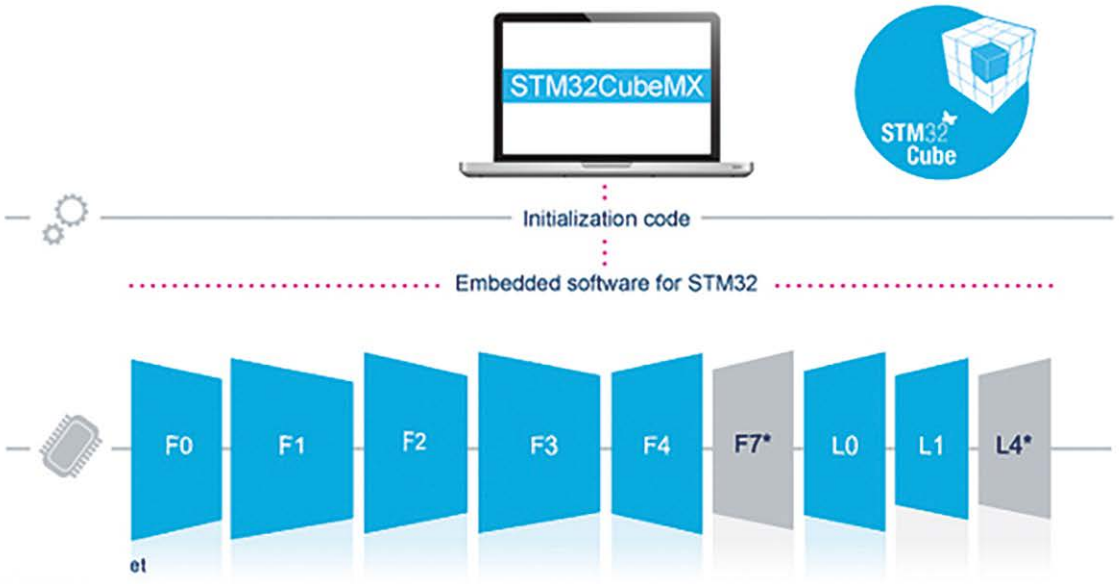


STM32Cube – ułatwienie tworzenia oprogramowania

Mikrokontrolery STM32 tworzą ogromną rodzinę procesorów z rdzeniem ARM Cortex. W jej skład wchodzą zarówno nieskomplikowane jednostki (np. STM32F0), jak i zaawansowane (np. STM32F4), czy najnowsze STM32F7 z rdzeniem Cortex- M7. Powoduje to pewne utrudnienia na wstępnym etapie tworzenia programu, w których rozwiązaniu pomaga STM32Cube.

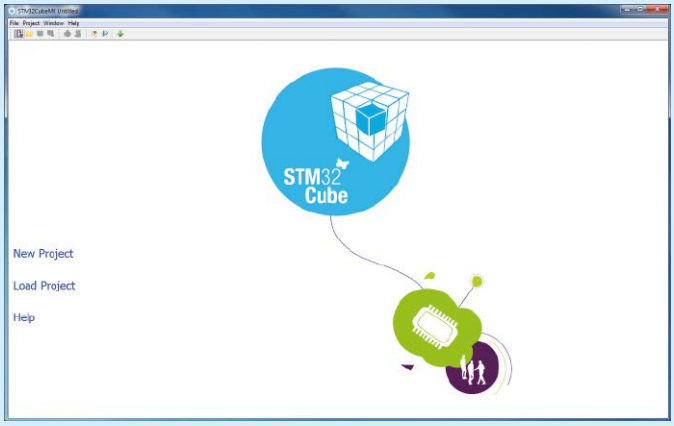
Mikrokontrolery 32-bitowe są przeznaczone do bardziej wymagających zadań sterowania i kontroli i z tego powodu wyposażono je w wiele modułów funkcjonalnych i konfigurowalny, rozbudowany układ taktowania. Z powodu różnych zadań stawianych mniej i bardziej zaawansowanym konstrukcjom nie jest możliwe, aby konfigurowanie na przykład modułów peryferyjnych w STM32F0 mogło odbywać się w taki sam sposób, jak STM32F4. Również w obrębie tej samej linii produktów mogą występować spore różnice wynikające z wyposażenia poszczególnych układów. To wszystko powoduje, że wstępny etap tworzenia oprogramowania polegający na konfigurowaniu mikrokontrolera, tj. funkcji wyprowadzeń, kontrolera przerwań, obwodów taktowania oraz napisaniu prostych driverów urządzeń peryferyjnych może być bardziej pracochłonny, niż wykonanie właściwej aplikacji. Szczególnie uciążliwe jest zapoznawanie się z dokumentacją zawierającą opis rejestrów konfiguracyjnych i wielu programistów nie miałoby nic przeciw temu, aby ktoś wykonał za nich ten etap pracy.

