

Wydział Informatyki AGH



Ćwiczenie laboratoryjne

Rozwój i badanie działania niskopoziomowego oprogramowania dla systemu mikroprocesorowego

Nazwa kodowa: **debug-devel**. Wersja **20251027**

dr inż. Ada Brzoza-Zajęcka

ada.brzoza@agh.edu.pl

1. Wstęp

1.1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z wydajnymi metodami tworzenia oprogramowania dla urządzeń mikroprocesorowych opartych na nowoczesnych mikrokontrolerach jednoukładowych przy wykorzystaniu współczesnych narzędzi programowych, w tym sterowników i warstwy abstrakcji (HAL) z generatora kodu startowego. Wykonując ćwiczenie będzie można przekonać się, jak w relatywnie prosty sposób utworzyć oprogramowanie dla systemu wbudowanego działającego na nowoczesnym mikrokontrolerze jednoukładowym klasy przemysłowej.

1.2 Plan wykonania ćwiczenia

W ramach ćwiczenia będzie można uruchomić:

- system operacyjny FreeRTOS,
- port szeregowy wraz z podstawowymi mechanizmami interakcji z programem terminalowym (np. PuTTY),
- interfejs Ethernet, stos TCP/IP i prosty serwer WWW,
- interfejs USB-OTG (On-the-Go) pracujący w trybie Host, obsługujący przenośną pamięć masową USB (pendrive) wraz z biblioteką obsługi systemu plików FAT.

Uruchomimy także profesjonalne zintegrowane środowisko IDE i poznamy sposoby debugowania kodu na platformie sprzętowej. Pełne możliwości takiego debugowania są dostępne na platformie NUCLEO-F429ZI.

Sposób utworzenia projektu startowego i wygenerowania sterowników w środowisku STM32 CubeMX omówiono w końcowych rozdziałach instrukcji. STM32CubeMX to graficzne środowisko, w którym możemy wykonać ustawienia nawet bardzo zaawansowanych i rozbudowanych układów peryferyjnych mikrokontrolera nie pisząc ani jednej linijki kodu. Zaczniemy od importowania uprzednio wygenerowanego w kodu startowego do środowiska IDE, gdzie w języku C napiszemy kod realizujący podstawową funkcjonalność wyższego poziomu obejmującą wykorzystanie wygenerowanych sterowników. Przeprowadzimy także uruchomienie kodu i debugowanie.

1.3 Wykorzystane narzędzia

Ćwiczenie zostało opracowane i jest realizowane przy pomocy następujących narzędzi programowych:

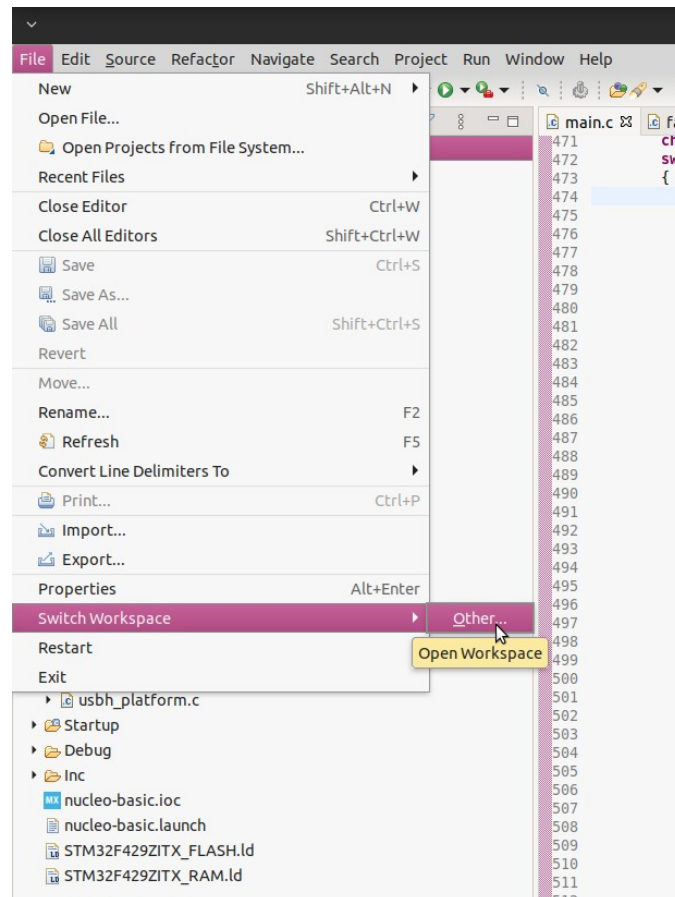
- STM32CubeMX w wersji 6.3.0
- STMicroelectronics STM32CubeIDE w wersji 1.7.0

Użyty przy tworzeniu instrukcji kod sterowników (firmware package) dla STM32CubeMX to: **FW_F4 V1.26.2**.

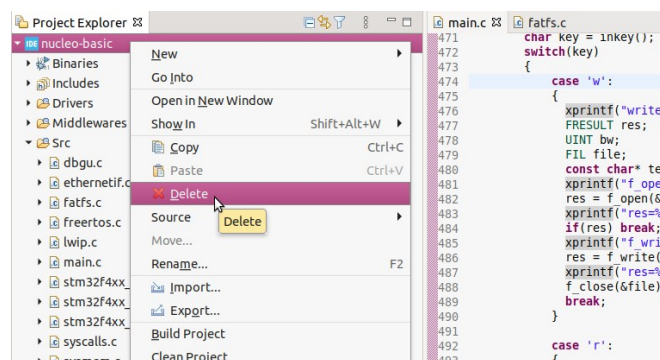
2. Import projektu do STM32CubeIDE

Hint: na początku pracy warto dla porządku oczyścić obszar roboczy zwany tutaj „workspace” z poprzednich projektów oraz zachować ogólny porządek w oprogramowaniu. To zmniejszy prawdopodobieństwo błędów i niezgodnego z oczekiwaniami zachowania oprogramowania.

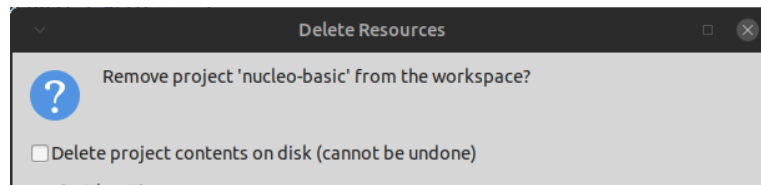
1. Po rozpakowaniu **warto nadać unikalną nazwę** dla katalogu z projektem.
2. Zalecane jest także, aby przełączyć się na domyślny obszar roboczy z katalogu `~/STM32CubeIDE/workspace_1.7.0`,



3. Dobrze jest także wyczyścić obszar roboczy z projektów, nad którymi pracowały poprzednie zespoły. Otwarte projekty można bez większych konsekwencji usunąć z workspace.

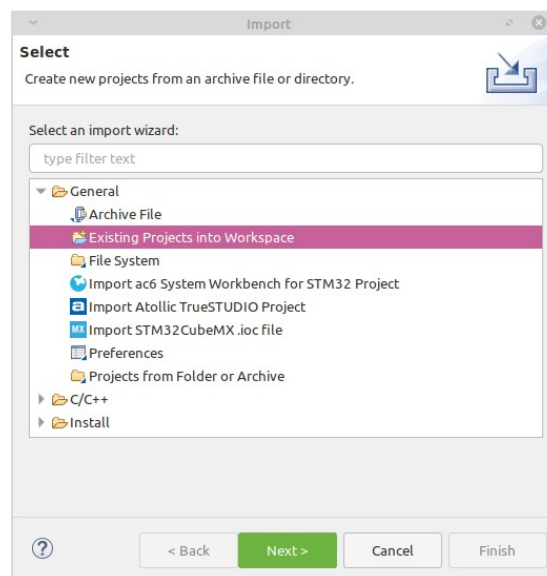


Raczej nie jest zalecane usuwanie ich zawartości z dysku, w razie jakby osoby, które nad tymi projektami pracowały, zapomniały o ich zarchiwizowaniu ;-) Dlatego **nie zaznaczamy** opcji *Delete project contents on disk (cannot be undone)*.

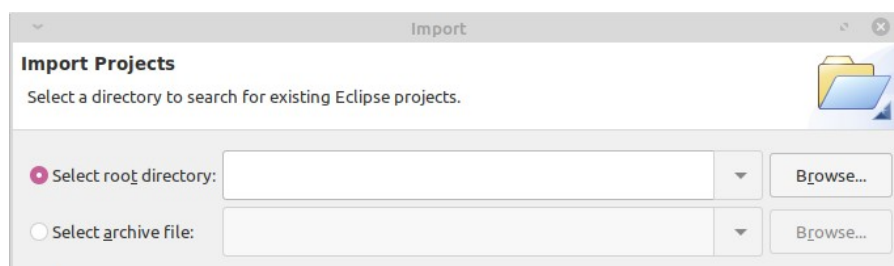


Gdy obszar roboczy jest już uporządkowany, możemy śmiało przechodzić do importu projektu startowego.

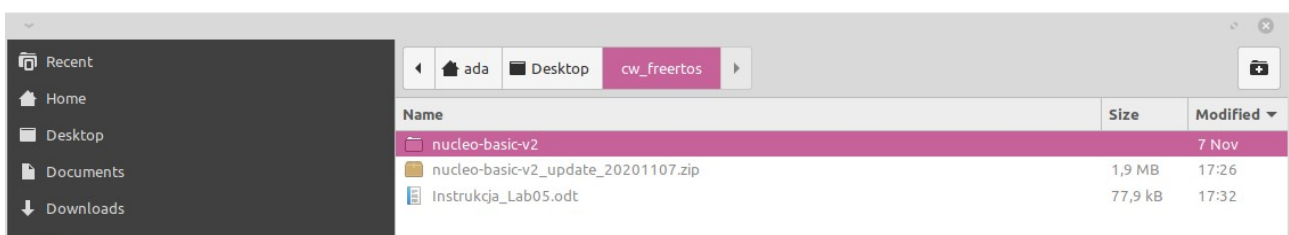
Pobrany projekt startowy nucleo-basic należy rozpakować i importować w środowisku STM32CubeIDE przy pomocy polecenia **File** → **Import...** a następnie rozwijamy sekcję **General** → **Existing Projects into Workspace** → **Next**.



W kolejnym oknie dialogowym *Import*, zaznaczamy **Select root directory** i wciskamy **Browse...**



Wskazujemy katalog z pobranym projektem, np.:



Następnie zatwierdzamy przyciskiem **Finish**.

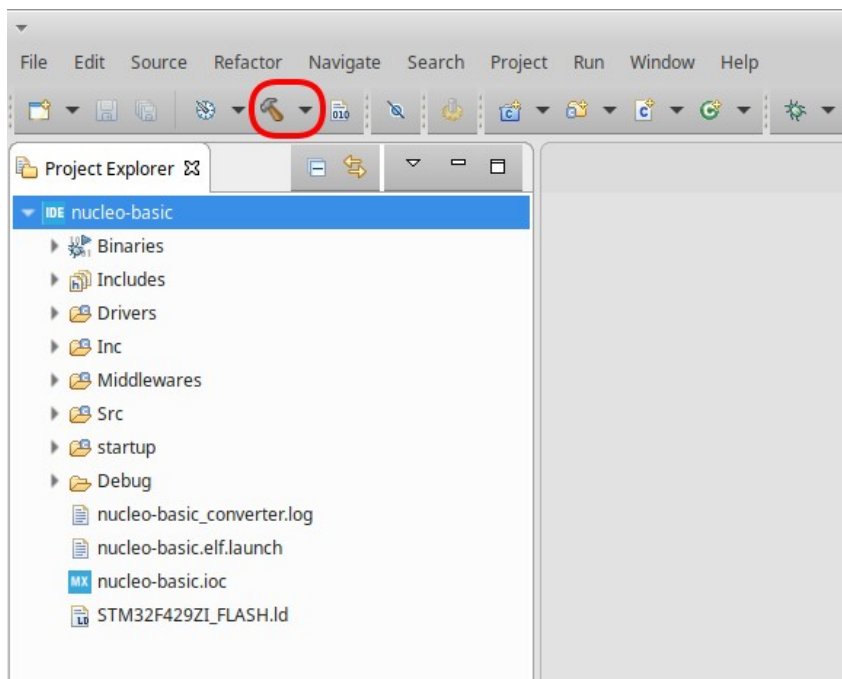
Po wybraniu katalogu wciskamy przycisk **Open**. Powinniśmy zobaczyć okienko podsumowania, np. takie jak poniżej. Jeśli wszystko się zgadza, wciskamy przycisk **Finish**.

3. Kompilacja i debugowanie

Na tym etapie powinniśmy widzieć otwarty projekt w środowisku STM32CubeIDE.

W tej części wykonamy próbną kompilację i debugowanie projektu.

