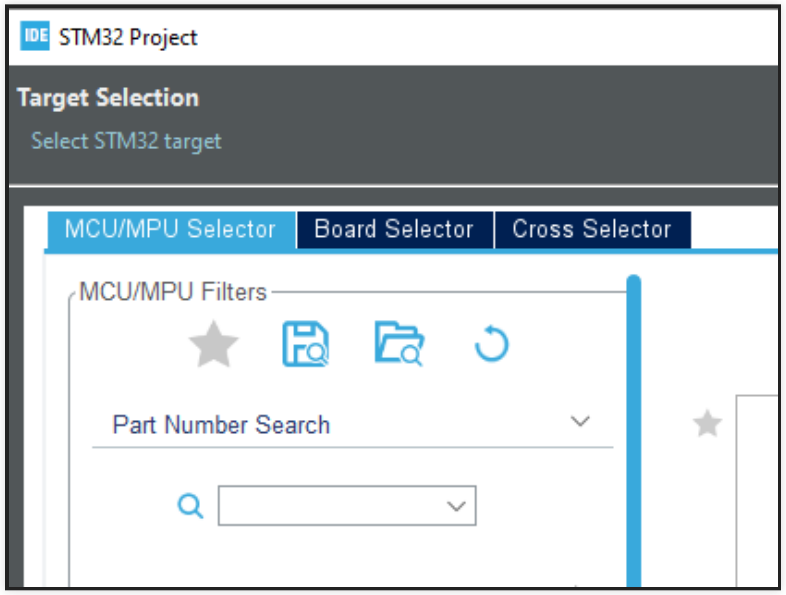
Żeby stworzyć nowy projekt w STM32CubeIDE, otwieramy Information Center i wybieramy opcję "Start new STM32 project", lub w menu File -> New wybieramy opcję "STM32 Project"



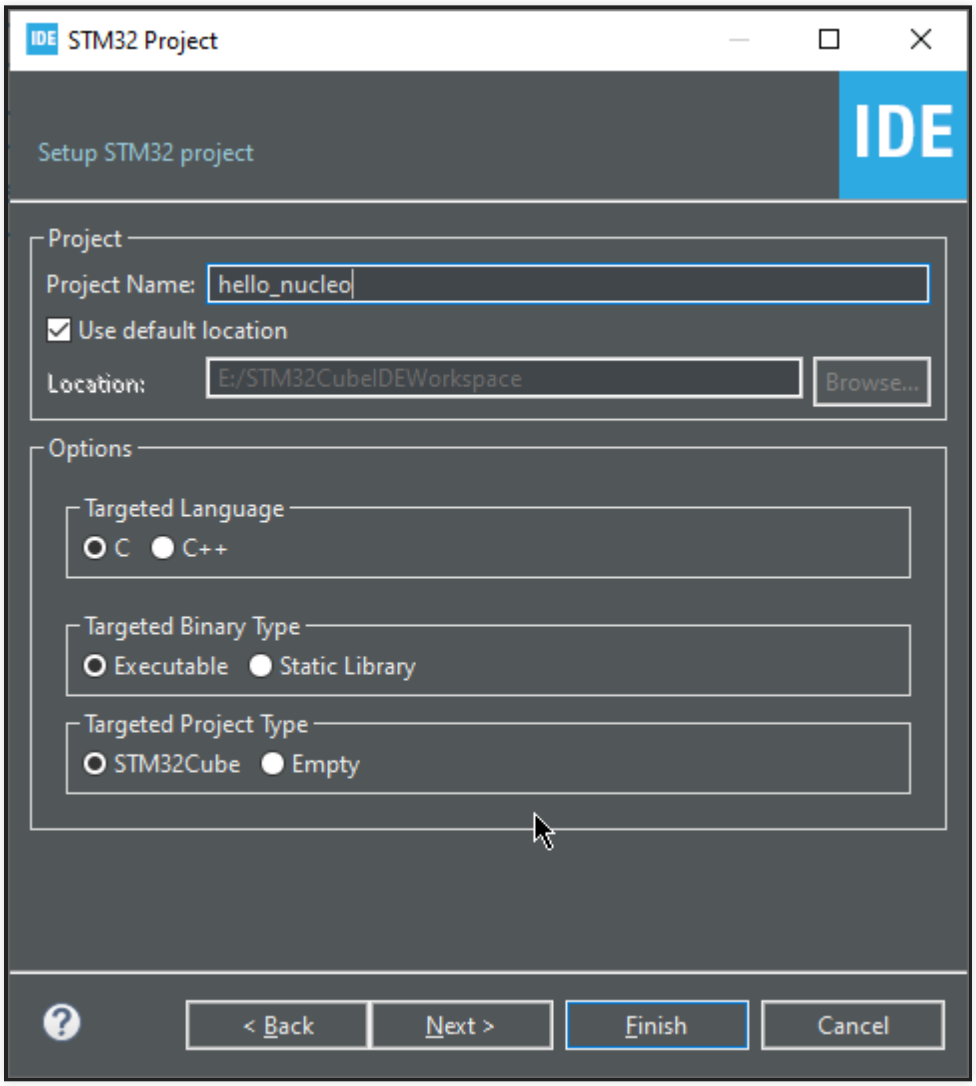
W oknie które się pojawiło możemy wybrać jedną z trzech zakładek:

- MCU/MPU Selector - jeśli tworzymy projekt pod MCU na naszej własnej, lub customowej płytce, interesuje nas ta zakładka
- Board Selector - jeśli mamy płytkę od ST (przykładowo Discovery lub Nucleo), to w tej zakładce znajdziemy szablony projektów dla tych płytek
- Cross Selector - pozwala dobrać MCU od ST na podstawie MCU innego producenta

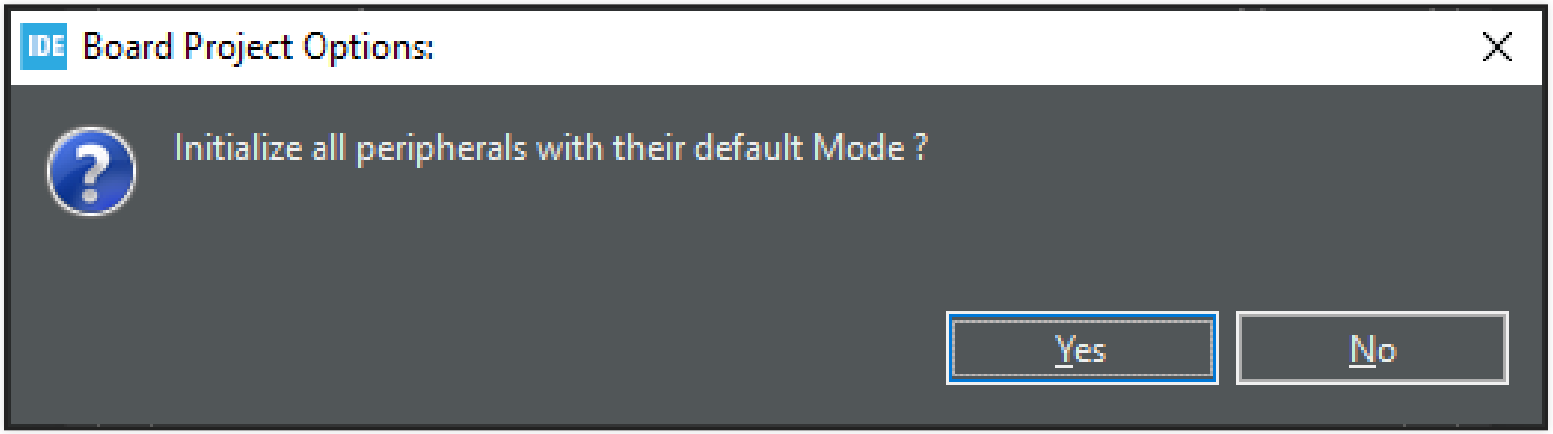
Nas interesuje zakładka "Board Selector". W pole wyszukiwania wpisujemy model naszego Nucleo