Programistom Corteksów oddano do dyspozycji bibliotekę CMSIS (*Cortex Microcontroller Software Interface Standard*). Tworzy ona uniwersalny interfejs programowy, który umożliwia obsługę układów peryferyjnych i rdzenia Cortex, wykorzystując do tego celu zestandaryzowane funkcje, makra i definicje. Funkcje CMSIS wspierają też użycie systemów operacyjnych czasu rzeczywistego, middleware, oraz aplikacji używających interfejsów komunikacyjnych np. I²C, UART, SPI, Ethernet itp. Każdy z producentów mikrokontrolerów dostarcza swoje implementacje bibliotek z funkcjami obsługującymi własne peryferia.

Idea CMSIS opiera się na założeniu, że programy pisane dla mikrokontrolerów z rdzeniem Cortex wytwarzanych przez różnych producentów mogą być łatwo przenoszone na różne platformy sprzętowe. Drugim założeniem było uproszczenie etapu konfigurowania i inicjowania modułów peryferyjnych oraz rdzenia. Żeby było to możliwe, funkcje CMSIS muszą charakteryzować się pewnym stopniem abstrakcji. Oznacza to, że „ukrywają" przed programistą działania na rejestrach konfiguracyjnych. Tworzy się w ten sposób umowna warstwa HAL (*Hardware Abstraction Layer*).

Takie podejście zostało różnie odebrane w środowisku programistów. Część z nich zaczęła chętnie stosować gotowe biblioteki, ale dla innych zastosowanie CMSIS było tylko niepotrzebnym komplikowaniem oprogramowania. Wątpliwości czy niechęć wynikały po części z przyzwyczajeń i chęci panowania nad wszystkim, co się dzieje w programie. Trochę przypominało to przejście od programowania w asemblerze do programowania w C.

Innym dużo poważniejszym argumentem przeciwko stosowaniu CMSIS była spora liczba błędów i nadmierne wykorzystywanie zasobów mikrokontrolera. Dodatkowo, procedury obsługi modułów peryferyjnych potrafiły



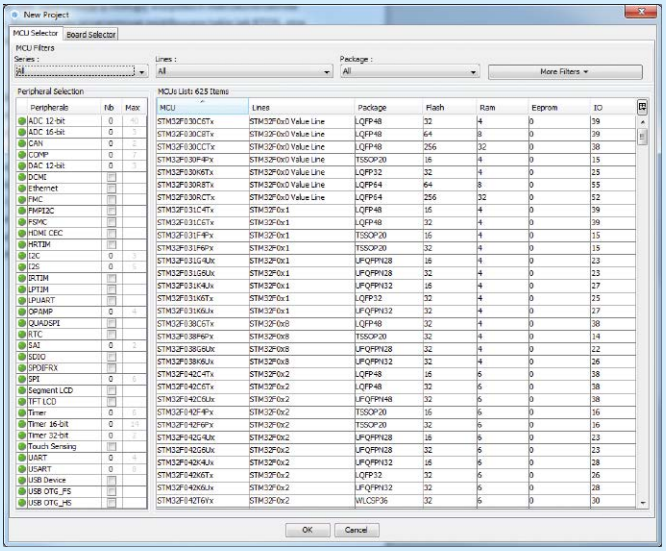
Rysunek 1. Okno startowe STM32CubeMX

czekać w nieskończoność na zmianę stanu jakiegoś bitu skutecznie zatrzymując działanie całej aplikacji. Dlatego znaczna cześć programistów wybrała rozwiązanie pośrednie: patrzymy jak to zrobiono w CMSIS i na tej podstawie robimy po swojemu.

Ponieważ CMSIS jest dostępne już od wielu lat, to większość błędów zostało zauważonych i poprawionych, a funkcje zoptymalizowane i dostosowane do wykorzystania w różnych konfiguracjach implementacji interfejsu API: polling, przerwania lub DMA. Jednak samo użycie bibliotek, chociaż sporo ułatwia (przy założeniu, że nie mają błędów), to nie zwalnia w całości od konieczności ręcznego skonfigurowania mikrokontrolera. I tutaj z pomocą przychodzi oprogramowanie STM – STM32Cube.

Ideą przyświecającą powstaniu STM Cube było ułatwienie zadania zastosowania mikrokontrolera przez znaczne zredukowanie wysiłku włożonego w projekt i tym samym ograniczenie czasu potrzebnego na jego wykonanie. Oprogramowanie STM32Cube składa się z:

- Graficznego środowiska STM32CubeMX generującego kod w języku C na podstawie projektu.
- Obszernej platformy programowej dostarczanej osobno dla każdej z rodzin mikrokontrolerów i zawierającą warstwę STM32Cube HAL zapewniającą obsługę wszystkich mikrokontrolerów STM32 oraz zgodne z tą warstwą komponenty programowe middleware, takie jak: RTOS, stos USB, stos TCP/IP i biblioteka funkcji graficznych.