1. Jednym ze sposobów uruchomienia kompilacji projektu jest kliknięcie na ikonkę „młotek” na pasku głównego okna CubeIDE:



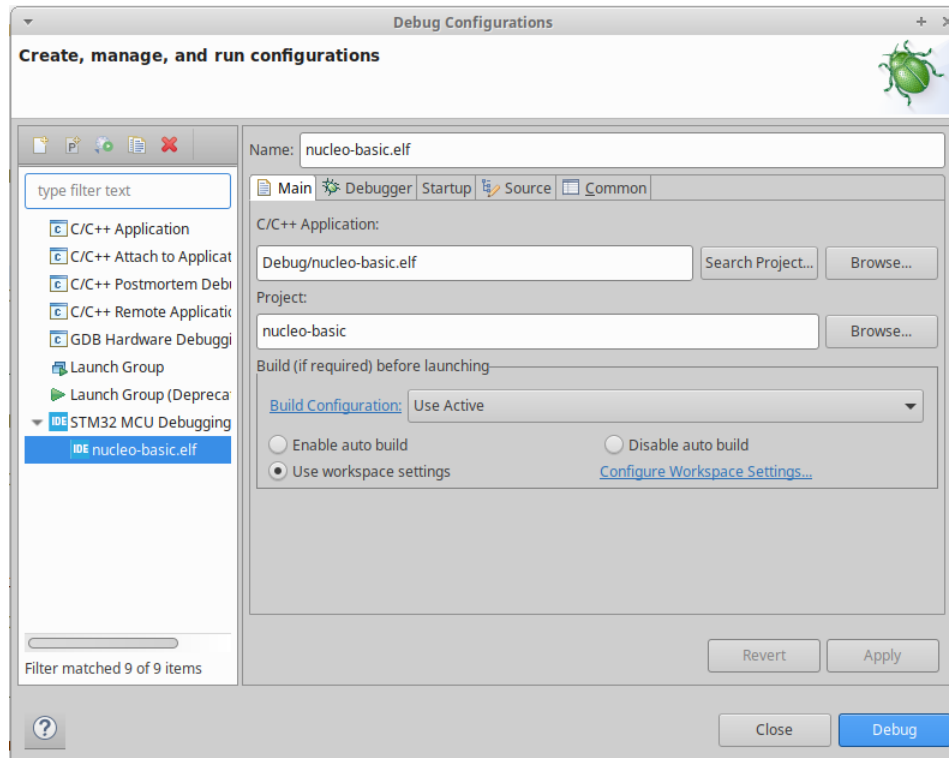
2. Jeśli pod koniec generowania kodu wynikowego pojawi się błąd kompilacji (właściwie błąd linkera), to z pliku:

Middlewares/Third_Party/LwIP/system/OS/sys_arch.c

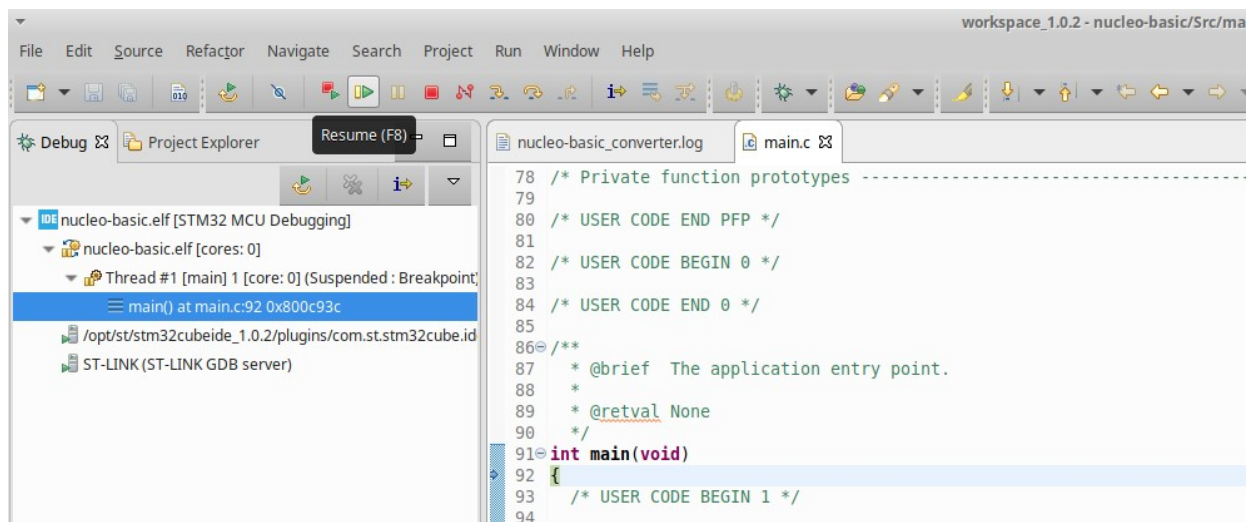
należy usunąć fragment kodu z definicją zmiennej **errno** (można usunąć cały zaznaczony fragment):



3. Debugowanie rozpoczynamy np. klikając ikonę „żuczek” na głównym pasku narzędzi. Przed włączeniem debugowania płytką powinna być podłączona przez interfejs USB swojego programatora do komputera PC, na którym pracujemy.
4. Następnie pojawi się okno konfiguracyjne procesu debugowania. Jeśli wszystko idzie zgodnie z planem, to tutaj powinno być wystarczające wciśnięcie przycisku **Debug**.

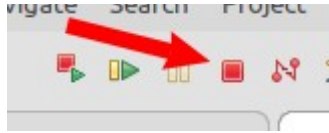


5. Po rozpoczęciu debugowania środowisko przejdzie do właściwej perspektywy i będziemy mieć dostępne narzędzia umożliwiające klasyczne debugowanie – jednak w tym przypadku debugowanie będzie się odbywało na docelowej platformie. Uruchomienie programu odbędzie się ciskając przycisk **Resume** (klawisz F8). Można także zatrzymać program przyciskiem „pauzy” oraz wykonywać typowe czynności „step into”, „step over”, „step out”.



6. Przed kontynuacją edycji projektu tymczasowo wychodzimy z trybu debugowania

wciskając przycisk wyglądający jak „Stop” lub używając skrótu klawiszowego Ctrl+F2.



4. FreeRTOS tips & tricks – warto przeczytać

Projekty startowe do większości ćwiczeń działają z systemem operacyjnym czasu rzeczywistego FreeRTOS. Nie jest to typowy system operacyjny, jakiego używamy na co dzień. Co prawda FreeRTOS nie pozwala na uruchamianie dostarczonych, skompilowanych programów, jednak pozwala na bardzo dokładne zarządzanie czasem wykonania zadań.

Zdania (tasks) w systemie operacyjnym FreeRTOS tworzone są z funkcji, w których typowo mamy nieskończoną pętlę *for(;;)* lub *while(1)*. Jeśli spróbujemy wyjść z takiej pętli (np. przez polecenie *break*), a następnie z funkcji zadania, to system nie poradzi sobie z taką sytuacją i pojawi się wyjątek sprzętowy. Trzeba na tego typu sytuacje uważać i nie kończyć funkcji, które stanowią zadania bez wywołania odpowiednich poleceń API, np. *vTaskDelete*. W niniejszym ćwiczeniu funkcją stanowiącą zadanie jest np. *StartDefaultTask*.

Używając projektów w takiej konfiguracji z systemem FreeRTOS jak na laboratorium warto także pamiętać, aby poza uzasadnionymi specyficznymi sytuacjami unikać używania funkcji opóźniających wykonywanie kodu *HAL_Delay*. Te funkcje zatrzymują działanie programu w swojej wewnętrznej pętli oczekiwania, przez co zatrzymane zadanie zużywa w czasie takiego opóźnienia całą dostępną moc obliczeniową. Jeśli nie chcemy „zabierać” mocy obliczeniowej innym zadaniom, wystarczy wywołać albo funkcję *osDelay* z dostępnego API wysokopoziomowego albo jedną z funkcji typu *vTaskDelay* lub *vTaskDelayUntil* z API systemu FreeRTOS. Zarówno *osDelay* jak i *vTaskDelay* w czasie realizacji opóźnienia w pewnym sensie „oddają” zasoby innym zadaniom, a same nie zużywają czasu procesora, przez co oprogramowanie może być znacznie bardziej efektywne, a same funkcje opóźniające nie zakłócą działania zadań o niższym priorytecie.

5. Interakcja z programem terminalowym

Aby móc wyświetlać komunikaty oraz ew. zadawać proste dane wejściowe, można dodać moduły obsługi portu szeregowego i terminala. Pliki te są dostępne w materiałach do ćwiczenia. Należy je dodać do podkatalogów **Inc** oraz **Src** projektu wg tego, jak zostały umieszczone w archiwum **serial.zip**. Następnie odświeżamy projekt (prawym przyciskiem na nazwie projektu w IDE i wybieramy **Refresh (F5)**).

Aby z nich korzystać, należy dodać do sekcji **/* USER CODE BEGIN Includes */** następujące pliki nagłówkowe:

```
#include "dbg_u.h"
#include "term_io.h"
```



```
#include "ansi.h"
```

A w funkcji **main**, po inicjalizacji układów peryferyjnych, dopisać (sekcja **USER CODE BEGIN 2**) inicjalizujemy sterownik i, opcjonalnie, wypisujemy pierwszy komunikat:

```
/* USER CODE BEGIN 2 */
debug_init(&huart3);
xprintf(ANSI_BG_BLUE "Nucleo-144 project" ANSI_BG_DEFAULT "\n");
printf("Zwykły printf też działa\n");
/* USER CODE END 2 */
```

Jeśli teraz uruchomimy program na platformie sprzętowej, a wcześniej połączymy się programem terminalowym (np. PuTTY) z odpowiednim portem szeregowym (prawdopodobnie będzie to **/dev/ttyACM0**), to powinniśmy zobaczyć komunikaty diagnostyczne naszego urządzenia.

Parametry transmisji są następujące:

- 115200 baud,
- 8 bitów danych,
- 1 bit stopu,
- brak bitu parzystości.

Warto także przy uruchamianiu PuTTY w oknie konfiguracji, w sekcji **Terminal** włączyć **Implicit CR in every LF**.

6. Błykanie diodą LED

W main.c odnajdujemy funkcję **StartDefaultTask**.

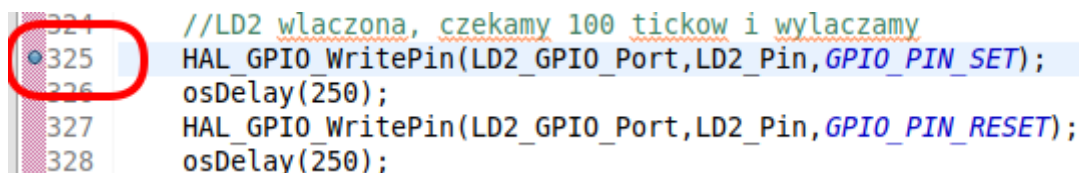
W pętli zadania, w sekcji **USER CODE BEGIN/END 5** wpisujemy następujący kod:

```
/* USER CODE BEGIN 5 */
/* Infinite loop */
for(;;)
{
    //LD3 zmieni stan (toggle)
    HAL_GPIO_TogglePin(LD3_GPIO_Port,LD3_Pin);

    //LD2 włączona, czekamy 250 ticków i wyłączamy
    HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_SET);
    osDelay(250);
    HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_RESET);
    osDelay(250);
}
/* USER CODE END 5 */
```

7. Krokowe wykonanie programu

W main.c w pętli głównej **USER CODE BEGIN/END 5** zakładamy breakpoint na linii, w której ustawiamy wyprowadzenie LD2_Pin w stan wysoki. Breakpoint ustawiamy klikając dwukrotnie na numerze linii, na której chcemy uaktywnić breakpoint, w tym przypadku będzie to linia z wywołaniem **HAL_GPIO_WritePin**.



```
324 //LD2 włączona, czekamy 100 ticków i wyłączamy
325 HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_SET);
326 osDelay(250);
327 HAL_GPIO_WritePin(LD2_GPIO_Port,LD2_Pin,GPIO_PIN_RESET);
328 osDelay(250);
```


Po ponownym uruchomieniu debuggowania program zatrzyma się najpierw tam gdzie ostatnio, czyli na początku funkcji main, a po przejściu dalej, na nowo ustawionym breakpointie.

Należy teraz skorzystać z dostępnych możliwości sterowania biegiem programu podczas debuggowania: **Step Into Step Over, Step Return** (F5, F6, F7):



Może być przydatne lub wygodne, aby założyć breakpointy ew. również w innych liniach programu.

W ramach realizacji tego punktu staramy się zaobserwować i wskazać fragmenty kodu odpowiadające za wpisywanie danych do bitów i rejestrów układów peryferyjnych modułu GPIO mikrokontrolera.

Należy umieć wyjaśnić osobie prowadzącej **gdzie (do których bitów rejestrów układów peryferyjnych) i dlaczego wpisywane są konkretne wartości** podczas:

1. włączania diody LED podłączonej do wybranego wyprowadzenia,
2. wyłączania diody LED podłączonej do wybranego wyprowadzenia,
3. zmiany (odwrócenia) stanu tej diody LED.

Na platformie UPEL umieszczamy 2-3 screenshoty prezentujące przebieg analizy działania oraz zmodyfikowany plik *main.c* (tworzymy kopię *main.c*, umieszczamy ją poza katalogiem projektu, zmieniamy nazwę tego pliku na ***main_led.txt*** i wtedy umieszczamy na UPEL)

Uwaga: w tym punkcie chodzi o **rejestry układów peryferyjnych**, które w typowych mikrokontrolerach (także w STM32) są odwzorowane w przestrzeni adresowej. Podpowiedź - proszę znaleźć w dokumentacji informacje na temat **rejestrów BSRR oraz ODR**. Zalecana dokumentacja to "ST Microelectronics STM32F4... Reference Manual, RM0090", zależnie od wersji dokumentu te informacje mogą być dostępne w rozdziale pt. "General-purpose I/Os (GPIO)".