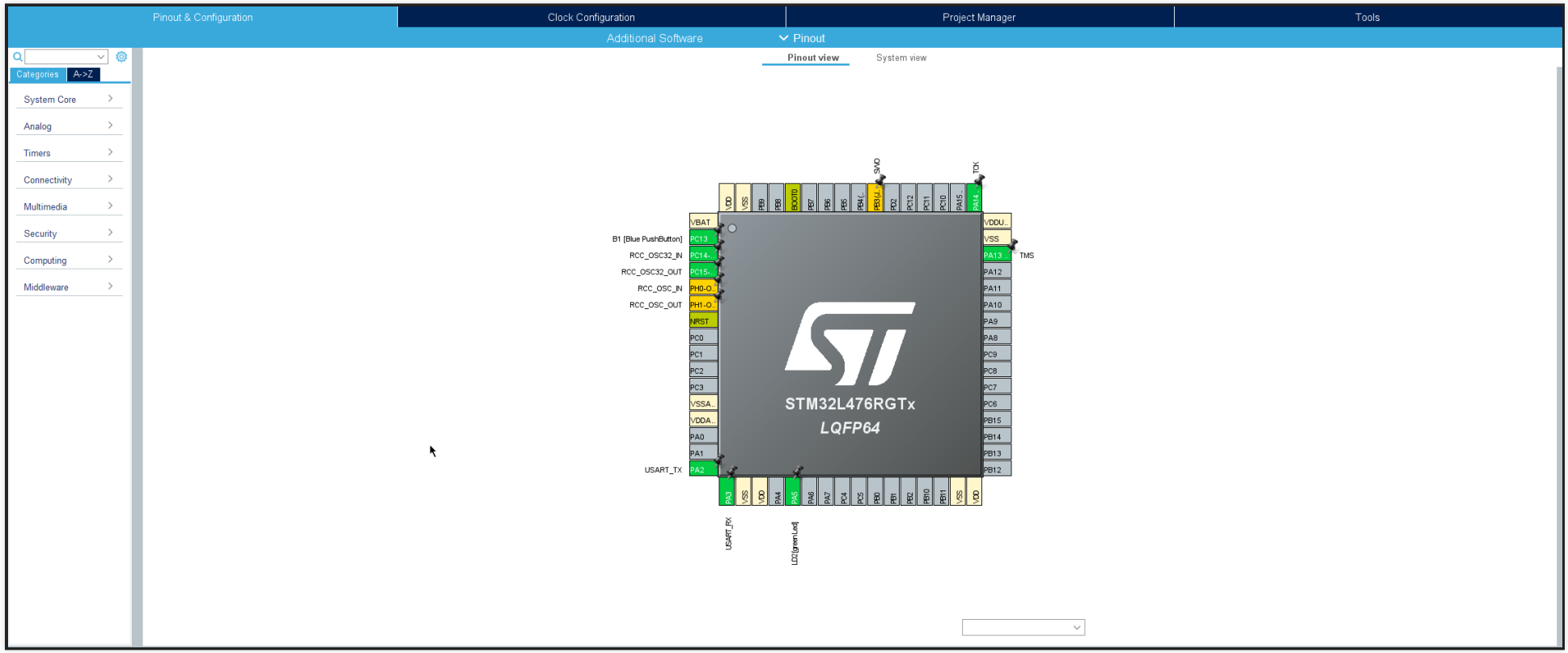
Po wybraniu płytki i kliknięciu "Next >" wyświetli nam się okno z wyborem lokalizacji projektu, języka i typu projektu. Zmieniamy tylko nazwę, pozostałe ustawienia zostawiamy domyślnie, po czym klikamy "Finish".


STM32 Project
Setup STM32 project
IDE
Project
Project Name: hello_nucleo
Use default location
Location: E:/STM32CubeIDEWorkspace
Browse...
Options
Targeted Language
C C++
Targeted Binary Type
Executable Static Library
Targeted Project Type
STM32Cube Empty
< Back
Next >
Finish
Cancel

Jeśli tworzymy projekt na płytkę ST, to pojawi nam się pop-up z pytaniem czy chcemy ustawić peryferia na domyślne tryby - klikamy "Yes".



Powinnien się automatycznie otworzyć następujący widok w STM32CubeIDE, z poziomu którego możemy konfigurować peryferia procesora oraz jego poszczególne piny.