Rysunek 2. Okno wyboru mikrokontrolera

Pracę z STM32Cube rozpoczynamy od pobrania ze strony stm.com i zainstalowania pakietu STM32CubeMX. Jak w innych tego typu programach narzędziowych praca odbywa się w oparciu o plik projektu. Na ekranie powitalnym pokazanym na **rysunku 1** są wyróżnione dwie akcje:

1. Tworzenie nowego projektu – New Project.
2. Otwieranie wcześniej zapisanego projektu – Load Project.

Zaczynamy od tworzenia nowego projektu. Jednym z pierwszych zadań konstruktora urządzeń embeded jest wybór mikrokontrolera. Nie jest to zadanie łatwe nawet wtedy, gdy zdecydowaliśmy się na mikrokontroler rodziny z STM32. Portfolio wyrobów tego producenta jest ogromne i składa się z wielkiej liczby produkowanych układów, różniących się nie tylko wyposażeniem w moduły peryferyjne, ale także wielkością pamięci i architekturą rdzenia. Dlatego nawet przejrzenie jedynie pierwszych stron kart katalogowych może zabrać sporo czasu. Oprogramowanie STM32CubeMX pomaga w poruszaniu się w gąszczu możliwości oferując rozbudowany moduł selektora mikrokontrolerów (**rysunek 2**).

Okno selektora ma dwie zakładki: *MCU Selector* i *Board Selector*. Wybierając *MCU Selector* można filtrować potrzebne zasoby wybierając:

- Serię mikrokontrolerów: STM32F0, STM32F1, STM32F2, STM32F3, STM32F4, STM32L0,i STM32L1.
- Linię produktów. Na przykład, STM32F0x0 Value Line.
- Obudowę układu. Na przykład, LQFP64.
- Inne filtry. Na przykład, ze względu na liczbę wyprowadzeń I/O.

Jako alternatywę dla filtrów można w oknie *Peripherial Selection* wybrać mikrokontroler ze względu na rodzaj i liczbę układów peryferyjnych. Zakładka *MCU Selector* pozwala na znaczne zawężenie obszaru poszukiwań i w konsekwencji wybranie mikrokontrolera odpowiedniego dla urządzenia.

Duga zakładka – *Board Selector* – jest używana, gdy chcemy wykonać projekt w oparciu o płytkę ewaluacyjną oferowaną przez STM. Zestawów ewaluacyjnych jest sporo. Zgodnie z intencją producenta podzielono je na trzy grupy:

- Płytki „Nucleo" z mikrokontrolerami z serii STM32F0, STM32F3, STM32F4, STM32L0, STM32L1.
- Płytki „Discovery" z mikrokontrolerami z serii STM32F0, STM32F3, STM32F4, STM32L0, STM32L1.
- Płytki „Eval Board" z mikrokontrolerami z serii STM32F0, STM32F1, STM32F2, STM32F3, STM32F4, STM32L1

Podobnie jak w zakładce *MCU Selector*, tak i tu można filtrować ustalając odpowiednie kryteria: producenta (na razie tylko STM), rodzaju płytki (*Nucleo*, *Discovery*, *Eval*) i serię MCU (STM32F0, STM32F1 i inne). Dodatkowo, wybór płytki ułatwia zdjęcie wyświetlające się po prawej stronie okna (**rysunek 3**).

Dla potrzeb przetestowania *STM32 Cube* wybrałem moduł **STM32F4Discovery** z mikrokontrolerem **STM32F407VG**. W zakładce *Peripherial Selection* można zobaczyć (oznaczone na zielono) układy peryferyjne możliwe do wykorzystania: akcelerometr, układ audio, przyciski, kompas, żyroskop, interfejs USB. Pod zdjęciem

modułu są wyświetlone jego podstawowe parametry, a po kliknięciu na przycisk *Load User Manual* jest automatycznie pobierana i otwierana instrukcja użytkownika. Dodatkowo, po kliknięciu na *Link to ST WebSite* otwiera się firmowa strona z opisem modułu i wszystkim dostępnymi materiałami, takimi jak: noty aplikacyjne, przykładowe programy, schematy itp.

Zatwierdzenie wybranej płytki powoduje przejście do kolejnego etapu tworzenia projektu. Otwiera się okno główne projektu z zakładkami: *Pinout*, *Clock Configuration*, *Configuration*, *Power Consumtion Calculator*.

Za pomocą zakładki Pinout nadaje się wyprowadzeniom mikrokontrolera funkcje alternatywne. Wyprowadzenie może być wejściem lub wyjściem cyfrowym, wejściem analogowym lub spełniać rolę wyprowadzenia sygnałów modułów peryferyjnych. Okno zakładki *Pinout* składa się z rozwijanej listy z nazwami modułów peryferyjnych i okna z narysowaną obudową mikrokontrolera z wyprowadzeniami (**rysunek 4**). Wyprowadzenia są oznaczane następującymi kolorami:

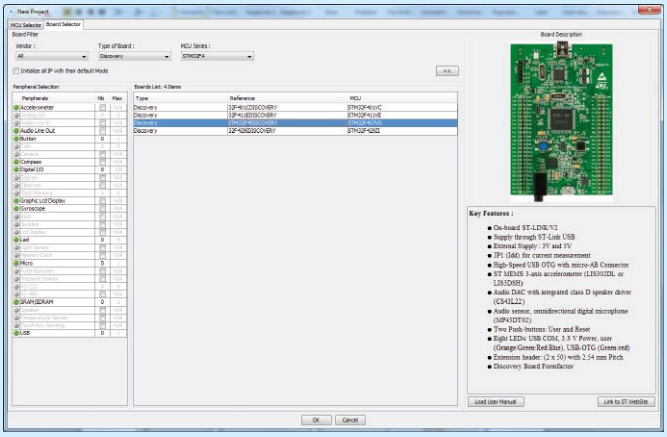
- Szarym, gdy do wyprowadzenia nie jest przypisana żadna funkcja.
- Zielonym, gdy do wyprowadzenia jest przypisana i zdefiniowana funkcja (na przykład sygnał modułu I²C lub wejście/wyjście GPIO).
- Pomarańczowym, gdy do modułu jest przypisana funkcja, ale nie jest aktywna.
- Żółtym – wyprowadzenia zasilania.

Najłatwiej konwencję kolorowania wyprowadzeń pokazać na przykładzie. W tym celu:

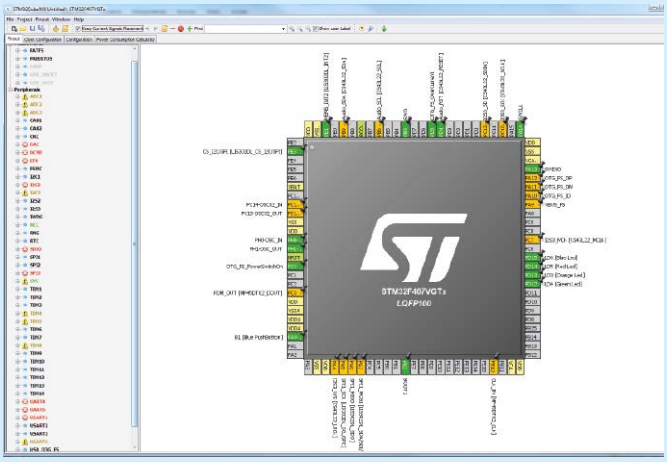
1. Klikamy na wyprowadzenie PC5 i wybieramy *GPIO_Input*. Ponieważ funkcja linii wejścia/wyjścia nie wymaga dodatkowej konfiguracji, to wyprowadzenie jest wyświetlane w kolorze zielonym.
2. Jeżeli chcemy, aby wyprowadzenie PC5 było wejściem przetwornika A/C, to ponownie na nie klikamy i wybieramy *ADC1_IN15*. Teraz wyprowadzenie jest wyświetlane na na pomarańczowo.
3. Aby wyprowadzenie zdefiniowane jako *ADC1_IN15* zostało wyświetlone na zielono, w oknie *Configuration* trzeba wybrać *ADC1* i zaznaczyć *IN15* (**rysunek 5**).

W taki sam sposób są konfigurowane pozostałe wyprowadzenia używane w projekcie. Każdy układ peryferyjny prawidłowo przypisany do wyprowadzenia jest wyświetlany na zielono w oknie *Configurations* zakładki *Pinout*. Ma to też inne konsekwencje: prawidłowo przypisany do wyprowadzenia moduł peryferyjny zostanie wyświetlony w oknie *Configuration*, w którym można go dokładnie skonfigurować. Definiowanie przypisania w oknie *Pinout* nie pozwala na przypisanie funkcji alternatywnej już zajętemu wyprowadzeniu. W naszym przykładzie w konfiguracji przypisania przetwornika ADC1 są zaznaczone na czerwono wejścia IN0, IN4. IN5, IN6, IN7, IN10 oraz IN3. Na przykład, wyprowadzenie PA4 jest już przypisane do *I2S3_WS* i nie może pełnić roli wejścia *ADC1_IN4*. STM32CubeMX skutecznie zabezpiecza przed próbą przypisania do jednego wyprowadzenia funkcji więcej niż jednego modułu peryferyjnego.

Po przypisaniu do wyprowadzenia PC5 wejścia IN15 przetwornika ADC1 przechodzimy do zakładki *Configuration*, w której przetwornik zostanie skonfigurowany. Zakładka ta jest podzielona na dwa okna.



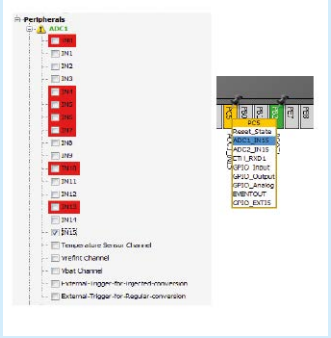
Rysunek 3. Okno wyboru płytki ewaluacyjnej