8. Obserwacja zawartości rejestrów wewnętrznych CPU

Uwaga na start: przed przejściem do zadań z tego punktu bardzo proszę **usunąć breakpointy ustawione wcześniej** w punkcie 7. Jeśli ich nie usuniemy, podczas debuggowania co chwilę będziemy otrzymywać przerwanie od układu Timera 6.

Obserwację wykorzystania rejestrów można przeprowadzić przy pomocy np. przy pomocy funkcji umieszczonej w sekcji pomiędzy `/* USER CODE BEGIN 0 */` a `/* USER CODE END 0 */`:

```
int foo(int a, int b)
{
    int c;
    c=a+b;
    return c;
}
```

oraz następującego fragmentu kodu w funkcji **main**, tuż przed `/* USER CODE END 2 */`:

```
int a=10;
int b=25;
while(1)
{
    xprintf("a=%d, b=%d\n",a,b);
    int res = foo(a,b);
    a++;
    b--;
    xprintf("res=%2d\n",res);
    HAL_Delay(1000);
}
/* USER CODE END 2 */
```

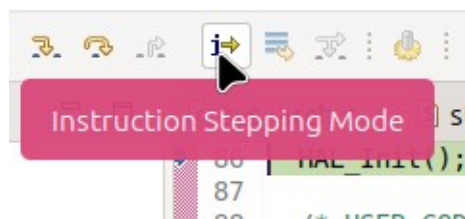
Uwaga: w powyższym kodzie używanie `HAL_Delay` jest uzasadnione, ponieważ w tym miejscu kodu system operacyjny FreeRTOS jeszcze nie jest uruchomiony. Przez to funkcje `vTaskDelay` lub `osDelay` nie zadziałałyby.

Dla ułatwienia, możemy założyć breakpoint na wywołaniu funkcji **foo**

```
int res = foo(a,b);
```

a następnie obserwujemy działanie programu i zajętość rejestrów w sesji debuggowania przy poziomie optymalizacji włączonym na **00** oraz wyższym, np. **0s**. Przy obserwacji wywołania funkcji szczególnie warto zwrócić uwagę na zajętość rejestrów **r0..r3**.

Dla ułatwienia oglądania wykonania niskopoziomowego kodu mikroprocesora wygodne może być włączenie wyświetlania pośredniego kodu w języku assemblera, który został wygenerowany w procesie kompilacji. W tym celu w perspektywie debuggowania można kliknąć przycisk **Instruction Stepping Mode**:

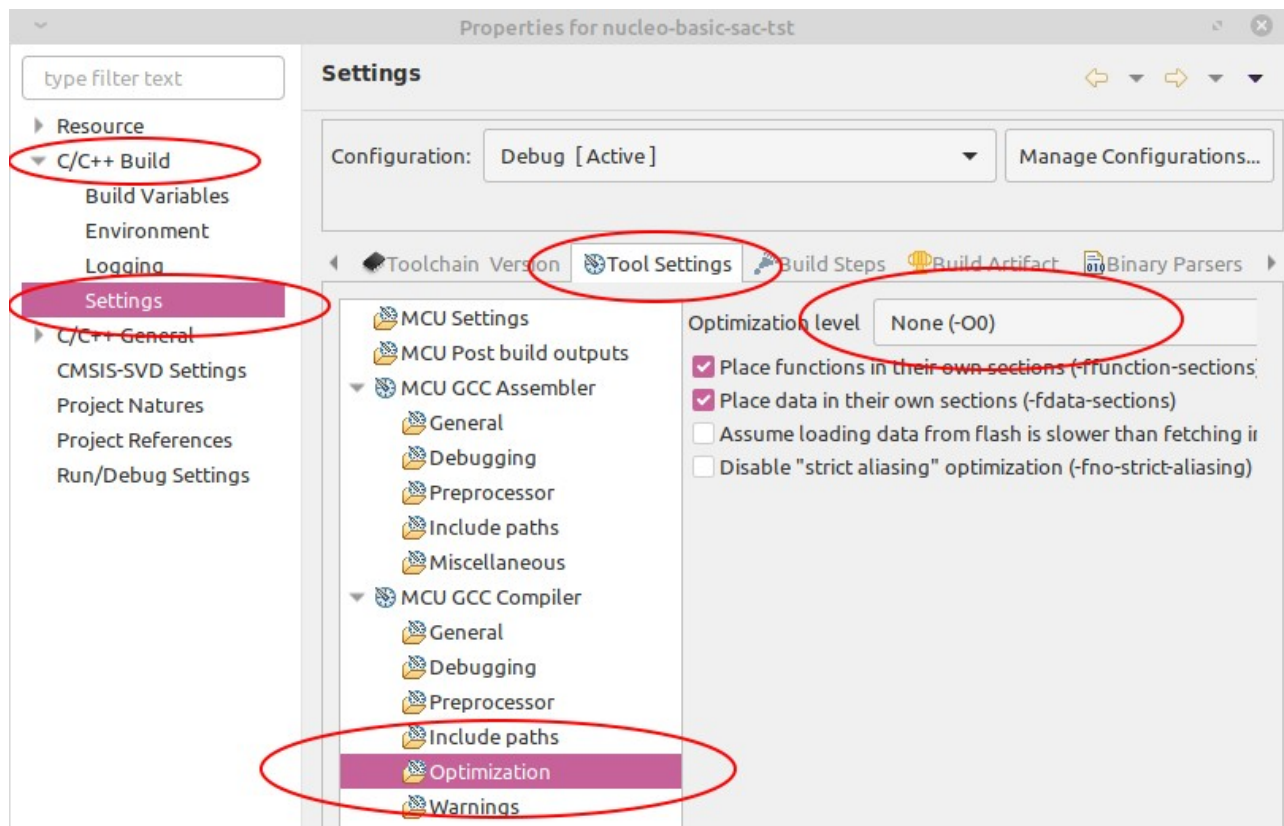


W ramach sprawozdania z tej części ćwiczenia należy **umieć opowiedzieć osobie prowadzącej**:

- W których rejestrach przechowywane są wartości przesyłane jako argumenty do funkcji?
- W którym rejestrze przekazywany jest wynik działania funkcji?
- Jak zmieniło się wykorzystanie rejestrów ogólnego przeznaczenia przy włączonej i wyłączanej optymalizacji kompilatora?
- Co stało z wywołaniem funkcji `foo` przy włączonej optymalizacji?

Ponadto na platformie UPEL umieszczamy screenshoty ukazujące zrealizowane prace oraz działanie procesu debuggowania oraz zmodyfikowany plik `main.c` (tworzymy kopię `main.c`, umieszczamy ją poza katalogiem projektu, zmieniamy nazwę tego pliku na **`main_reg.txt`** i wtedy umieszczamy na UPEL).

Dalej znajdują się „hinty i spoilery” ;-) ↓↓↓



Hint 1: poziom optymalizacji możemy ustawić we właściwościach projektu, w: *C/C++ Build* → *Settings* → *Tool Settings* → *MCU GCC Compiler* → *Optimization*.

Hint 2: Zawartość rejestrów możemy oglądać po uruchomieniu sesji debugowania i kliknięciu odpowiedniej zakładki w panelu z narzędziami do obserwacji działania programu:

Name	Value	Description
General Register File		General Purpose Registers
r0	35	
r1	24	
r2	11	
r3	35	
r4	536886932	
r5	0	
r6	0	
r7	537067472	
r8	0	
r9	0	
r10	0	
r11	0	
r12	2002	
sp	0x2002ff00	
lr	134244991	
pc	0x8006a80 <main>	
xpsr	16777216	
d0	0	
d1	0	

Może być przydatne także zachowanie plików pośrednich w języku assemblera (poniżej fragment

dokumentu *STM32CubeIDE User Guide*):

2.4.4

Temporary assembly file and preprocessed C code

Save the temporary assembly file by adding the `-save-temps` flag to the compiler:

1. In the menu, select [Project]>[Properties]
2. Select [C/C++ build]>[Settings]
3. Open the *Tool Settings* tab
4. Add `-save-temps` in the [C Compiler]>[Miscellaneous] settings
5. Rebuild the project

The assembler file is located in the build output directory with name `filename.s`.

The file `filenamez.i` containing the preprocessed C code is generated also. It shows the code after the preprocessor but before the compilation. It is advise to examine the content of this file in case of problems with defines.

Uwaga: Po zakończeniu obserwacji **warto pamiętać, aby usunąć przynajmniej fragment kodu z pętlą `while(1)` w funkcji `main`** – inaczej wykonanie programu nie przejdzie do kolejnych, interesujących nas etapów.

Najlepiej będzie przywrócić kod z sekcji `USER CODE BEGIN/END 2` do następującej postaci:

```
/* USER CODE BEGIN 2 */
debug_init(&huart3);
xprintf(ANSI_BG_BLUE "Nucleo-144 project" ANSI_BG_DEFAULT "\n");
printf("Zwykly printf tez dziala\n");
/* USER CODE END 2 */
```

9. Obsługa pamięci masowej

Obsługa USB jest w dużej części zaimplementowana w sterownikach wygenerowanych z STM32CubeMX. Musimy jednak dopisać kod umożliwiający interakcję z tymi sterownikami.

9.1 Modyfikacje w `Src/fatfs.c`

W module `fatfs.c` projektu startowego zostały utworzone zmienne i struktury potrzebne przy obsłudze systemu plików:

- **USBHPath** – ścieżka do urządzenia magazynującego (właściwie zawiera tylko ciąg znaków `0:/`)
- **USBHFatFS** – struktura reprezentująca system plików.

Struktura **USBHFatFS** musi zostać zainicjalizowana zanim wykonamy jakiekolwiek operacje na plikach. Możemy zainicjalizować ją w funkcji `MX_FATFS_Init` lub w wybranym miejscu modułu `main.c`, np. w reakcji na podłączenie pamięci masowej. Tutaj wybrano wariant najprostszy – inicjalizację w module **Src/fatfs.c** w funkcji **MX_FATFS_Init**. Kod inicjalizacji systemu plików umieszczamy w sekcji `/* USER CODE BEGIN Init */`:

```
/* USER CODE BEGIN Init */
/* additional user code for init */
f_mount(&USBHFatFS, USBHPath, 0);
/* USER CODE END Init */
```

9.2 Modyfikacje w `Src/main.c`

1. W pliku `main.c` w sekcji dodajemy nagłówek *usbh_platform.h*:

```

/* USER CODE BEGIN Includes */

#include "dbg.h"
#include "term_io.h"
#include "ansi.h"
#include "usbh_platform.h"
/* USER CODE END Includes */

```

2. Na początku sekcji `/* USER CODE BEGIN 0 */` dodajemy deklarację zmiennej z modułu:

```
extern ApplicationTypeDef Appli_state;
```

Ta zmienna pozwala na śledzenie stanu podłączonego urządzenia USB. Gdy przyjmie ona wartość zadaną stałą `APPLICATION_READY` to urządzenie jest gotowe do pracy.

3. W głównym zadaniu, w funkcji **StartDefaultTask**, sekcji rozpoczynającej się od `/* USER CODE BEGIN 5 */` wpisujemy kod służący do oczekiwania na gotowość dołączonej do płytki pamięci masowej USB.

```

MX_DriverVbusFS(0); //włącza zasilanie urządzenia USB
xprintf("waiting for USB device...");
do{
    xprintf(".");
    vTaskDelay(100);
}while(Appli_state != APPLICATION_READY);
xprintf("USB device ready!");

```

4. Dalej w pętli głównej zadania głównego (po komentarzu `/* Infinite loop */`) wklejamy prosty kod do tworzenia, zapisu i odczytu pliku – zastępujemy tym kodem całą nieskończoną pętlę zadania:

```

/* Infinite loop */
for(;;)
{
    osDelay(10);

    char key = inkey();
    switch(key)
    {
        case 'w':
        {
            xprintf("write test\n");
            FRESULT res;
            UINT bw;
            FIL file;
            const char* text = "Linijka tekstu!\n";
            xprintf("f_open... ");
            res = f_open(&file,"0:/test.txt",FA_WRITE|FA_OPEN_APPEND);
            xprintf("res=%d\n",res);
            if(res) break;
            xprintf("f_write... ");
            res = f_write(&file,text,strlen(text),&bw);
            xprintf("res=%d, bw=%d\n",res,bw);
            f_close(&file);
            break;
        }

        case 'r':
        {
            xprintf("read test!\n");
            FIL file;
            FRESULT res = f_open(&file,"0:/test.txt",FA_READ);
            xprintf("f_open res=%d\n",res);

```

```

        if(res) break;

        const uint32_t BUF_SIZE = 64;
        char buf[BUF_SIZE];

        UINT br;
        xprintf("reading file contents:\n");
        do
        {
            res = f_read(&file,buf,BUF_SIZE,&br);

            if((res == FR_OK) && (br))
            {
                xprintf("chunk:\n");
                debug_dump(buf,br);
            }
            else
            {
                xprintf("f_read res=%d\n",res);
                break;
            }
        }while(br>0);
        f_close(&file);
        break;
    }
}

```

9.3 Działanie programu testowego

Po uruchomieniu programu możemy wysłać do urządzenia z terminala znak 'w' (jeden lub więcej razy), a następnie znak 'r'. Po wysłaniu znaku 'w' na nośniku pamięci masowej zostanie utworzony plik tekstowy test.txt (za pierwszym wywołaniem) i zostanie dopisany do niego ciąg znaków:

```
"Linijka tekstu!\n"
```

Kolejne wywołania 'w' spowodują dopisanie do pliku kolejnych identycznych ciągów znaków.