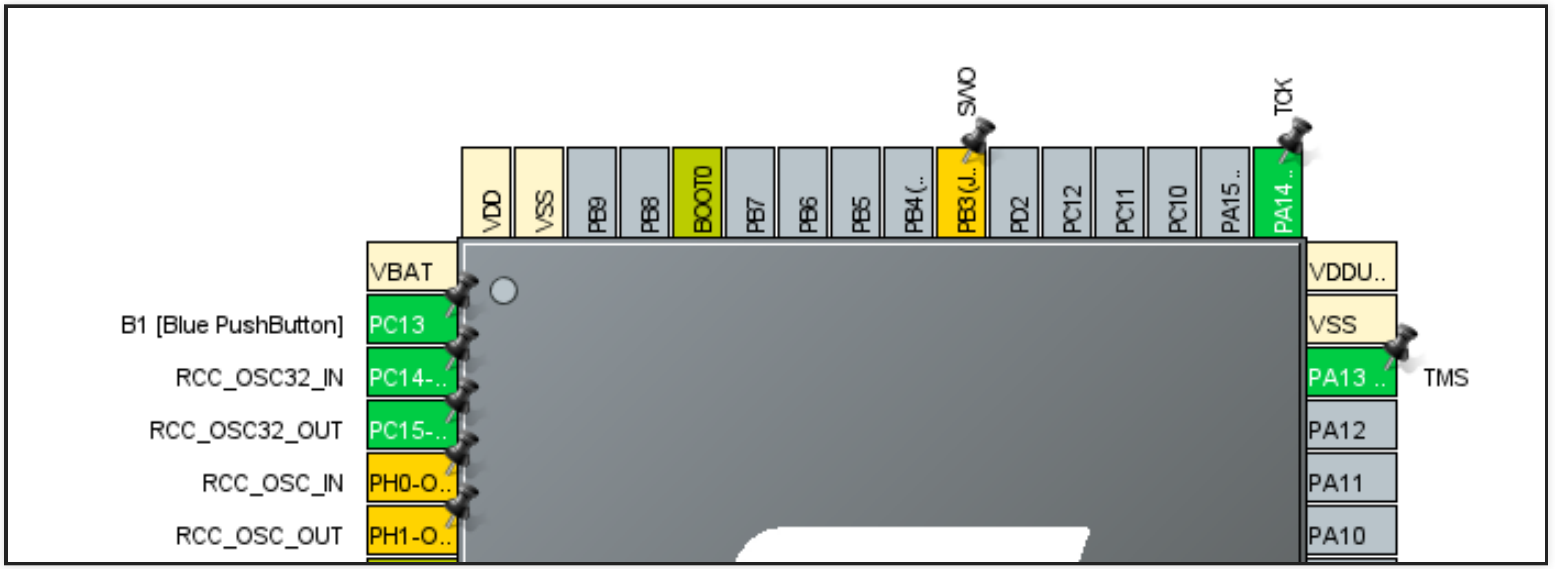


Po lewej stronie mamy pogrupowane peryferia zgodnie z ich kategoriami

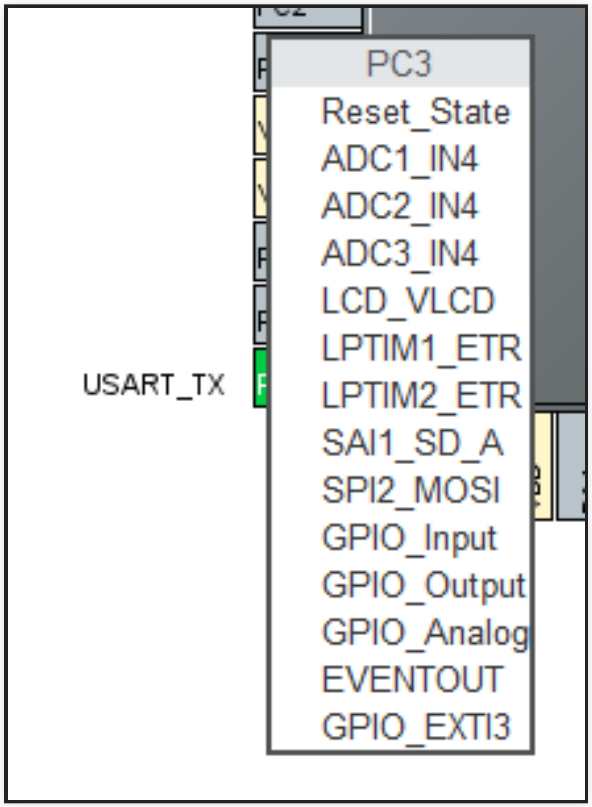
- System Core - główne peryferia systemowe, m. in. kontroler zegarów, przerwań, GPIO
- Analog - peryferia analogowo/cyfrowe: ADC, DAC, wzmacniacze operacyjne, komparatory
- Timers - timery, RTC i timery niskiej mocy
- Connectivity - magistrale danych - UART, USART, SPI, I2C, CAN, USB, SDMMC
- Multimedia - magistrale danych do peryferiów multimedialnych - LCD, SAI, I2C
- Security - peryferia powiązane z bezpieczeństwem - RNG, szyfrowanie
- Computing - peryferia wspomagające obliczenia - CRC, filtry
- Middleware - dodatkowe biblioteki, przykładowo RTOS, FATFS, USB

Na obrazku procesora możemy zauważyć kilka różnych kolorów pinów.

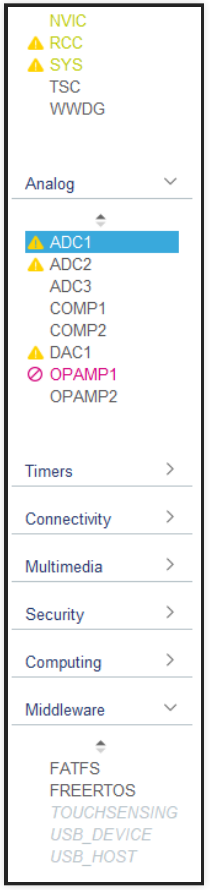
- Szary - nieużywany
- Kremowy (VBAT/VDD/VSS) - linia zasilania procesora
- Limonkowy (BOOT0) - piny zarezerwowane przez niskopoziomowe mechanizmy (boot/reset)
- Pomarańczowy - piny potencjalnie zarezerwowane przez peryferia które nie są jeszcze skonfigurowane
- Zielony - skonfigurowane i używane przez dane peryferia piny



Żeby skonfigurować dany pin, możemy na niego kliknąć i wybrać jedną z wielu funkcjonalności do których będzie używany. Zazwyczaj nie wystarczy jednak tylko tutaj zaznaczyć jego trybu, ponieważ w większości przypadków należy również skonfigurować ustawienia peryferii (jeśli pin zaświeci się na pomarańczowo).



Kolorystyczne oznaczenia pojawiają się również na liście peryferiów



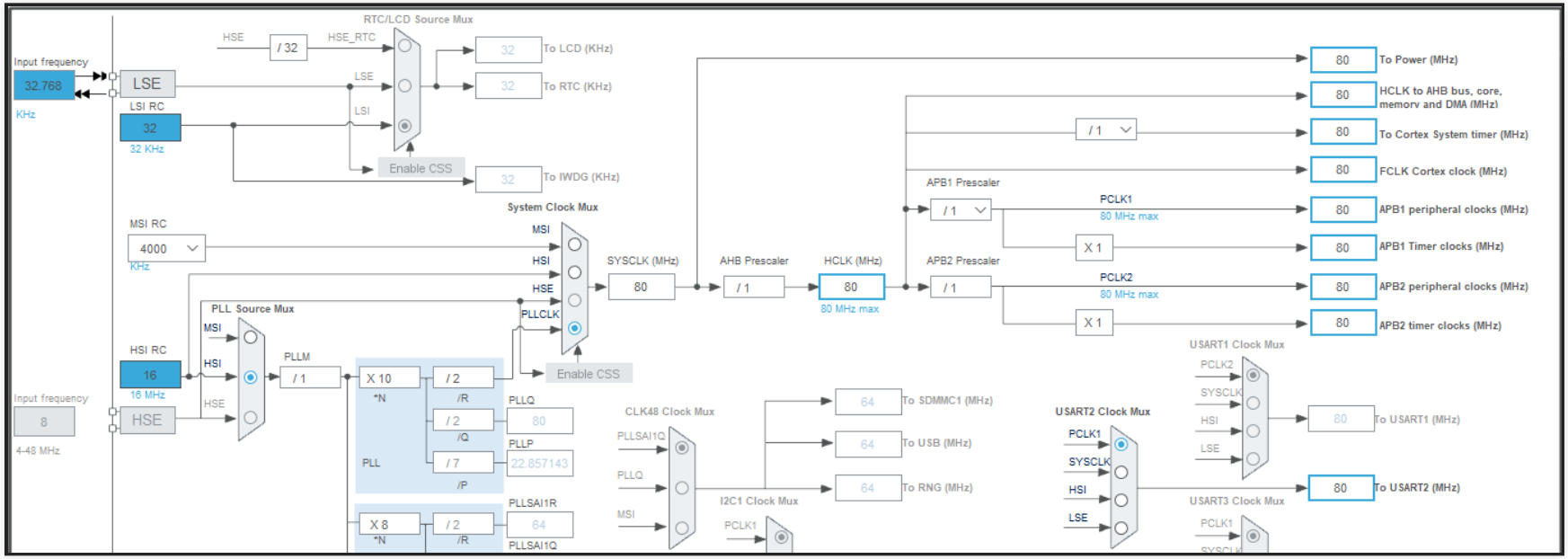
- Zielone - skonfigurowane poprawnie
- Żółty trójkąt - część ustawień jest niedostępna przez konflikty z innymi peryferiami
- Czerwone - peryferium niedostępne przez konflikt z innymi peryferiami
- Czarne - peryferium nieskonfigurowane
- Szare - peryferium niedostępne przez brak konfiguracji innego peryferium

KLUCZOWE PERYFERIA SYSTEMOWE

Kluczowymi peryferiami do których warto zaglądać przy tworzeniu każdego projektu są **RCC** (Reset and Clock Control) i **SYS** (System)

W RCC możemy ustawić z jakich źródeł zegara ma korzystać nas procesor. W SYS możemy m.in. zmienić tryb debugowania i bazowy zegar "tików" naszego programu.

W zakładce "Clock Configuration" mamy pokazane drzewo zegarów dostępnych dla naszego procesora.



Na ten moment nie musimy się nim przejmować, automatycznie jest ono ustawione tak żeby wszystkie peryferia działały.
W niektórych przypadkach jednak będziemy chcieli zmienić częstotliwości taktowania procesora lub peryferiów - wystarczy wtedy wpisać wartość w żądane pole i CubeMX automatycznie przeliczy dzielniki i mnożniki tak, żeby uzyskać dany zegar, lub zwróci błąd jeśli nie będzie to możliwe.