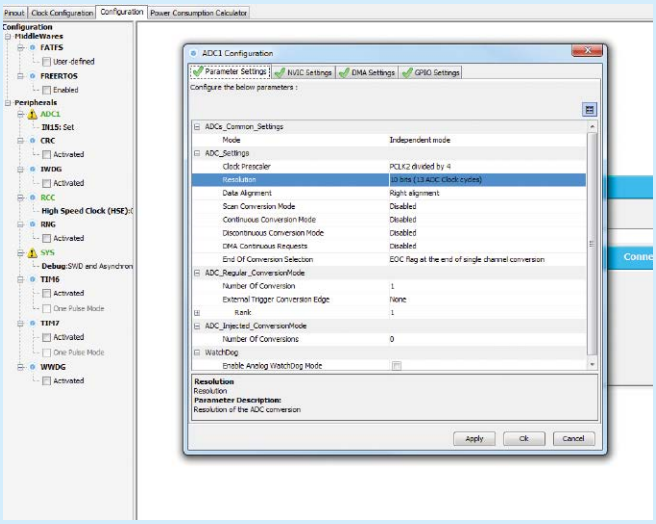
Rysunek 4. Okno *Pinout*

W oknie *Configurations* są wyświetlone wszystkie elementy, które mogą zostać skonfigurowane w projekcie. W przykładowym projekcie są to elementy Middleware FATFS i FREERTOS oraz moduły peryferyjne możliwe do skonfigurowania, w tym również przetwornik ADC1. Drugie okno zawiera pola: *Middlewares*, *Multimedia*, *Control*, *Analog*, *Connectivity* i *System*. W polu *Analog* jest wyświetlana ikona przetwornika ADC1, a w polu *System* ikony: DMA, GPIO, NVIC i RCC. Żeby skonfigurować przetwornik trzeba kliknąć na ikonę ADC1. Wtedy w polu *Analog* zostanie wyświetlone okno *ADC1 Configuration* (**rysunek 6**).

Okno konfiguracji ma cztery zakładki: *Parameter Settings*, *NVIC Settings*, *DMA Settings* i *GPIO Settings*. Za pomocą *Parameter Settings* ustawia się parametry pracy przetwornika, miedzy innymi: tryb pracy, częstotliwość taktowania (preskaler PCLK2), rozdzielczość w bitach, sposób umieszczenia danych w słowie (data Alignment) itp. Za pomocą zakładki NVIC odblokowuje się przerwania od modułów ADC1, ADC2 i ADC3. W zakładce DMA można zdefiniować kanał DMA do przesyłania danych konwersji bezpośrednio do pamięci. Ostatnia zakładka *GPIO Settings* jest przeznaczona do konfigurowania trybu pracy wyprowadzenia ADC1_IN15. Ponieważ w trakcie



Rysunek 5. Konfigurowanie wyprowadzenia PC5 w roli wejścia ADC1_IN15



Rysunek 6. Konfiguracja przetwornika ADC1

przypisywania wejście PC5 zostało skonfigurowane jako Analog Input, to w naszym przypadku nie potrzeba wykonywać tu żadnych zmian.

Można teraz zobaczyć jak STM32CubeMX poradzi sobie z wygenerowaniem plików źródłowych dla ustalonej konfiguracji przetwornika A/C. W pierwszym kroku trzeba określić sposób generowania plików źródłowych. Klikamy na *Project* → *Settings* (Alt+P). Trzeba tu ustawić:

- Nazwę projektu.
- Ścieżkę dostępu do katalogu projektu.
- Środowisko projektowe IDE.

STM32CubeMX może wygenerować kompletny projekt dla EWARM, uVision4 i uVision 5 (MDK-ARM), True STUDIO i SW4STM32 – wybrałem pakiet **uVision4** firmy Keil (ARM). W zakładce *Code Generator* okna *Project Settings* można zdecydować czy generator kodu skopiuje wszystkie biblioteki warstwy HAL do projektu, czy tylko skopiuje tylko te biblioteki, które są potrzebne po wygenerowaniu kodu przez STM32CubeMX.

W projektach generowanych przez tego typu aplikacje zawsze pojawia się problem, co zrobić z kodem już napisanym przez użytkownika w momencie, gdy ten zdecyduje się na zmianę konfiguracji modułów peryferyjnych lub middleware przez STM32CubeMX. Najlepiej, gdyby automatycznie nowo generowany kod nie niszczył kodu użytkownika, a tylko robił w plikach niezbędne zmiany konfiguracyjne. Dlatego w opcjach generowania kodu należy zaznaczyć *Keep User Code when re-generatting*. Pewnym ograniczeniem tej funkcji jest to, że kod przeznaczony do zachowania musi znaleźć się pomiędzy komentarzami /* USER CODE BEGIN */ i /* USER CODE END */. Jeżeli nasze procedury – łącznie z definicjami zmiennych – zostaną umieszczone poza komentarzami, to przy kolejnym generowaniu projektu z konfiguracją zostaną usunięte. Trzeba o tym pamiętać, żeby nie zniszczyć efektów swojej pracy.

Kod inicjalizacji przetwornika wygenerowany przez STM32CubeMX pokazano na **listingu 1**. Skonfigurowany przetwornik trzeba jeszcze zainicjalizować. Inicjalizacja będzie wymagała włączenia taktowania modułu A/C i ustawienia wyprowadzenia PC5 jako wejścia analogowego. STM32Cube MX automatycznie generuje funkcję inicjalizacyjną *HAL_ADC_MspInit* (Msp od *MCU support package*). Ta funkcja jest umieszczana przez generator w pliku *stm32f4xx_hal_msp.c*, a jej wywołanie wygląda następująco – *„HAL_ADC_MspInit(&hadc1);”*.