Wysłanie znaku 'r' spowoduje odczyt całego pliku w formie zrzutu pamięci wraz z interpretacją ASCII.

W ramach sprawozdania z tej części ćwiczenia, należy zaprezentować osobie prowadzącej działanie zapisu i odczytu danych z pliku. Ponadto screenshoty z działania oraz zmodyfikowany kod *main.c* umieszczamy na UPEL (tworzymy kopię *main.c*, zmieniamy nazwę tej kopii *main.c* na *main_msd.txt*, a utworzony plik umieszczamy na UPEL)

10. Serwer WWW

W tej części zajmiemy się uruchomieniem prostego serwera WWW zaimplementowanego w programie działającym na mikrokontrolerze.

10.1 Przygotowanie kodu

Na początek należy ustawić także unikalny adres fizyczny interfejsu sieciowego płytki Nucleo 144. Jeśli wiele urządzeń będzie miało takie same adresy fizyczne, serwer DHCP nie będzie w stanie przydzielić adresów IP. Adres fizyczny płytki możemy zmienić w pliku **Src/ethernetif.c**, w funkcji **low_level_init**.

```

static void low_level_init(struct netif *netif)
{
    uint32_t regvalue = 0;
    HAL_StatusTypeDef hal_eth_init_status;

    /* Init ETH */

    uint8_t MACAddr[6] ;
    heth.Instance = ETH;
    heth.Init.AutoNegotiation = ETH_AUTONEGOTIATION_ENABLE;
    heth.Init.Speed = ETH_SPEED_100M;
    heth.Init.DuplexMode = ETH_MODE_FULLDUPLEX;
    heth.Init.PhyAddress = LAN8742A_PHY_ADDRESS;
    MACAddr[0] = 0x00;
    MACAddr[1] = 0x80;
    MACAddr[2] = 0xE1;
    MACAddr[3] = 0x00;
    MACAddr[4] = 0x66;
    MACAddr[5] = 0x00;
    heth.Init.MACAddr = &MACAddr[0];
    heth.Init.RxMode = ETH_RXINTERRUPT_MODE;
    ...

```

Dalsze modyfikacje wykonujemy w *main.c*.

1. W sekcji zaczynającej się od `/* USER CODE BEGIN Includes */` dodajemy niezbędny nagłówek z biblioteki LwIP:

```
#include "lwip/api.h"
```

2. W sekcji od `/* USER CODE BEGIN 0 */` dopisujemy deklarację zmiennej zewnętrznej z biblioteki LwIP:

```
extern struct netif gnetif;
```

3. W sekcji kodu zaczynającej się od `/* USER CODE BEGIN 4 */` wpisujemy zadanie oraz kod serwera WWW z pliku **fragmenty_serwer.txt**.
4. Kodzie głównego zadania, na początku sekcji `/* USER CODE BEGIN 5 */` dopisujemy oczekiwanie na uzyskanie adresu z DHCP, wyświetlenie tego adresu oraz utworzenie zadania serwera WWW.

```
xprintf("Obtaining address with DHCP...\n");
```

```
struct dhcp *dhcp = netif_dhcp_data(&gnetif);
```

```
do
```

```
{
```

```
    xprintf("dhcp->state = %02X\n", dhcp->state);
```

```
    vTaskDelay(250);
```

```
}while(dhcp->state != 0x0A);
```

```
xprintf("DHCP bound\n");
```

```
displayOwnIp();
```

```
osThreadDef(
```

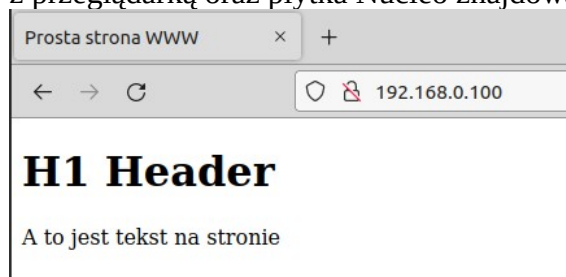
```
    netconn_thread, http_server_netconn_thread, osPriorityNormal, 0, 1024);
```

```
osThreadCreate(osThread(netconn_thread), NULL);
```


Gdy uruchomimy powyższy kod, można sprawdzić, czy serwer działa. Jeśli kompilacja przebiegnie prawidłowo i wgramy nowy firmware do pamięci wewnętrznej mikrokontrolera, to na terminalu będzie można ujrzeć np. takie komunikaty jak poniżej:

```
Nucleo-144 project
Zwykly printf tez dziala
Obtaining address with DHCP...
dhcp->state = 01
dhcp->state = 01
dhcp->state = 01
dhcp->state = 08
dhcp->state = 08
dhcp->state = 08
DHCP bound
My IP: 192.168.0.100
```

Adres IP, który otrzymała używana przez nas platforma Nucleo, możemy wpisać w przeglądarce i sprawdzić, czy wyświetli się wygenerowana strona WWW. Trzeba jednak pamiętać, aby komputer z przeglądarką oraz płytka Nucleo znajdowały się w tej samej sieci lokalnej.



10.2 Co zrobić, gdy serwer nie działa? (basic troubleshooting)

Warto sprawdzić:

1. Czy przeglądarka przypadkiem nie próbuje wysłać zapytania HTTPS? (serwer obsługuje wyłącznie HTTP)
2. Czy kabel sieciowy jest podłączony do płytki? (tak, serwer korzysta z interfejsu Ethernet ;))
3. Czy komputer jest w tej samej sieci? Możemy to sprawdzić w terminalu, np. przy pomocy polecenia **ifconfig**.
4. Czy na pewno przeczytaliśmy rozdział 4 niniejszej instrukcji?

10.3 Zdanie do wykonania i zaraportowania

Należy zmodyfikować serwer w taki sposób, aby wyświetlał oryginalny tekst, unikalny dla zespołu realizującego ćwiczenie. Mile widziane są także inne kreatywne rozwiązania związane z działaniem serwera (np. wyświetlanie grafiki). **W ramach sprawozdania należy zaprezentować osobie prowadzącej działanie serwera wyświetlającego unikalny tekst, a screenshoty i zmodyfikowany kod źródłowy należy umieścić na UPEL w odpowiedniej sekcji (towrzymy kopię *main.c* i zmieniamy jej nazwę na *main_serwer.txt*).**

11. Co należy umieć?

Warto utrwalić zdobytą wiedzę i umiejętności zależnie od ogłoszonej przez osobę prowadzącą formy weryfikacji efektów uczenia dla tego zadania. Poniżej przedstawiono przykładowe zagadnienia, których znajomość może być przydatna przy weryfikacji efektów uczenia.

1. Jaka jest organizacja przestrzeni adresowej w mikrokontrolerze stosowanym w ćwiczeniu, w tym rodzaje zastosowanych w nim pamięci:
 - a. ile jest przestrzeni adresowych,
 - b. jakie mamy dostępne bloki pamięci (proszę podać ich nazwy),
 - c. w których blokach pamięci przechowujemy lub możemy przechowywać: wykonywalny kod programu, dane tylko do odczytu, zmienne.
2. Rozumienie budowy i działania stanowiska do rozwoju oprogramowania firmware dla mikrokontrolerów w systemach mikroprocesorowych wbudowanych.
3. Rozumienie interakcji pomiędzy narzędziami do rozwoju niskopoziomowego oprogramowania oraz mechanizmów ich działania:
 - a. Proces kompilacji, asemblacji, linkowania, generowanie plików wynikowych,
 - b. Zastosowane w ćwiczeniu narzędzia do rozwoju i obserwacji działania oprogramowania dla mikrokontrolera:
 - zastosowane środowisko IDE,
 - debugowanie – mechanizmy rządzące tą czynnością oraz które interfejsy były używane (JTAG, SWD, czy inne?),
 - rozumienie koncepcji zastosowania asynchronicznego portu szeregowego UART oraz wirtualnego łącza szeregowego ttyUSBn, ttyACMn – gdzie w ćwiczeniu stosujemy port fizyczny, a gdzie wirtualny utworzony jako urządzenie USB?
 - c. Debugowanie przez debugger lub przez wyświetlanie komunikatów (np. *printf*) – wady, zalety i przydatność obu podejść w kontekście rozwoju oprogramowania niskopoziomowego (było również omawiane na wykładzie), których sposobów używamy/możemy używać w ćwiczeniu.
 - d. Umieszczanie kodu wynikowego w pamięci Flash mikrokontrolerów – na czym polega, jak przebiega, jakich narzędzi używa się do tego celu w ćwiczeniu oraz ogólnie.

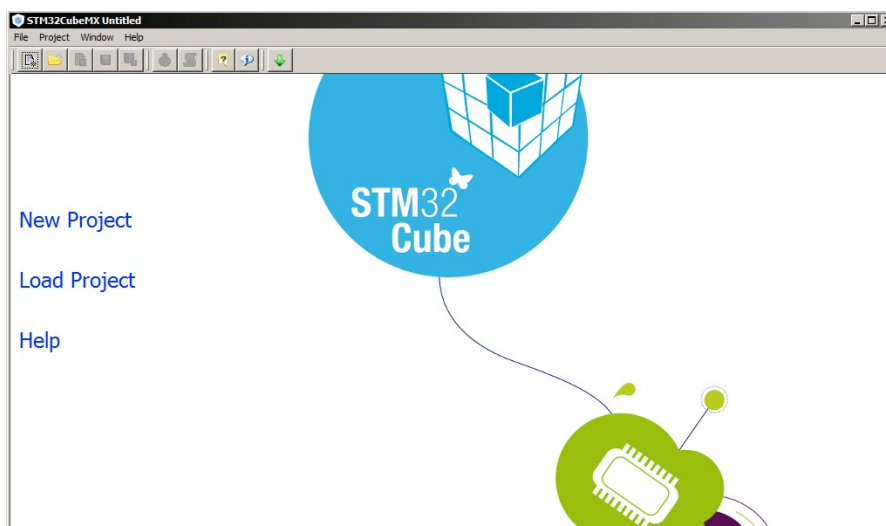
12. Dodatek: tworzenie projektu w STM32CubeMX v4.26.1

Uwaga: Informacje zawarte w niniejszym rozdziale mają charakter jedynie poglądowy. Opis dotyczy tworzenia bardzo podobnego kodu startowego, jak używany w ćwiczeniu, jednak przy pomocy starszej wersji STM32CubeMX.

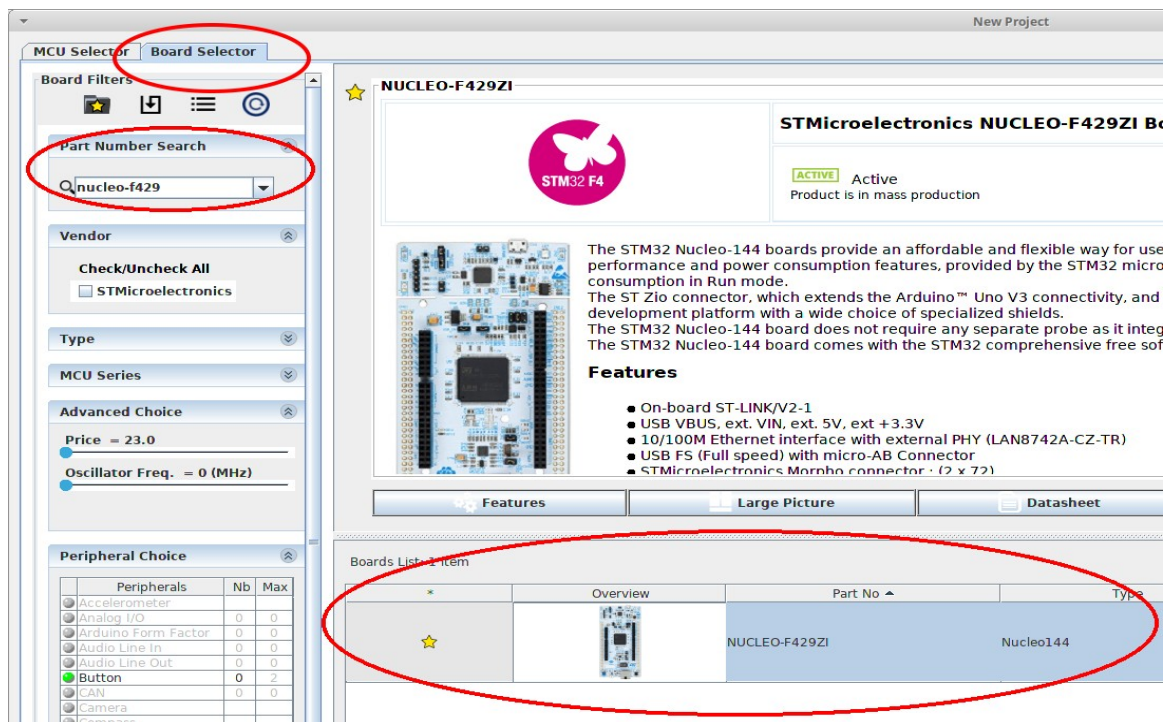
Zintegrowane środowiska do tworzenia oprogramowania dla systemów wbudowanych tradycyjnie wymagały co najmniej dobrej znajomości platformy sprzętowej, dla której ma powstać oprogramowanie. Oprócz tego należało bardzo ściśle pracować z dokumentacją sprzętu. Obecnie wymagana jest jedynie ogólna znajomość sprzętu, natomiast wiele trudniejszych szczegółów technicznych załatwiają zautomatyzowane środowiska.