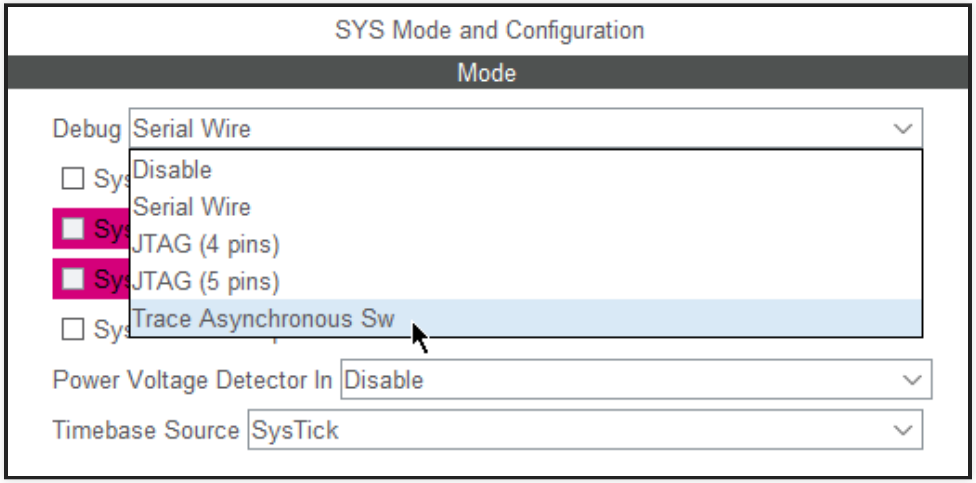
STMy posiadają kilka głównych źródeł zegarowych.

- **HSI** - High Speed Internal, szybki zegar wewnętrzny (używany do taktowania rdzenia i peryferiów)
- **LSI** - Low Speed Internal, wolny zegar wewnętrzny (używany do taktowania RTC)
- **HSE** - High Speed External, zewnętrzny odpowiednik HSI
- **LSE** - Low Speed External, zewnętrzny odpowiednik LSI
- **System Clock Mux** - Multiplekser który pozwala na programowy wybór źródła zegara taktującego procesor
- **PLL Source Mux** - Pętla synchronizacji fazy, pozwalająca na zwiększanie częstotliwości zegara

CubeMX automatycznie dostosowuje ustawienia PLL i źródła jeśli podamy mu częstotliwość zegara w polu z niebieską ramką.

KONFIGURACJA MIKROKONTROLERA POD DEBUGGING

Podstawowy debugging możemy skonfigurować w zakładce System Core -> Sys

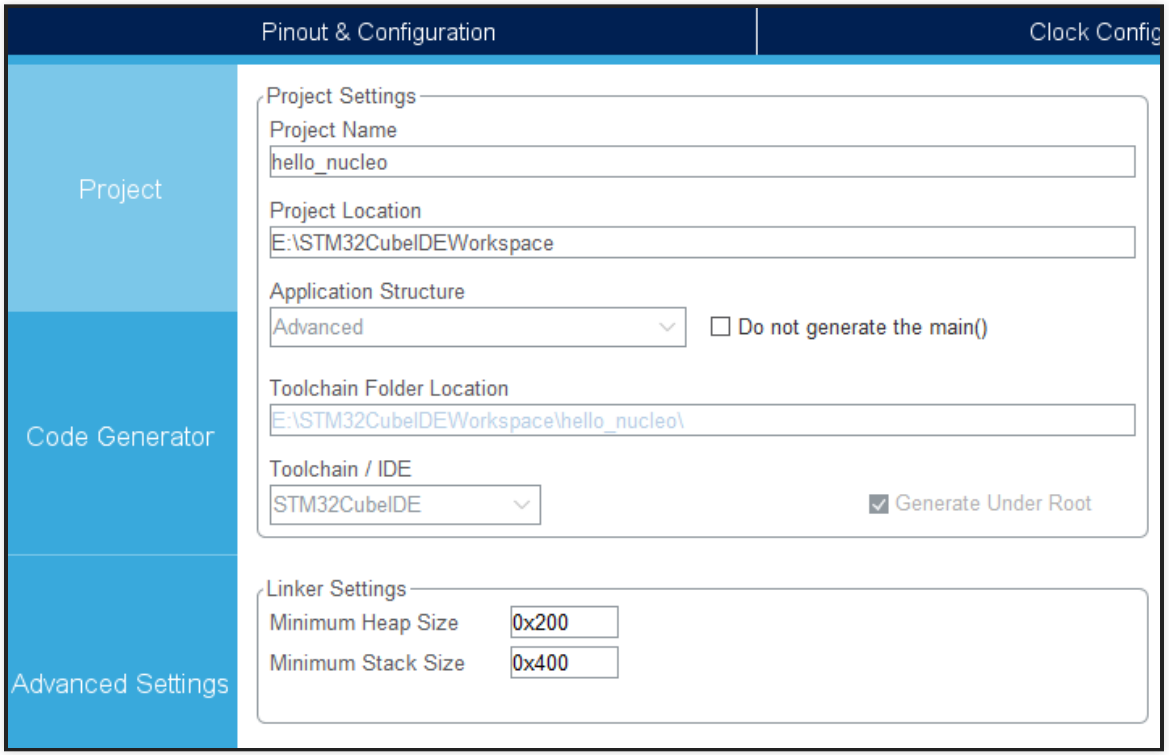


- Serial Wire - najprostszy tryb debugowania pozwalający na podgląd pamięci i chodzenie po kodzie krok po kroku
- Trace Asynchronous Sw - bardziej zaawansowany tryb debugowania, niedostępny w STM32L0, STM32G0 i STM32F0 (wymagany Cortex-M3 lub mocniejszy), pozwala między innymi na komunikację poprzez debuger z procesorem (wysyłanie tekstu z procesora do debugera)
- **Debugowanie musi być skonfigurowane żeby można było używać pełnych możliwości debugera - jeśli tego nie zrobimy i spróbujemy debugować, debuger wyrzuci błąd (ale program zostanie uploadowany na procesor)**

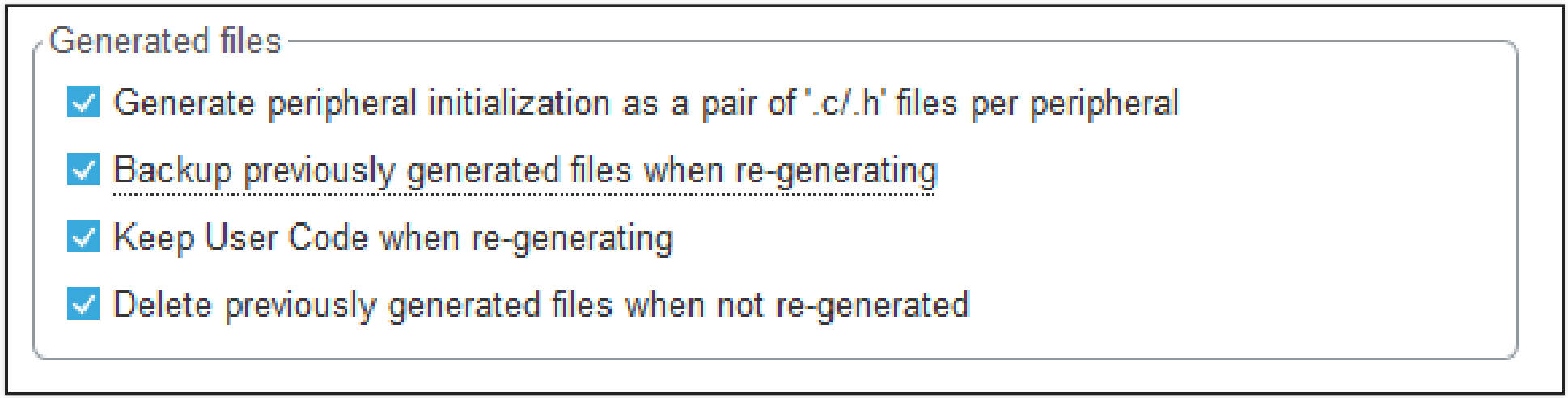
USTAWIENIA PROJEKTU

W zakładce **Project Manager -> Project** możemy ustawić

1. Nazwę projektu
2. Lokalizację
3. Minimalną wielkość stosu/sterty programu



Dodatkowo, w podzakładce **Code Generation** możemy kazać Cube'owi generować pliki inicjalizacyjne jako para nagłówek+źródło i robić backup kodu przy każdym generowaniu. Polecam zaznaczyć następujące opcje:



Kod automatycznie zostaje wygenerowany przy zapisie. Po każdym zapisie projektu w widoku CubeMX'a zostaje także re-generowany.