Listing 1. Konfiguracja przetwornika ADC1 wygenerowana przez STM32Cube MX

```
void MX_ADC1_Init( void)
{
    ADC_ChannelConfTypeDef sConfig;
/* Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion) */
    hadc1.Instance = ADC1;
    hadc1.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV4;
    hadc1.Init.Resolution = ADC_RESOLUTION10b;
    hadc1.Init.ScanConvMode = DISABLE;
    hadc1.Init.ContinuousConvMode = DISABLE;
    hadc1.Init.DiscontinuousConvMode = DISABLE;
    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.NbrOfConversion = 1;
    hadc1.Init.DMAContinuousRequests = DISABLE;
    hadc1.Init.EOCSelection = EOC_SINGLE_CONV;
    HAL_ADC_Init(&hadc1);
/* Configure for the selected ADC regular channel its corresponding rank in the sequencer and its sample
time. */
    sConfig.Channel = ADC_CHANNEL_15;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
}
```

Listing 2. Inicjalizacja przetwornika ADC1

```
void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if(hadc->Instance==ADC1)
    {
/* USER CODE BEGIN ADC1_MspInit 0 */
/* USER CODE END ADC1_MspInit 0 */
/* Peripheral clock enable */
        __ADC1_CLK_ENABLE();
/**ADC1 GPIO Configuration
        PC5     ------> ADC1_IN15
*/
        GPIO_InitStruct.Pin = GPIO_PIN_5;
        GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
/* USER CODE BEGIN ADC1_MspInit 1 */
/* USER CODE END ADC1_MspInit 1 */
    }
}
```

Kolejną ważną konfiguracją mikrokontrolera jest ustawienie taktowania. Układ taktowania nowoczesnych mikrokontrolerów jest bardzo rozbudowany w porównaniu z nieskomplikowanymi konstrukcjami sprzed kilkunastu lat. Możliwość zaprogramowania w szerokim zakresie częstotliwości taktowania rdzenia i osobno częstotliwości taktowania modułów peryferyjnych wymusiła zastosowanie programowanych układów PLL i programowych układów przełączających. Ponadto, można wybrać źródło zegara: z oscylatora kwarcowego z dołączanym zewnętrznym rezonatorem lub z wewnętrznego, precyzyjnego oscylatora RC o stałej częstotliwości. Ta elastyczność w konfigurowaniu taktowania jest okupiona skomplikowaniem układu i dlatego sprawia trudności nawet

Listing 3. Ustawienia taktowania mikrokontrolera

```
/* System Clock Configuration */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct;
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    __PWR_CLK_ENABLE();
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 8;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    HAL_RCC_OscConfig(&RCC_OscInitStruct);
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_SYSCLK|RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
    HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5);
}
```

bardziej doświadczonym programistom, nie mówiąc już o początkujących. Żeby zadanie konfiguracji uczynić tak łatwym, jak to tylko możliwe, w pakiecie STM32Cube MX umieszczono pokazaną na **rysunku 7** zakładkę *Clock Configuration* z graficznym przedstawieniem układu taktowania.

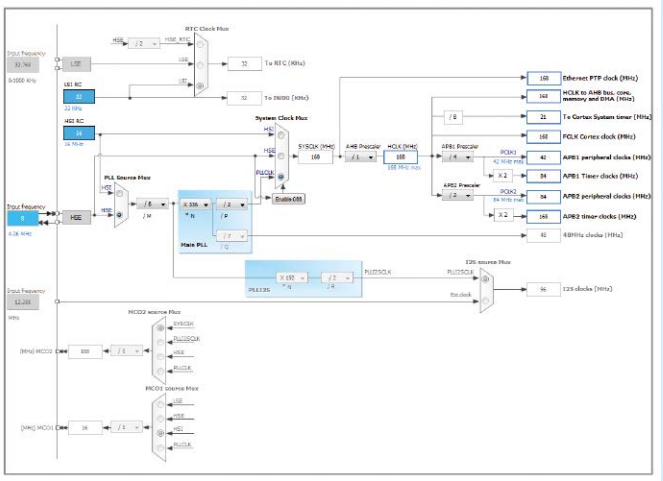
Domyślnie układ jest tak skonfigurowany żeby rdzeń był taktowany z maksymalną częstotliwością taktowania równą w tym wypadku 168 MHz. Źródłem sygnału zegarowego jest oscylator kwarcowy HSE o częstotliwości 8 MHz umieszczony na płytce STM32F4Discovery. Alternatywnie można wybrać oscylator RC HSI o częstotliwości taktowania 16 MHz. Wyboru oscylatora dokonuje się w bloku *PLL Source Mux*.

Wysoką częstotliwość taktowania uzyskuje się w układzie powielania częstotliwości z pętlą PLL (*Main PLL*), w którym częstotliwość wejściowa jest najpierw powielana przez 336, a potem dzielona przez 2. Sygnał z wyjścia układu PLL jest doprowadzony do wejścia selektora *System Clock Mux*.