12.1 Rozpoczęcie pracy z nowym projektem

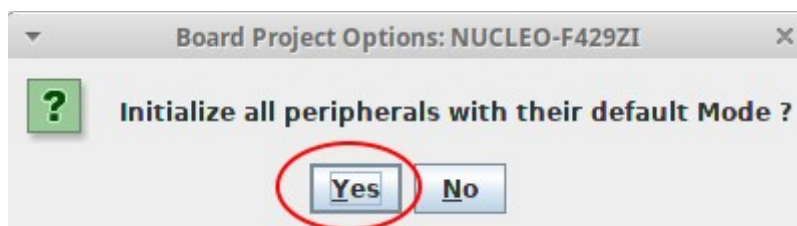
1. Uruchamiamy środowisko STM32CubeMX w wersji . W oknie głównym środowiska wybieramy oczywiście **New Project**.



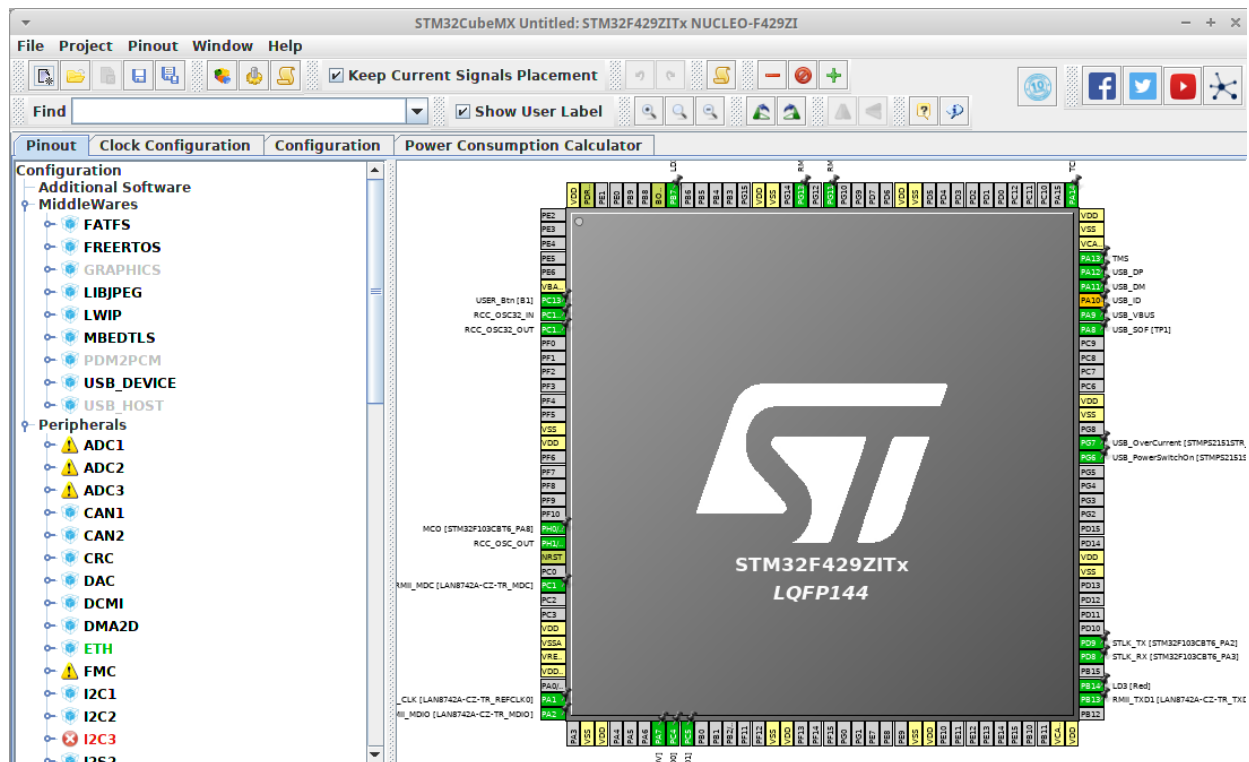
2. Tworzenie nowego projektu rozpoczynamy od zdefiniowania platformy sprzętowej: płytki lub mikrokontrolera, na których mamy zamiar pracować. Tutaj przyjmujemy, że tworzymy oprogramowanie dla platformy NUCLEO-F429ZI lub NUCLEO-F746ZG. Dlatego wybieramy zakładkę **Board Selector**, a dla ułatwienia wyszukiwania odpowiedniej płytki wpisujemy w polu **Part Number Search** fragment jej nazwy, np. **NUCLEO-F429**. Po zaznaczeniu płytki, wystarczy kliknąć dwa razy na wybranej płytce.



3. pojawi się dialog z pytaniem, czy wykonać domyślną inicjalizację układów peryferyjnych na wybranej płytce – zalecane jest wyrażenie zgody przyciskiem **Yes**.

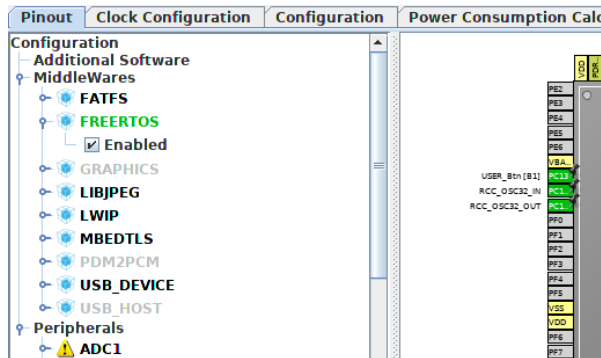


4. Pojawi się główne okno STM32CubeMX ze wstępnie zainicjalizowanymi układami peryferyjnymi dla wybranej płytki testowej. W głównym oknie możemy wyróżnić dwa podstawowe obszary: z lewej strony mamy dostępną listę modułów sprzętowych i programowych, a z prawej widok układu scalonego mikrokontrolera.



Z modułów programowych (*MiddleWares*), możemy tutaj dodać m.in. obsługę systemu plików FAT (znana nam już biblioteka FatFS), system operacyjny FreeRTOS (także znany z techniki mikroprocesorowej), programowy stos TCP/IP (biblioteka LWIP). Nie wszystkie moduły programowe są jednak dostępne dla danej konfiguracji sprzętowej (ustawianej w dolnej części listy), np. nie możemy dodać Middleware dla hosta USB, jeśli wcześniej nie uaktywnimy modułu sprzętowego hosta.

- Na początek listy **MiddleWares** rozwijamy **FreeRTOS** i uaktywniamy go (**Enabled**).

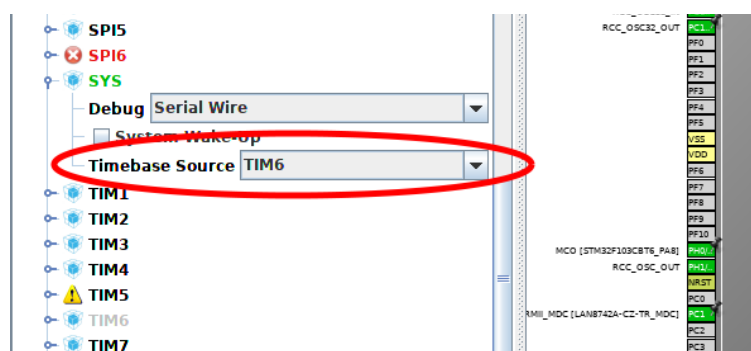


Warto zwrócić uwagę, że korzystając z nowoczesnych narzędzi, różnica w generowaniu ustawień początkowych dla urządzenia „bez systemu operacyjnego” a „z systemem operacyjnym” właściwie dotyczy wybrania jednej opcji przez interfejs graficzny. Należy mieć świadomość, że jeszcze kilka lat temu „dodanie” do

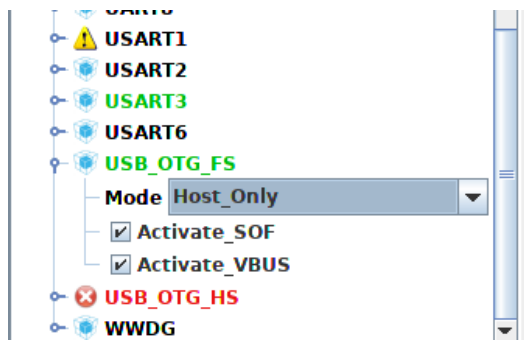
urządzenia systemu operacyjnego takiego jak FreeRTOS nie było takie proste, a na pewno nie sprowadzało się do dosłownie dwóch kliknięć w GUI.

12.2 Konfiguracja modułów peryferyjnych (sekcja *Pinout*)

- Z listy modułów sprzętowych uaktywniamy kanał debugowania: rozwijamy **SYS**, a następnie w polu **Debug**, wybieramy **Serial Wire** (JTAG nie jest dostępny na płytce -DISC1) oraz wskazujemy timer **TIM6** jako źródło podstawy czasu (**Timebase Source**). Układy TIM6 i TIM7 są najprostszymi timerami dostępnymi w stosowanym mikrokontrolerze. TIM6 i TIM7 są wystarczające do zastosowania jako podstawa czasu dla biblioteki HAL a z drugiej strony mają na tyle małą funkcjonalność, że nie stracimy cennych funkcji dostępnych tylko w niektórych timerach (np. w bardzo zaawansowanym TIM1). Sama zmiana źródła podstawy czasu jest konieczna ze względu na to, że włączyliśmy system operacyjny FreeRTOS domyślnie korzystający ze źródła SYSTICK i pozostawienie tego ustawienia prowadziłoby do konfliktu, który środowisko CubeMX zasygnalizowałoby przed generowaniem kodu. Po uaktywnieniu TIM6 jako **Timebase Source** przestaje on być dostępny na liście układów peryferyjnych (zostaje „wyszarzony” jak widać na rysunku poniżej).

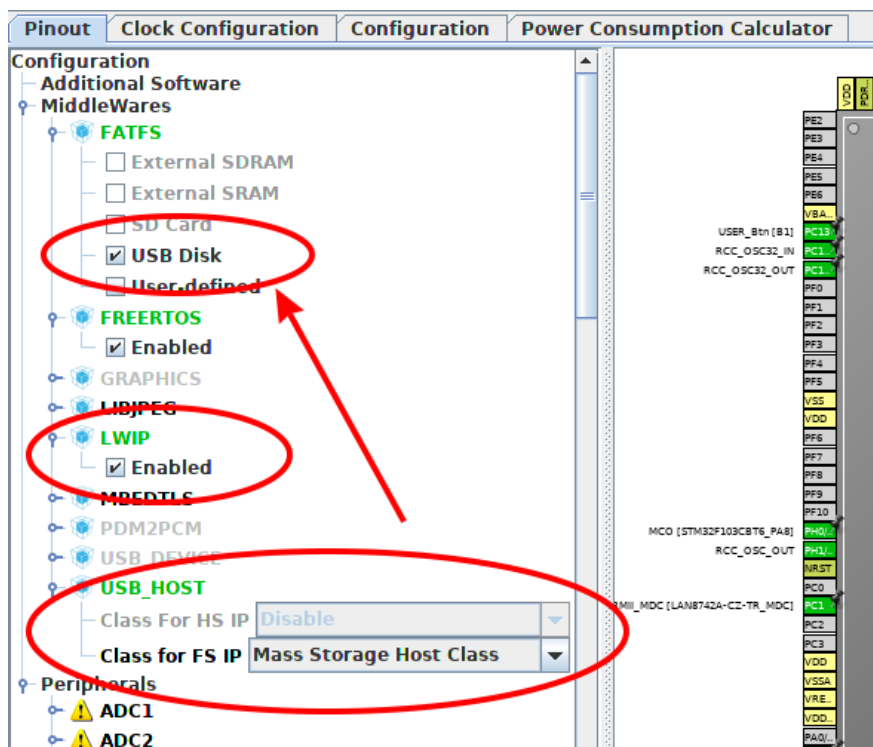


- W ćwiczeniu będziemy konfigurowali także moduł Host USB-OTG obsługujący dołączone urządzenia magazynujące. W tym celu najpierw konfigurowujemy moduł sprzętowy **USB_OTG_FS** (USB-OTG Full-speed) w trybie **Host_Only**, tj. do pracy tylko jako Host USB.