Należy zwracać uwagę na to, w jakich miejscach piszemy kod - CubeMX w plikach którymi zarządza generuje sekcje oznaczone jako komentarze w stylu C, przykładowo

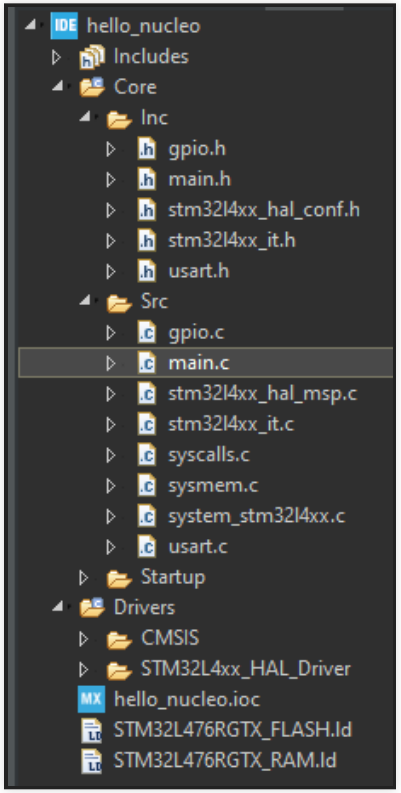
```
/* USER CODE BEGIN 2 */

/* USER CODE END 2 */
```

Kod musimy umieszczać w tych sekcjach, żeby nie został **permanentnie usunięty** po re-generowaniu kodu. Polecam korzystać z systemu kontroli wersji (Git).

STRUKTURA PROJEKTU

Wygenerowany projekt powinien wyglądać mniej-więcej w ten sposób:



Najbardziej interesują nas pliki

- `main.c` - w tym pliku jest główna funkcja programu, `main`
- `stm32f4xx_it.c` - w tym pliku są funkcje przerwań

Oprócz nich, mamy również pliki w których są inicjowane i deklarowane uchwyty do obiektów peryferii oraz niskopoziomowych elementów uC. W katalogu `Drivers` mamy niskopoziomowe sterowniki procesora - CMSIS oraz HAL

HELLO, WORLD!

Zadanie 1: napisać program który będzie migał diodką.

Funkcje które należy wykorzystać: `HAL_GPIO_TogglePin`, `HAL_Delay`

`HAL_GPIO_TogglePin` przyjmuje dwa argumenty: port i pin portu którego chcemy zmienić stan
`HAL_Delay` przyjmuje jeden argument: czas w milisekundach

Przydatne skróty klawiszowe:

- **Ctrl+Shift+F** - Formatowanie kodu
- **Ctrl+Tab** - Przełącz między nagłówkiem a plikiem źródłowym
- **Ctrl+Space** - Autopodpowiadanie