Oprócz konfigurowania ścieżki sygnału taktowania zakładka *Clock Configuration* pokazuje na bieżąco, jakie częstotliwości są ustawione dla wszystkich bloków mikrokontrolera: rdzenia, magistrali AHB, modułów peryferyjnych APAB1, APB2 itp. Po wygenerowaniu kodu ustawienia z zakładki *Clock Configurator* są zapisywane w zamieszczonej na **listingu 3** funkcji *SystemClockConfig* dostępnej w pliku *main.c*.

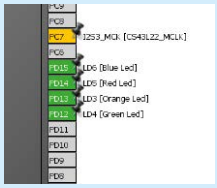
Moduł STM32F4Discovery ma zamontowane 4 diody LED dołączone do linii portu PD: zieloną do PD12, pomarańczową do PD13, czerwoną do PD14, niebieską do PD15. Jeżeli w projekcie wybierzemy tworzenie oprogramowania dla tego modułu, to w oknie Pinout wyprowadzenia tych linii portów zostaną zdefiniowane jako linie wyjściowe i opisane etykietami z nazwami diod, co pokazano na **rysunku 8**.

Spróbujemy teraz tak wykonać projekt, aby cyklicznie zaświecał i gasił jedną lub kilka diod. Do tego celu będzie nam potrzebne odliczanie opóźnienia. Można to zrobić programowo, ale lepszym wyjściem jest użycie licznika i systemu przerwań. Do odliczania opóźnień wybrałem licznik *TIM10*. Aby go użyć, w zakładce *Pinout* trzeba wybrać *TIM10* i zaznaczyć *Activated*. Wtedy ikona *TIM10* zostanie wyświetlona w okienku *Control* zakładki *Configuration*, jak na **rysunku 9**. Wybrany licznik TIM 10 trzeba teraz skonfigurować. Jak pokazano na rys. 7 liczniki są taktowane częstotliwością 84 MHz z *APB1*.

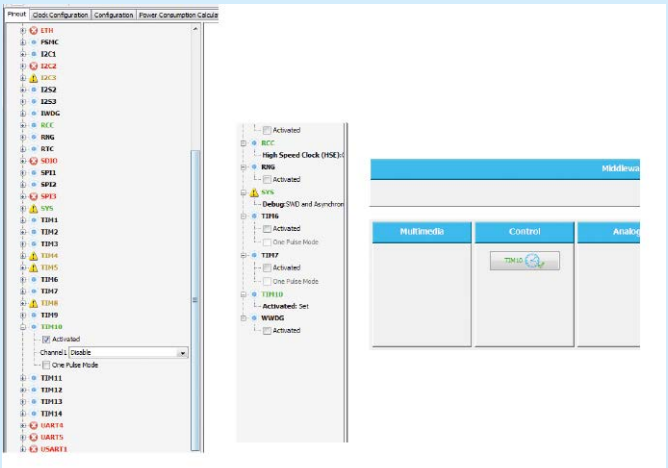


Rysunek 7. Zakładka Clock Configuration

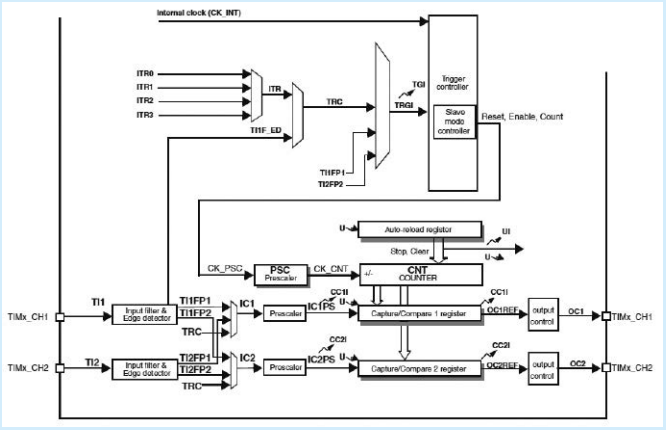
TIM10 jest licznikiem 16-bitowym z funkcją automatycznego przeładowania oraz z 16-bitowym preskalerem. Jego schemat blokowy pokazano na **rysunku 10**. W trybie zliczania UP (Upcounting) licznik zlicza od zera do wartości zapisanej do rejestru *Auto-reload register*. Kiedy zawartość licznika i rejestru *Auto-reload* są sobie równe, to rejestr licznika jest zerowany i odliczanie rozpoczyna się od nowa. W czasie zerowania licznika może być zgłaszane przerwanie.