3. Przechodzimy do sekcji modułów programowych wyższego poziomu *MiddleWares*. Mamy tutaj możliwość dodania bibliotek przydatnych w wielu projektach w dziedzinie systemów wbudowanych. Do realizacji ćwiczenia potrzebna będzie obsługa klasy magazynującej USB, systemu plików FAT oraz stosu TCP/IP. Dlatego powracamy do sekcji Middlewares, a w niej wykonujemy następujące działania:

- W **USB_HOST** ustawiamy *Class for FS IP* na **Mass Storage Host Class**, dzięki czemu dodamy stos obsługi klasy magazynującej w trybie Host przez uprzednio włączony moduł USB-OTG.
- Po włączeniu obsługi pamięci masowej, mamy dostępną możliwość wybrania **USB Disk** jako urządzenia magazynującego obsługiwanego bibliotekę systemu plików FatFS (na liście widniejąca pod nazwą *FATFS*).
- Włączamy także obsługę stosu TCP/IP zaznaczając **Enabled** przy **LWIP**. LWIP (Light-weight IP) to bardzo popularna biblioteka obsługi nie tylko stosu TCP/IP wraz z popularnymi protokołami UDP i TCP, ale zawierająca także wiele przydatnych narzędzi komunikacji sieciowej i protokołów, np. DHCP, MQTT,



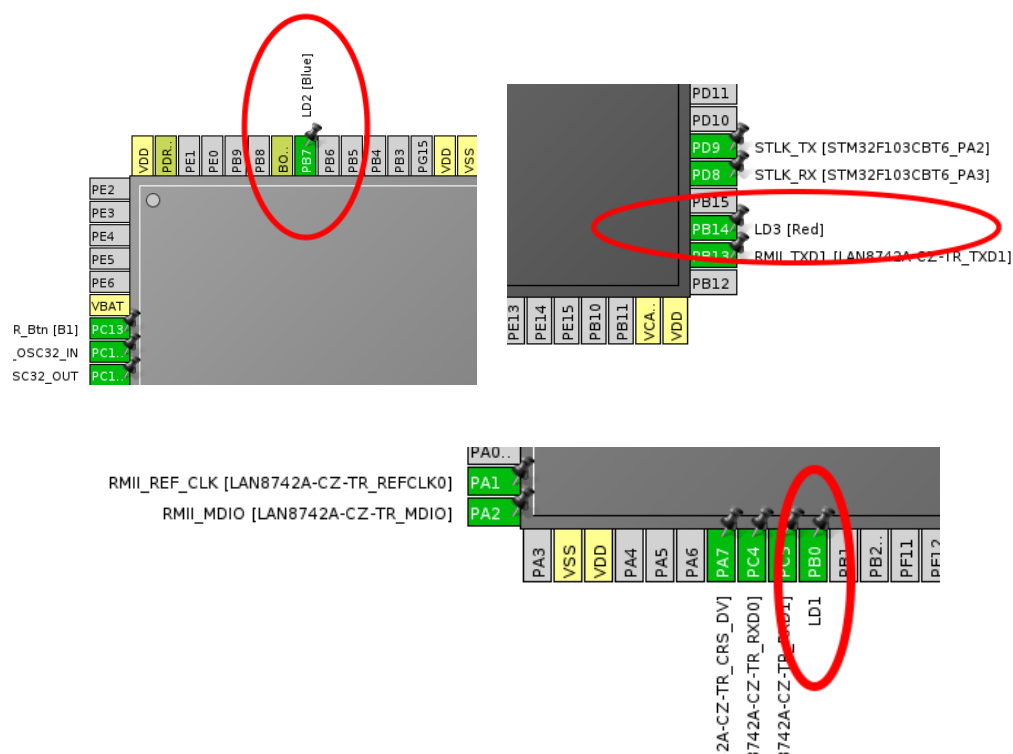
d) Zauważmy, że mamy wstępnie skonfigurowane wyprowadzenia dla dwóch diod LED na

płytkę testową:

1. LD2, niebieskiej, dołączonej do wyprowadzenia PB7,
2. LD3, czerwonej, dołączonej do wyprowadzenia PB14;

Obie diody LED są podłączone do wyprowadzeń wstępnie skonfigurowanych jako GPIO Output i po wygenerowaniu projektu możemy ich bez problemu użyć w swoim programie bez dodatkowej inicjalizacji.

3. Możemy również dodać obsługę diody LED LD1 (zielonej). W tym celu konfigurujemy wyprowadzenie **PB0** jako **GPIO_Output** (menu kontekstowe – lewy przycisk myszy) i nadajemy mu nazwę **LD1** (prawy przycisk myszy)

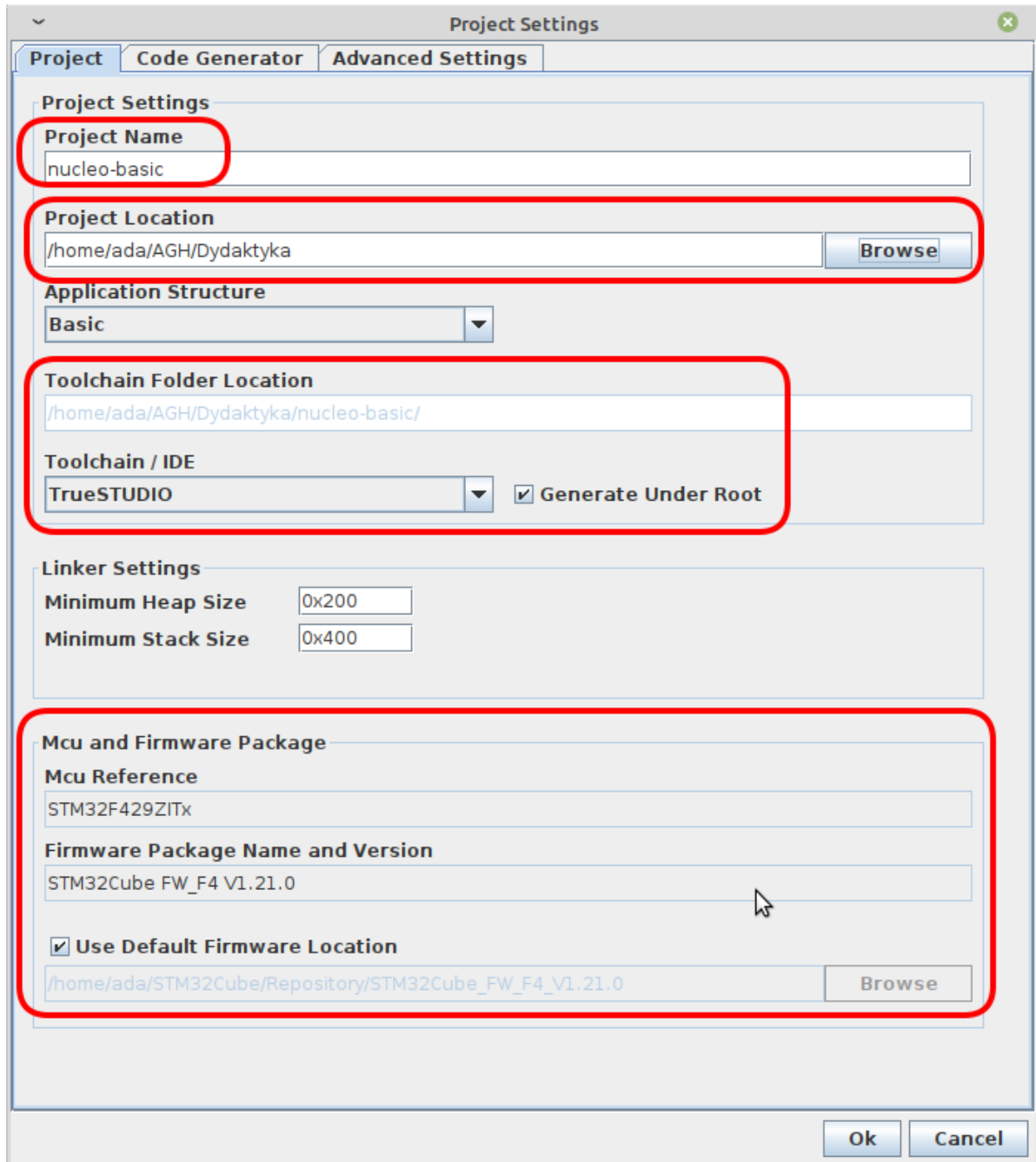


12.3 Ustawienia i zapisanie projektu

1. Na tym etapie jest zalecane jest zapisanie projektu, jednak najpierw potrzebne będzie wstępne skonfigurowanie ścieżek. W tym celu z menu głównego wybieramy **Project** → **Settings**.
 - a) Pojwi się okno ustawień projektu, w którym należy wybrać katalog dla projektu oraz wpisać nazwę głównego pliku konfiguracyjnego projektu (w przykładzie poniżej jest to **nucleo-basic**).
 - b) Zależnie od środowiska, w którym chcemy kontynuować pracę, wybieramy odpowiednią opcję z listy **Toolchain / IDE** – tutaj zalecane jest wybranie **TrueSTUDIO**. **Bardzo zalecane jest pozostawienie włączonej opcji **Generate Under Root****. Wyłączenie jej spowoduje wygenerowanie innej struktury katalogów, co nie uniemożliwi, ale utrudni

wykonanie ćwiczenia – instrukcja była pisana dla **włączonej** opcji **Generate Under Root**.

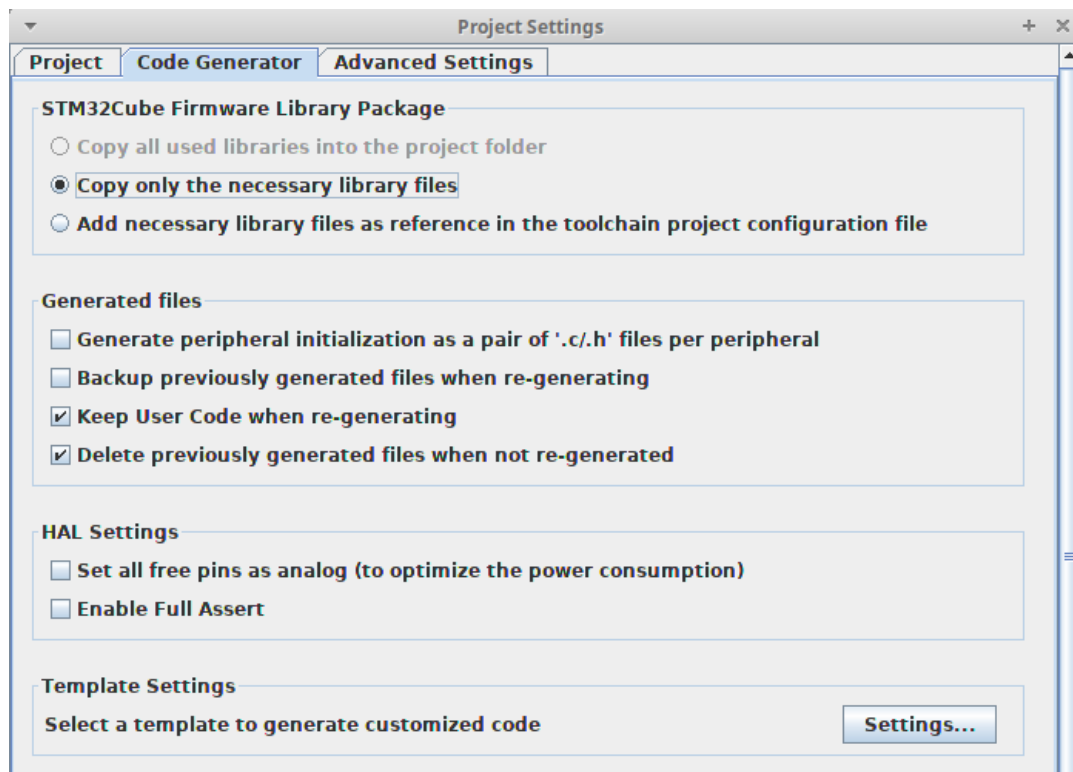
- c) Ważny jest także wybór wersji kodu (tutaj *Firmware*), z którego wygenerowany zostanie projekt bazowy. Nie wszystkie wersje firmware działają bez problemów. Niniejsza instrukcja była testowana na firmware **FW_F4 V1.21.0**.



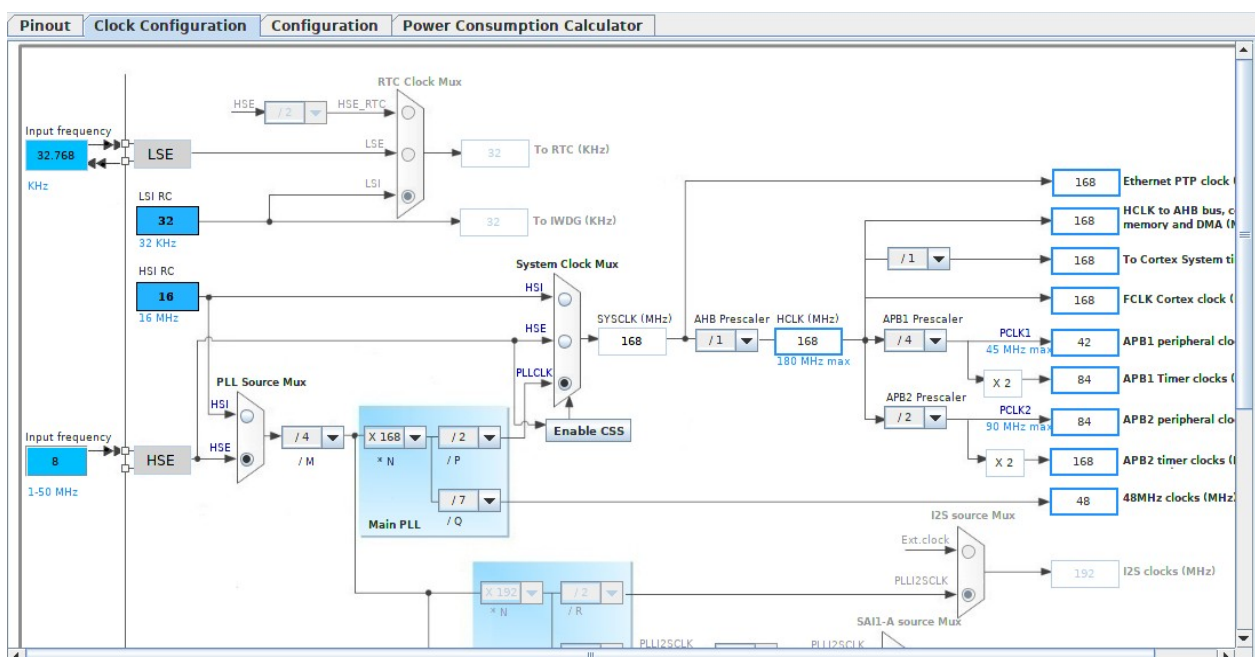
2. W oknie *Project Settings* warto także sprawdzić ustawienia w zakładce **Code Generator**. Przedstawione poniżej opcje zapewnią:
- a) skopiowanie wymaganych plików do katalogu projektu (właściwie znajdują się tam też dodatkowe pliki, ale nie ma to wpływu na działanie projektu),
 - b) zachowanie własnego kodu przy ponownym generowaniu projektu – jest to opcja

przydatna na przyszłość, gdy zechcemy zmodyfikować ustawienia projektu w STM32CubeMX.