SOLUCJA: ZADANIE 1

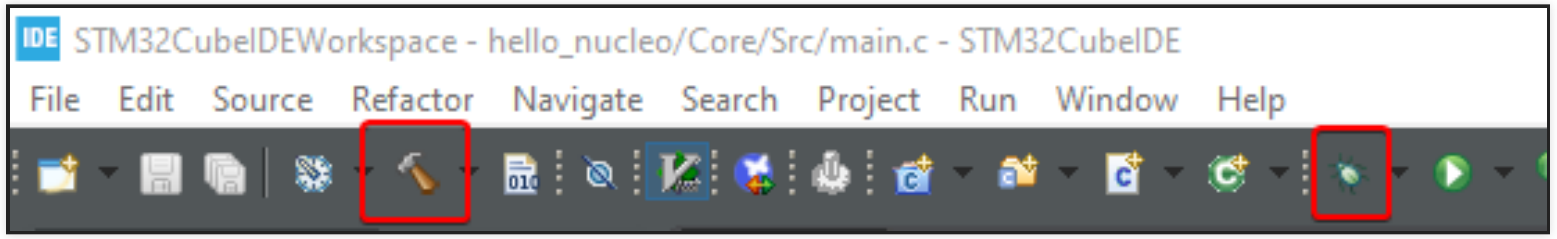
```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1) {
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    HAL_Delay(500);
    /* USER CODE END WHILE */

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

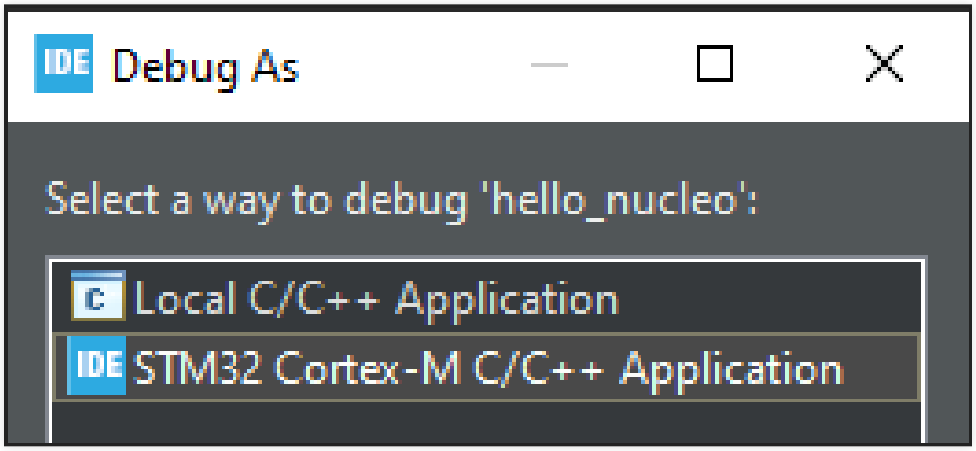
PODSTAWY DEBUGOWANIA

STM32CubeIDE umożliwia upload i debugowanie programu na mikrokontrolerze.

Żeby uruchomić program i rozpocząć debugging, należy wcisnąć **F11** lub przycisk z "robaczkiem" na górnym pasku. Można uprzednio zbudować program (**Ctrl+B** lub "młotek" na górnym pasku) żeby sprawdzić czy nie ma w nim błędów.

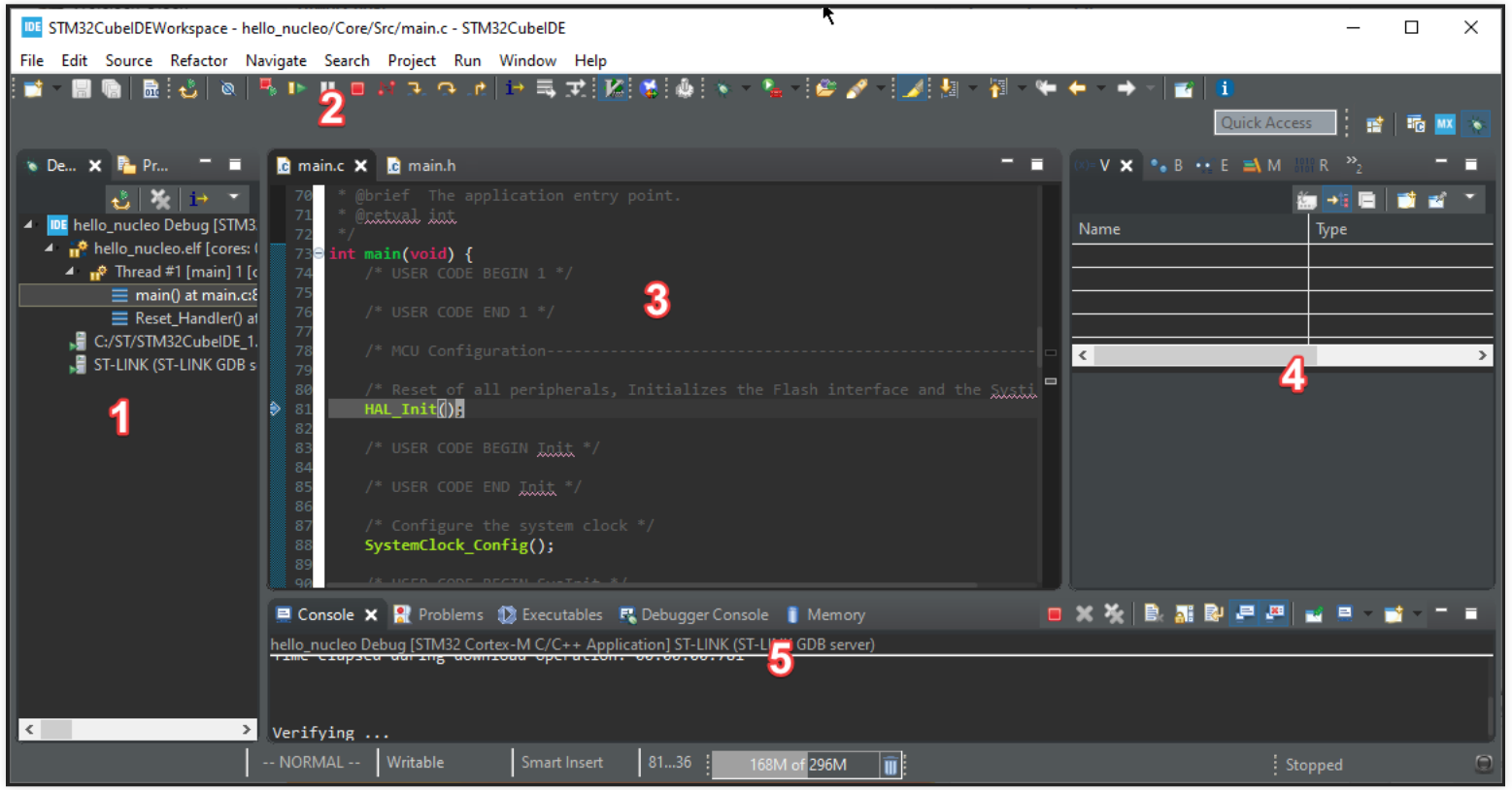


Po kliknięciu *Debug* może nam się pokazać okienko z wyborem trybu debugowania. Wybieramy **STM32 Cortex-M C/C++ Application**



Następnie zostanie wyświetlone okienko z ustawieniami debugowania - nic tutaj nie zmieniamy, klikamy **OK**

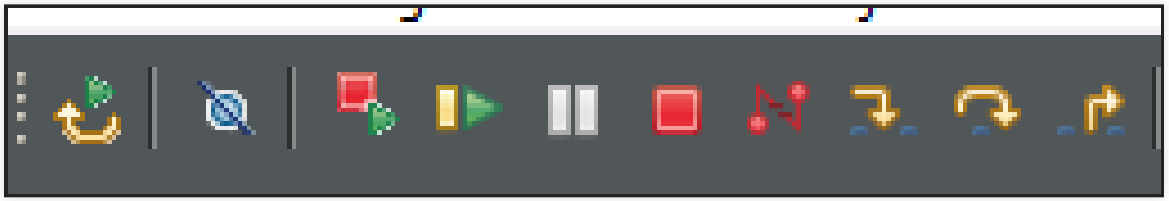
CubeIDE powinno przejść do widoku debugowania, który prezentuje się następująco:



1. Widok wątków procesora oraz stosu wywołań funkcji
2. Pasek z narzędziami
3. Widok kodu
4. Toolbar z tabami do debugowania - m. in. widok zmiennych w aktualnym zakresie, breakpointów, rejestrów i peryferiów
5. Konsola i logi debugowania

STEROWANIE DEBUGOWANIEM

Domyślnie, program automatycznie zatrzyma się na pierwszej linijce funkcji `main`.
W pasku na górze mamy kilka przycisków do sterowania przebiegiem debugowania.



Od lewej:

- Restartuj chip
- Pomiń wszystkie breakpointy (**Ctrl+Alt+B**)
- Pełny restart procesora i debugowania
- Kontynuuj do następnego breakpointa (**F8**)
- Zatrzymaj program
- Zakończ sesję debugowania i zrestartuj procesor (**Ctrl+F2**)
- Rozłącz się z debugerem
- Wejdź do linijki na której procesor stoi (**F5**)
- Przejdź nad linijką na której procesor stoi (**F6**)
- Wróć do wyższego zakresu (funkcji która wywołała funkcję w której procesor teraz jest) (**F7**)

Żeby rozpocząć działanie programu, klikamy "Kontynuuj do następnego breakpointa" (**F8**)

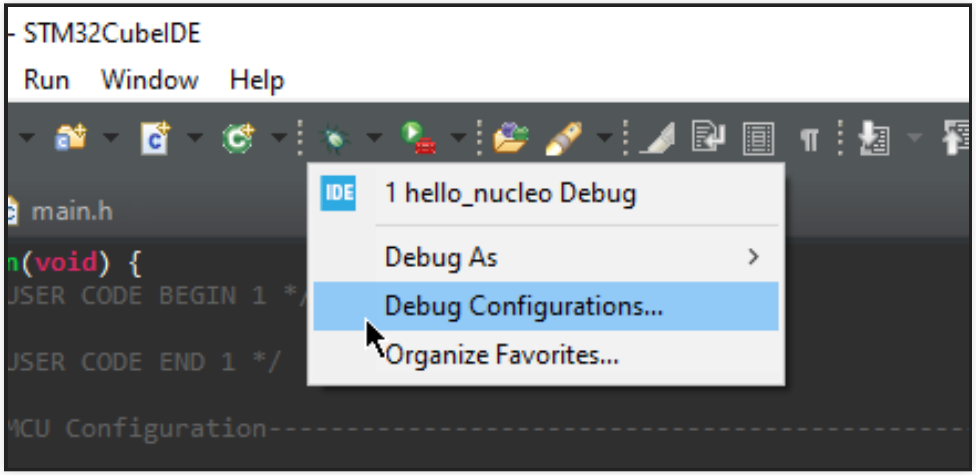
KOMUNIKACJA POPRZEZ DEBUGGER

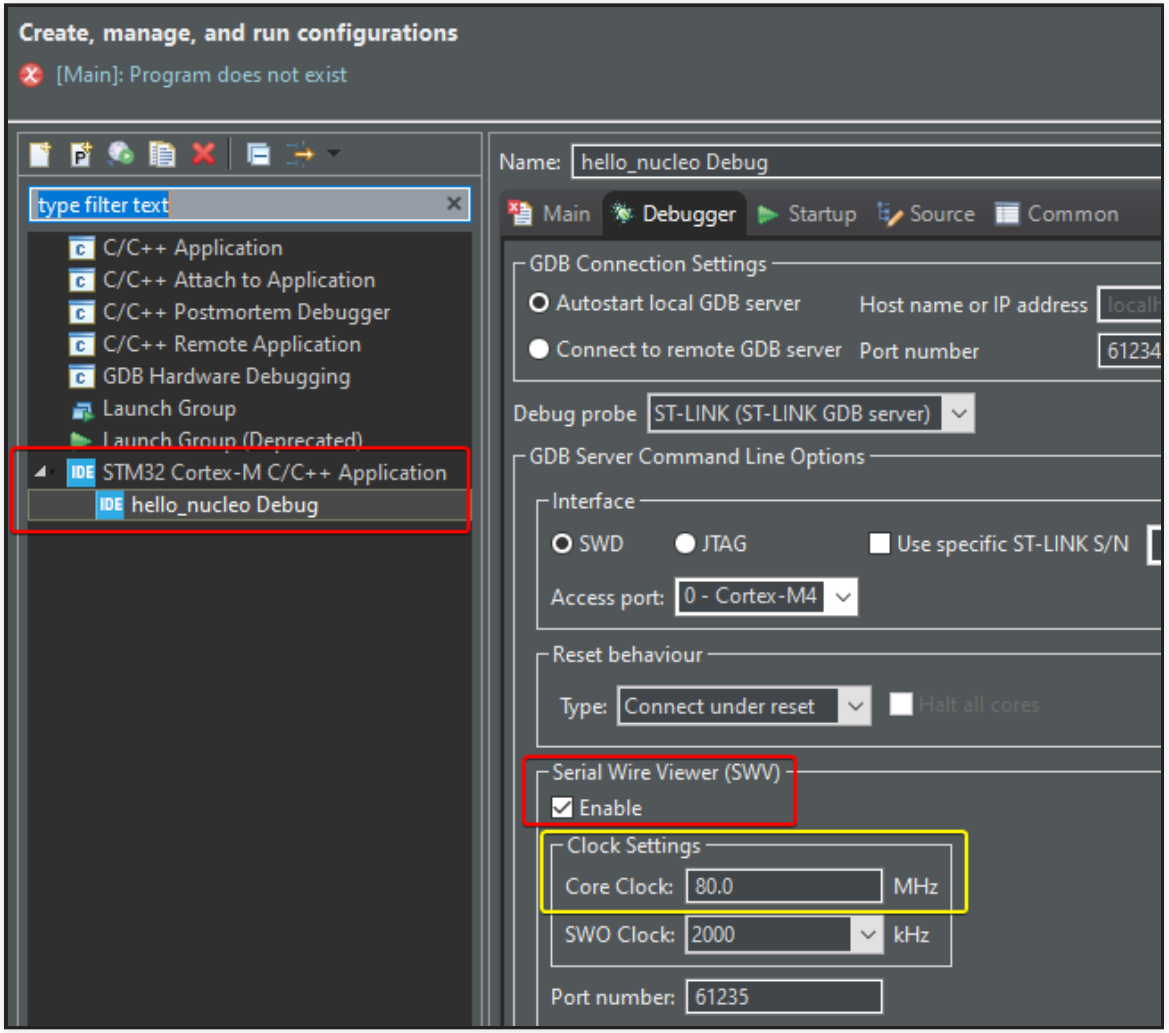
Mikrokontrolery ARM z serii Cortex-M3, Cortex-M4 i Cortex-M7 posiadają moduł ITM (Instrumented Trace Macrocell), który razem z SWO (Serial Wire Output) pozwala na tracing i komunikację z mikrokontrolerem poprzez debuger. ITM posiada 32 kanały, z czego dwa są zarezerwowane dla CMSIS: 0 do wysyłania danych, 31 dla RTOSa.

My będziemy wykorzystywać kanał 0 jako nasze standardowe wyjście, co pozwoli nam wygodnie używać funkcji `printf` do wyświetlania komunikatów podczas debugowania.

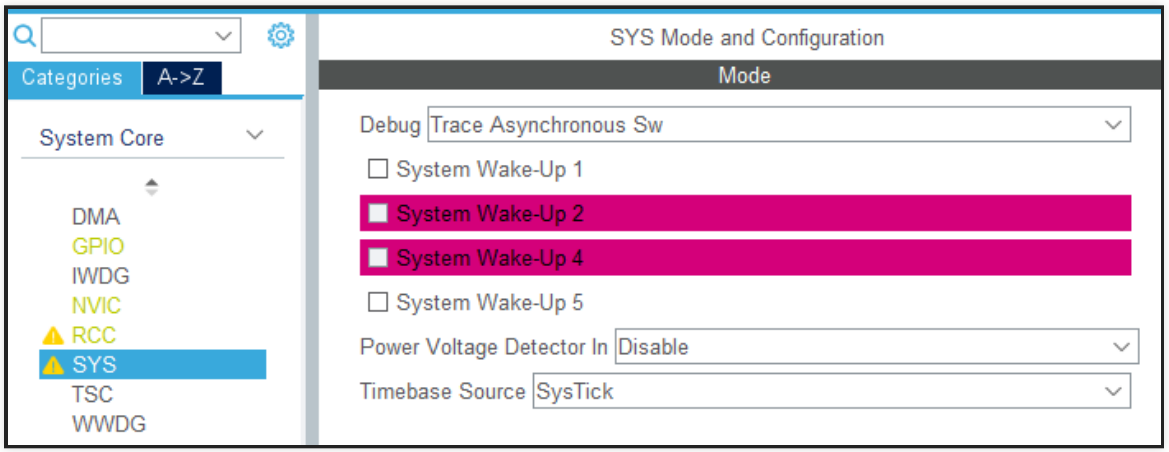
Żeby przechwytywać dane z ITMa w naszym IDE, należy włączyć SWV (Serial Wire Viewer) w ustawieniach debugowania.

Wchodzimy w "Debug -> Debug Configurations", następnie wybieramy naszą konfigurację i przechodzimy do zakładki "Debugger". Zaznaczamy tam checkboxa "Enable" w sekcji "Serial Wire Viewer" i poprawiamy "Core Clock" na zegar naszego mikrokontrolera (HCLK) (można sprawdzić go w zakładce "Clock Configuration" w Device Configuration Toolu - jeśli go zamknęliśmy, trzeba dwa razy kliknąć na plik *.ioc w drzewku projektu)