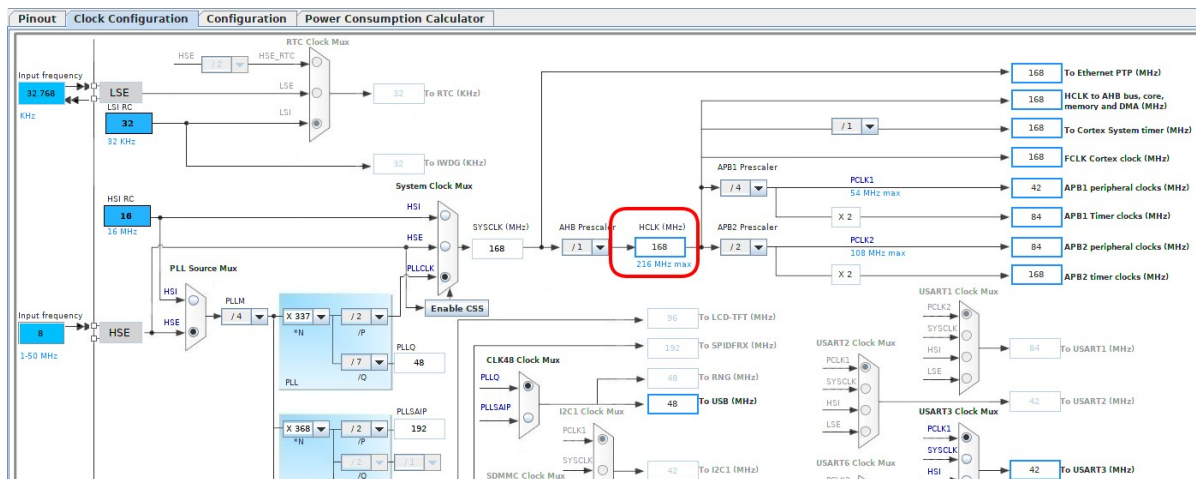
3. Jeśli uznamy, że ustawienia projektu są poprawne, zatwierdzamy przyciskiem OK i potwierdzamy nadpisanie pliku projektu.



4. Z menu głównego wybieramy **File** → **Save Project** i kontynuujemy prace nad projektem.
5. Przechodzimy do zakładki **Clock Configuration**. Widzimy teraz schemat wewnętrzny obwodów zarządzania sygnałem zegarowym w mikrokontrolerze.



a) W przypadku NUCLEO-F429ZI wystarczy jedynie sprawdzić ustawienia.



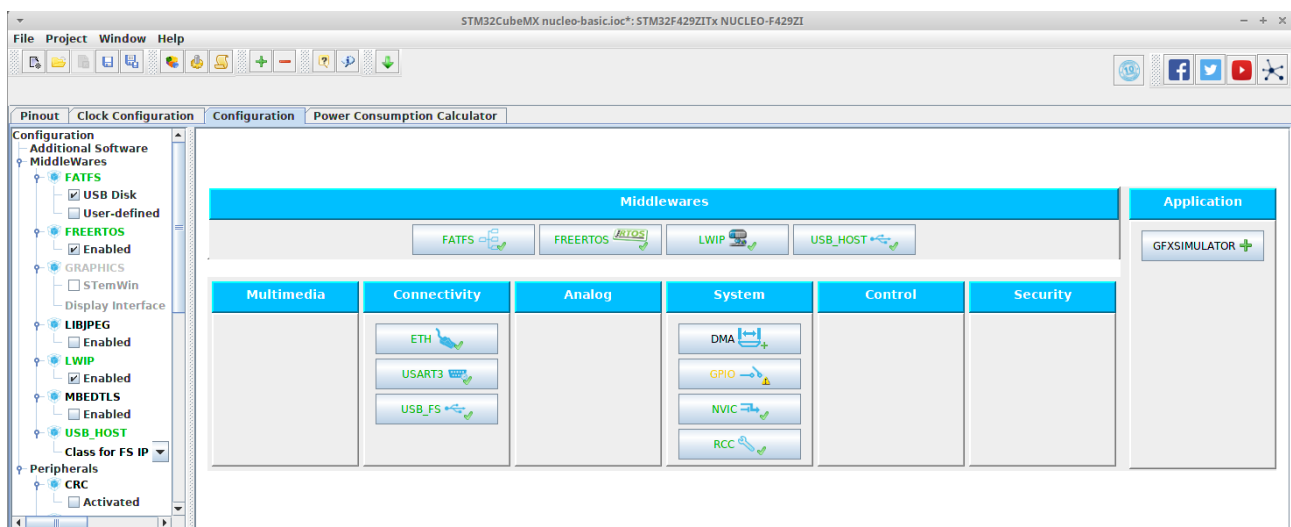
- b) W przypadku NUCLEO-F746ZG domyślnie częstotliwość HCLK jest dużo niższa od maksymalnej. Dla uzyskania zgodności z wersją F4 **zmieniamy ją na 168 MHz** – taką samą jak ustawiana domyślnie dla NUCLEO-F429ZI.

Ewentualne poważne błędy są zgłaszane przez aplikację STM32CubeMX w formie komunikatu, który trudno przeoczyć.

Warto zwrócić uwagę na nietypowy sposób taktowania mikrokontrolera na zastosowanej platformie sprzętowej. Do modułu HSE typowo dołącza się rezonator kwarcowy. Tutaj jednak do HSE dostarczony zostaje „gotowy” cyfrowy sygnał zegarowy o częstotliwości 8 MHz generowany przez zintegrowany na płycie moduł programatora. Takie rozwiązanie najprawdopodobniej podyktowane jest aspektami ekonomicznymi: w takiej konfiguracji jeden rezonator kwarcowy może zapewnić taktowanie zarówno programatora jak i głównego mikrokontrolera.

12.4 Konfiguracja sterowników i bibliotek

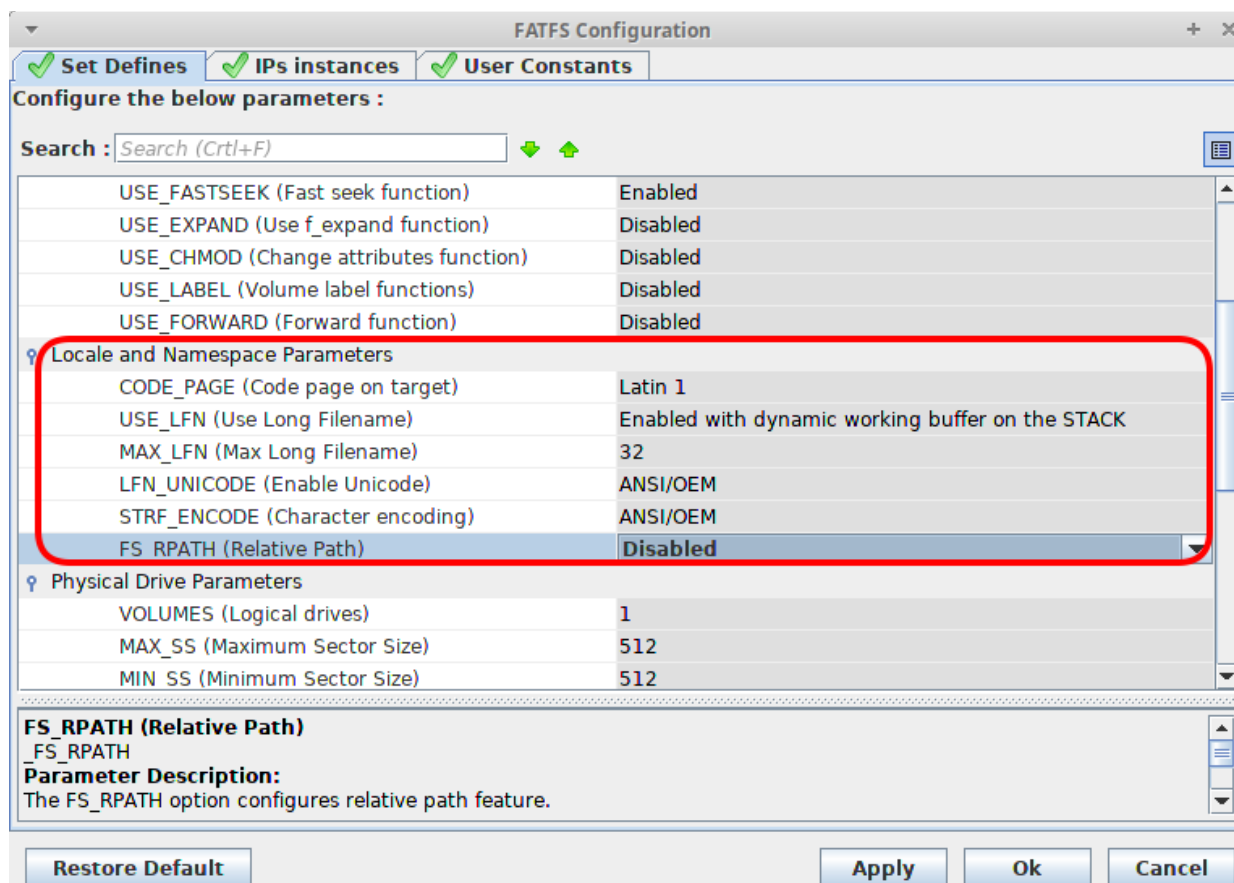
Dodane do projektu moduły sprzętowe (*Peripherals*) i programowe (*Middlewares*) mają wiele opcji konfiguracyjnych, które również ustawiamy w STM32CubeMX. W wersjach tej aplikacji niższych niż 5.0, są one dostępne w zakładce **Configuration** okna głównego.



1. Konfigurację można zacząć od interfejsu Ethernet (przycisk **ETH** panelu *Connectivity*).

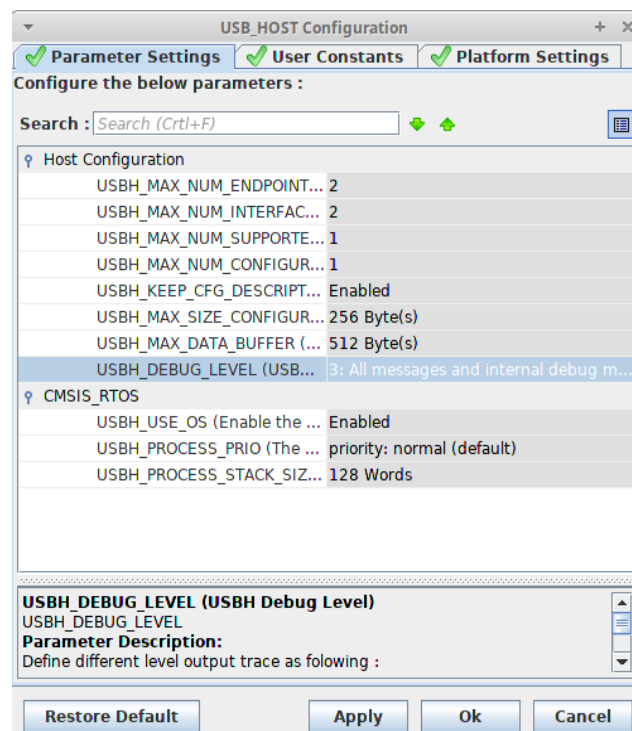
Głównym elementem, który warto zmienić, to adres MAC wynoszący domyślnie `00:80:E1:00:00:00`. Aby wiele osób w laboratorium mogło korzystać z przydzielania adresów IP przez DHCP, warto zadbać o ustawienie ich na unikalne wartości, tj. w tym przypadku wymyślić liczbę, która nie będzie się powtarzała wśród innych zespołów pracujących na płytce NUCLEO w tym samym czasie i w tej samej sieci lokalnej. Istnieją oczywiście sposoby, aby wygenerować tę liczbę tak, aby była unikalna (przynajmniej z dużym prawdopodobieństwem), jednak nie będziemy się tym zajmować w niniejszej instrukcji.

2. Kolejnym elementem, który należy skonfigurować, jest biblioteka FatFS (przycisk **FATFS**). Tutaj mamy wiele opcji konfiguracyjnych, jednak przede wszystkim należy włączyć obsługę długich nazw plików i katalogów (Long File Name, LFN) oraz skonfigurować kodowanie znaków. Sprawdzoną opcją jest tutaj **ANSI/OEM** dla kodowania **LFN_UNICODE**. Opcja **STRF_ENCODE** dotyczy funkcji działających na łańcuchach znakowych w plikach. Od **MAX_LFN** zależy maksymalna długość obsługiwanych długich nazw, jednak w celu zaoszczędzenia pamięci można rozważyć ograniczenie jej do kilkudziesięciu znaków (jak poniżej – **32 znaki**). Po edycji zatwierdzamy przyciskiem **OK**.

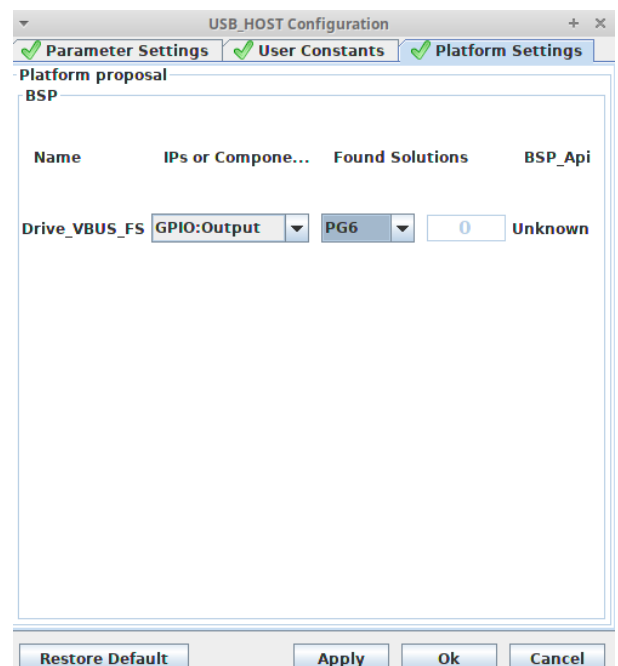
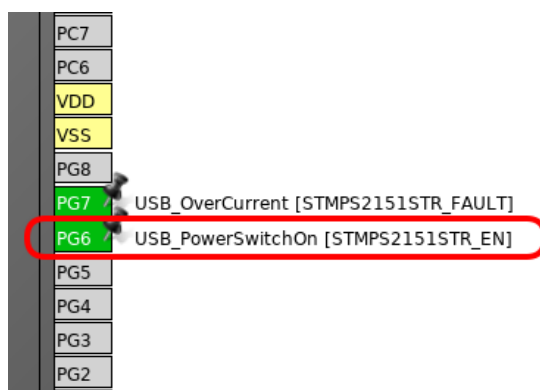


3. Przechodzimy do edycji parametrów sterownika USB-OTG w trybie Host (przycisk **USB_HOST**). Tutaj warto ustawić następujące parametry:
 - a) W zakładce Parameter Settings przydatne może być ustawienie **USBH_DEBUG_LEVEL** (*USBH Debug Level*) umożliwiające wyświetlanie komunikatów diagnostycznych przy pomocy funkcji printf. Im wyższy poziom, tutaj ustawimy, tym więcej komunikatów

możemy otrzymywać. Na poniższym zrzucie ekranu włączono wszystkie komunikaty diagnostyczne.



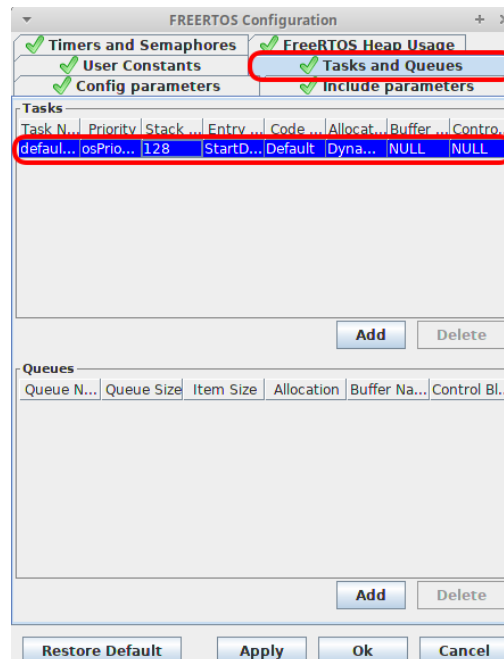
- b) W zakładce **Platform Settings** zalecane jest wskazanie wyprowadzenia sterującego zasilaniem urządzenia podłączonego do portu USB. W konfiguracji domyślnej na ekranie głównym to wyprowadzenie zostało zaznaczone jako **USB_PowerSwitchOn** i wg projektu płytki testowej zostało ono podłączone do **PG6** mikrokontrolera. Wybieramy je spośród dostępny w kategorii **GPIO:Output**. Na koniec zatwierdzamy przyciskiem **OK**.



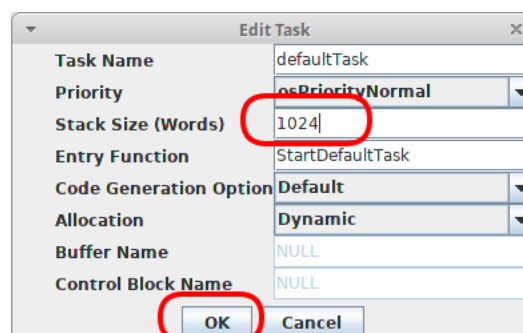
4. W wielu przypadkach zalecane jest zwiększenie domyślnego rozmiaru stosu dla głównego

zadania systemu operacyjnego **StartDefaultTask**. W tym celu w zakładce *Configuration* klikamy przycisk **FREERTOS**.

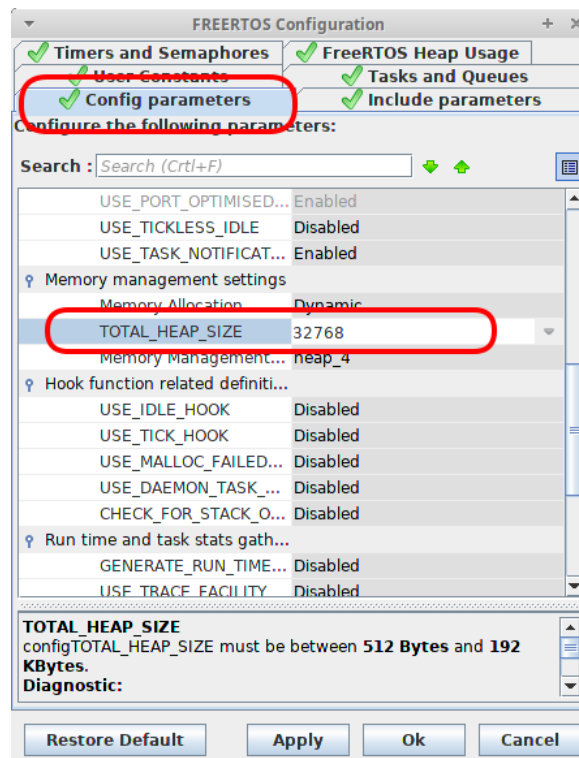
- a) Pojawi się okno *FreeRTOS Configuration*. Ustawienie parametrów zadań (tu tylko jednego konfigurowalnego) znajdziemy w zakładce *Tasks and Queues*.



- b) Wystarczy teraz kliknąć dwa razy na jednym zadaniu na liście (*defaultTask*), by otworzyć okienko konfiguracji jego parametrów. W nim z kolei można zmienić jego rozmiar stosu liczony w słowach 32-bitowych. Do typowych eksperymentów powinien wystarczyć rozmiar **1024 słowa** (tj. 4096 bajtów).



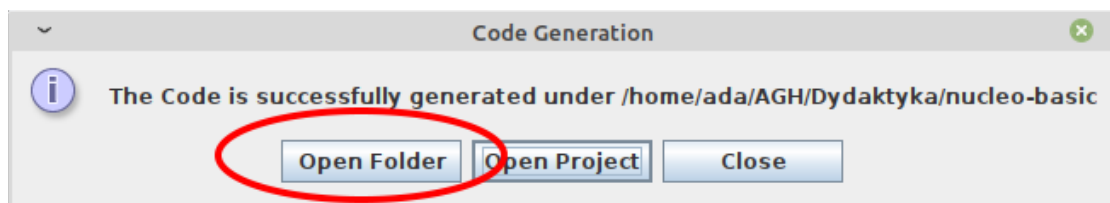
5. Gdy potrzebujemy wielu zadań wymagających dużego rozmiaru stosu, zalecane jest zwiększenie całkowitego dostępnego obszaru heap dla systemu operacyjnego FreeRTOS: *TOTAL_HEAP_SIZE*. Tę opcję konfiguracyjną znajdziemy w zakładce *Config Parameters*. W omawianym projekcie można bez negatywnych konsekwencji zwiększyć ten rozmiar do **32 KiB**.



12.5 Generowanie kodu projektu

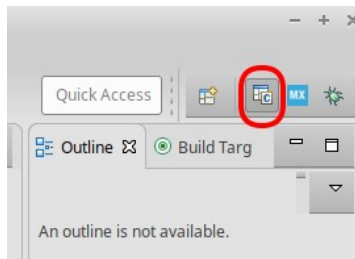
Po zakończonej konfiguracji zapisujemy projekt w STM32CubeMX i generujemy kod dla wybranego środowiska. Kod można wygenerować m.in. wybierając z menu głównego **Project** → **Generate Code**.

Po prawidłowym wygenerowaniu kodu zostaniemy zapytani, co zrobić dalej. Jeśli planujemy dalej edytować projekt w STM32CubeIDE, nie korzystamy z domyślnej opcji *Open Project*, tylko wybieramy **Close** lub **Open Folder**, aby wygodniej było podejrzeć zawartość katalogu z wygenerowanym kodem.

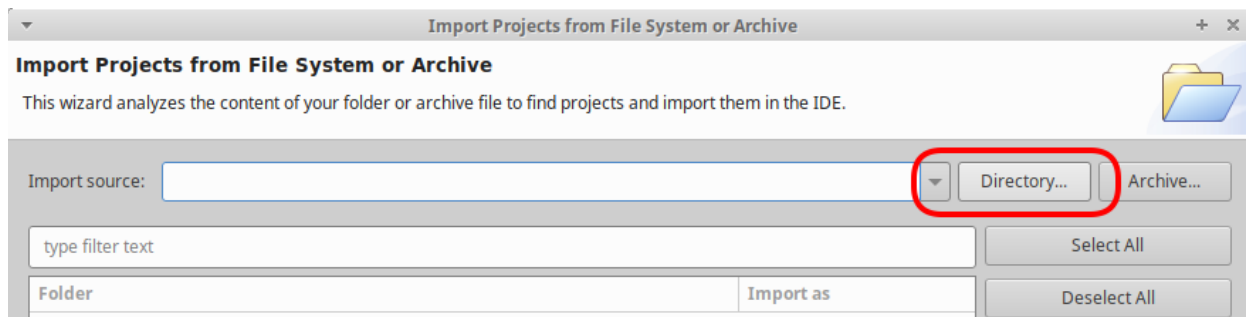


13. Otwarcie STM32CubeIDE i import projektu

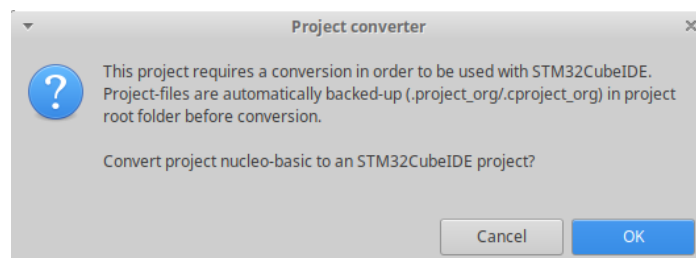
1. Włączamy oprogramowanie STM32CubeIDE. Domyślnie powinien pojawić się dialog wyboru *workspace*. Tutaj można wybrać domyślny *workspace* i wcisnąć przycisk **Launch**.
2. W głównym oknie CubeIDE sprawdzamy, czy wybrana jest perspektywa C/C++ i w razie potrzeby wybieramy ją:



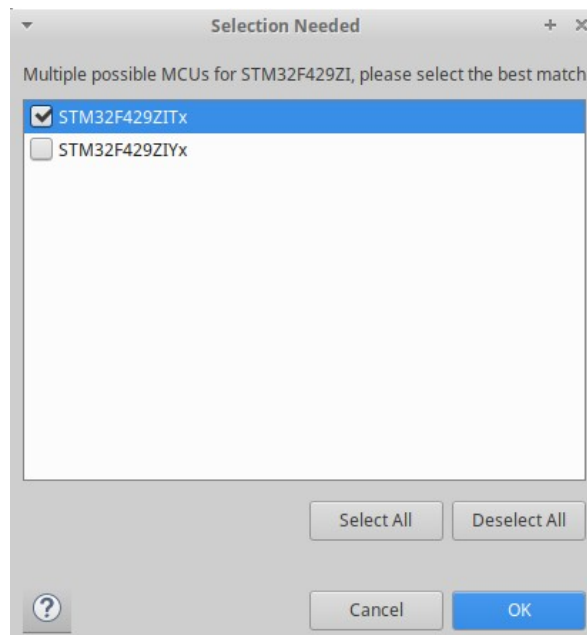
3. Aby rozpocząć import projektu, z menu głównego wybieramy **File** → **Open Projects from File System**. Pojawi się okno importowania, w którym wskazujemy katalog z uprzednio wygenerowanym w STM32CubeMX projektem.



4. Zatwierdzamy przyciskiem **Finish**.
5. Pojawi się dialog informujący, że jest potrzebna konwersja (projekt był wygenerowany dla środowiska TrueSTUDIO). Zgadamy się na nią przyciskiem **OK**.



6. Do konwersji będzie potrzebne doprecyzowanie, dla której wersji mikrokontrolera był generowany kod.
 - a) W przypadku NUCLEO-F429ZI wybieramy wersję **STM32F429ZITx**.
 - b) W przypadku NUCLEO-F746ZG wybieramy **STM32F746ZGTx**.
 - c) Zatwierdzamy **OK**.



7. Zakończenie konwersji zostanie podsumowane również odpowiednim komunikatem.