Create, manage, and run configurations
[Main]: Program does not exist
type filter text
C/C++ Application
C/C++ Attach to Application
C/C++ Postmortem Debugger
C/C++ Remote Application
GDB Hardware Debugging
Launch Group
Launch Group (Deprecated)
STM32 Cortex-M C/C++ Application
hello_nucleo Debug
Name: hello_nucleo Debug
Main
Debugger
Startup
Source
Common
GDB Connection Settings
Autostart local GDB server
Host name or IP address
Connect to remote GDB server
Port number
61234
Debug probe
ST-LINK (ST-LINK GDB server)
GDB Server Command Line Options
Interface
SWD
JTAG
Use specific ST-LINK S/N
Access port:
0 - Cortex-M4
Reset behaviour
Type:
Connect under reset
Halt all cores
Serial Wire Viewer (SWV)
Enable
Clock Settings
Core Clock:
80.0
MHz
SWO Clock:
2000
kHz
Port number:
61235

Należy również upewnić się, że w Device Configuration Toolu wybraliśmy tryb debugowania "Trace Asynchronous Sw" - można to zmienić w zakładce "System Code -> SYS -> Debug", domyślnie jest "Serial Wire"



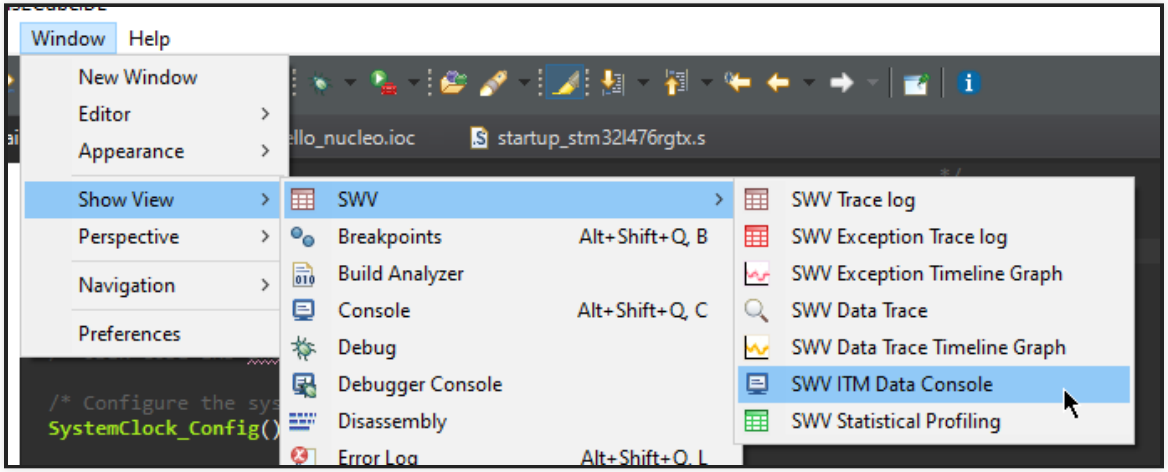
Po włączeniu SWV, możemy zobaczyć czy odbieramy dane z mikrokontrolera.

W głównej pętli, po zmianie stanu diody należy wstawić linijkę

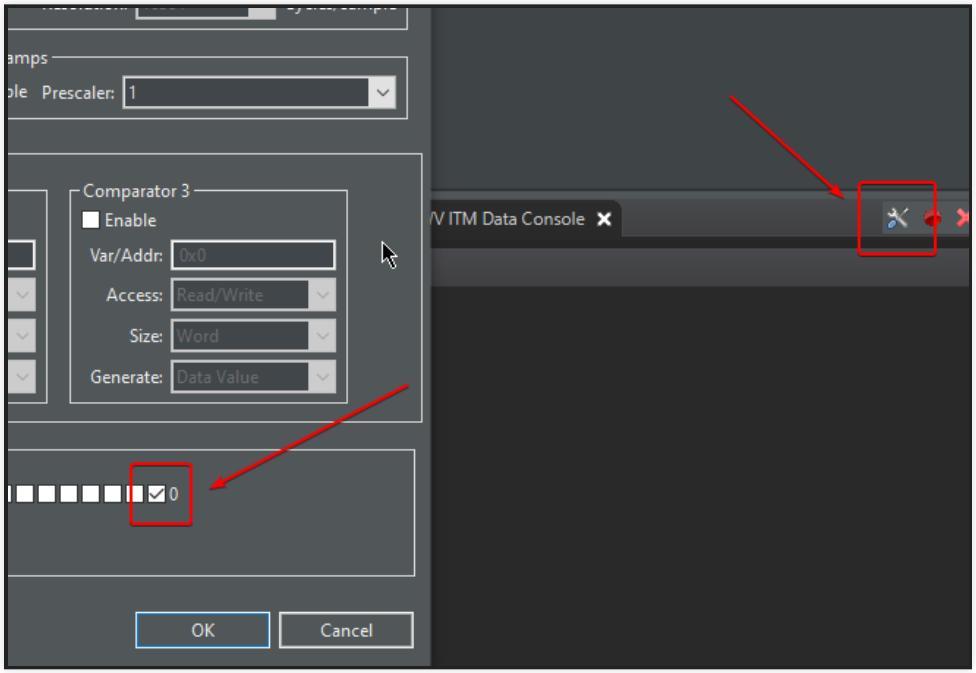
```
ITM_SendChar('x');
```

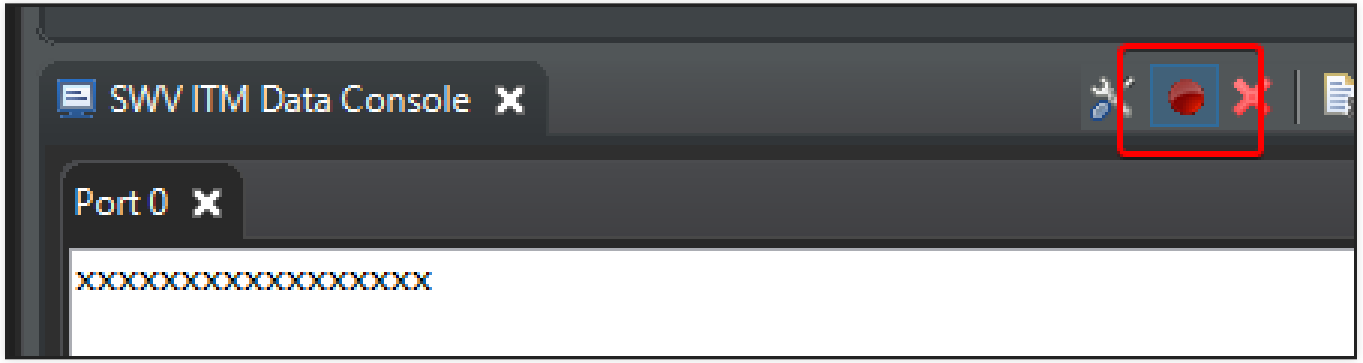
i rozpocząć debugowanie

Przed kliknięciem "Continue" należy stworzyć okienko z konsolą SWV.
Na pasku na górze wchodzimy w menu "Window" -> "Show View" -> "SWV" -> "SWV ITM Data Console".
Następnie ustawiamy sobie stworzone menu w wygodnej dla nas pozycji.



Po ustawieniu okna, klikamy "Configure trace" i zaznaczamy port 0. W "Data Console" powinna się pojawić nowa karta z opisem "Port 0". Klikamy "Start trace" (czerwone kółko) i możemy uruchomić program - w karcie powinny zacząć pojawiać się znaki *x*




SWV ITM Data Console
Port 0
xxxxxxxxxxxxxxxxx

ERROR HANDLING

Mikrokontrolery STM32 mają wbudowaną obsługę krytycznych błędów. W momencie w którym procesor wejdzie w błędny stan (na przykład przez odwołanie do błędnego wskaźnika lub krytyczny błąd peryferiów), procesor automatycznie skoczy do jednej z poniższych funkcji (znajdują się one w pliku `stm32X_it.c`, gdzie `X` to seria procesora, na przykład `f4xx` dla STM32F4):

- `HardFault_Handler` - krytyczny błąd który występuje w momencie w którym pojawi się nieobsłużony (lub niemożliwy do obsłużenia) stan programu.
- `MemManage_Handler` - wywołuje się w momencie w którym program próbuje dostać się do chronionej przez MPU (*Memory Protection Unit*) pamięci
- `BusFault_Handler` - wywołuje się w momencie wystąpienia krytycznego błędu na szynie danych
- `UsageFault_Handler` - wywołuje się w momencie nieprawidłowego obsłużenia procesora (na przykład przy próbie wykonania niewłaściwej instrukcji, lub próbie niepoprawnego dostępu do pamięci, albo dzielenia przez zero jeśli procesor jest odpowiednio skonfigurowany)

Program domyślnie zawiesi się na tym handlerze, żeby programista mógł sprawdzić stan pamięci i rejestrów.

BONUS: PRZEKIEROWANIE `printf` NA SWV

Istnieje bardzo prosta metoda na używanie `printf`'a do debugowania naszego kodu poprzez SWV. Wystarczy nadpisać funkcję `__io_putchar` z której korzysta `printf` do wyświetlania tekstu w następujący sposób:

```
int __io_putchar(int c) {
    if (c == '\n') {
    // Obsługa znaku nowej linii, żeby nie trzeba było za każdym razem używać \r\n
        ITM_SendChar('\r');
    }
    ITM_SendChar(c);

    return c;
}
```

Funkcja `ITM_SendChar` wysyła pojedynczy znak po SWV. Jeśli poprawnie skonfigurowaliśmy debugging, tekst powinien wyświetlić się w konsoli SWV na porcie 0.

Czasami standardowy `printf` potrafi crashować procesor. W takim przypadku, polecam skorzystać z alternatywnej implementacji, na przykład https://github.com/mpaland/printf (w tej konkretnej, należy nadpisać funkcję `void _putchar(char character)`)

DODATKOWE MATERIAŁY

Mastering STM32 (świetna książka opisująca w ludzki sposób działanie STMa oraz obsługę peryferiów)
https://www.carminenoviello.com/mastering-stm32

Manual programowania procesora (zaawansowane i szczegółowe źródło informacji na temat programowania STMów opartych o daną serię Cortexów):
PM0214 - Programming Manual - STM32 Cortex-M4 and MPUs programming manual
https://www.st.com/content/ccc/resource/technical/document/programming_manual/6c/3a/cb/e7/e4/ea/44/9b/DM

https://www.google.com/ - w szukajkę wpisujemy `site:st.com filetype:pdf XXX`, gdzie `XXX` to rzecz która nas interesuje. Na stronie ST jest mnóstwo materiałów szkoleniowych w formie dokumentów i prezentacji które wyjaśniają mniej lub bardziej szczegółowo działanie peryferiów STM32.
Jeśli nie znajdziemy wyników które nas interesują, to pomijamy `site:st.com` (i/lub `filetype:pdf`).