Rysunek 8. Definicja linii diod modułu STM32F4 Discovery



Rysunek 9. Wybór licznika TIM10



Rysunek 10. Schemat blokowy licznika TIM10

Listing 4. Inicjalizacja licznika TIM10

```
void MX_TIM10_Init(void)
{
    htim10.Instance = TIM10;
    htim10.Init.Prescaler = 82;
    htim10.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim10.Init.Period = 1000;
    htim10.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    HAL_TIM_Base_Init(&htim10);
    __TIM10_CLK_ENABLE();
}
```

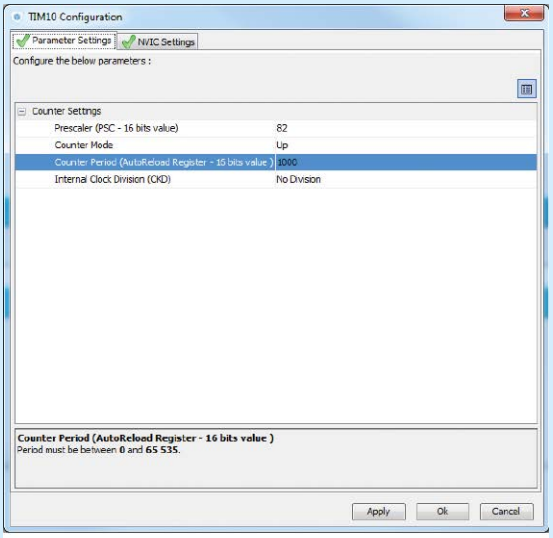
Listing 5. Funkcja obsługi przerwania od TIM10

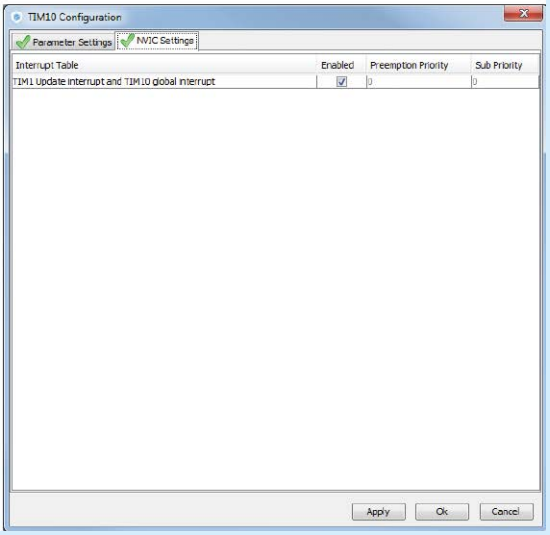
```
void TIM1_UP_TIM10_IRQHandler(void)
{
/* USER CODE BEGIN TIM1_UP_TIM10_IRQn 0 */
/* USER CODE END TIM1_UP_TIM10_IRQn 0 */
    HAL_TIM_IRQHandler(&htim10);
/* USER CODE BEGIN TIM1_UP_TIM10_IRQn 1 */
}
/* USER CODE END TIM1_UP_TIM10_IRQn 1 */
}
```

Listing 6. Procedura przerwania od przepełnienia TIM10

```
int blink=0;
void TIM1_UP_TIM10_IRQHandler(void)
{
/* USER CODE BEGIN TIM1_UP_TIM10_IRQn 0 */
/* USER CODE END TIM1_UP_TIM10_IRQn 0 */
    HAL_TIM_IRQHandler(&htim10);
/* USER CODE BEGIN TIM1_UP_TIM10_IRQn 1 */
    ++blink;
    if(blink==1000)
    {
        HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_15);
        blink=0;
    }
/* USER CODE END TIM1_UP_TIM10_IRQn 1 */
}
```

Za pomocą STM32CubeMX tak skonfigurujemy licznik *TIM10*, aby zgłaszał przerwanie co 1 ms, czyli 1000 razy na sekundę. W pierwszym kroku trzeba ustawić zliczanie w górę (UP) i preskaler na 82, aby na wejściu rejestru licznika była częstotliwość równa 1 MHz. Jeżeli teraz do *Auto-reload register* wpiszemy liczbę 1000, to licznik będzie się przepełniał z częstotliwością 1 MHz/1000 = 1 kHz, czyli co 1 ms. W zakładce *Configuration* klikamy na ikonkę *TIM10*, co powoduje wyświetlenie okna *TIM10 Settings*. W zakładce *Parameter Settings* ustawiamy: wartość preskalera równą 82, tryb zliczania *UP*, okres zliczania równy 1000. Aby licznik zgłaszał przerwanie należy z w zakładce *NVIC Settings* wybrać *TIM10 Update*





Rysunek 11. Konfiguracja TIM10

Interrupt and TOM10 global Interrupt Enabled. Oba ustawienia pokazano na **rysunku 11**. Na ich podstawie STM32CubeMX wygeneruje kod zawierający pokazaną na **listingu 10** funkcję inicjalizacji licznika TIM10 oraz funkcję obsługi przerwania – **listing 5**.

Należy pamiętać o tym, że kod generowany przez STM32CubeMX zawiera jedynie procedury konfigurujące moduły peryferyjne. Jeżeli chcemy, aby te układy zaczęły działać, to musimy je samodzielnie uruchomić wywołując odpowiednią funkcję. W tym przykładzie timer zostanie skonfigurowany, ale nie będzie zliczać impulsów wejściowych. Aby wszystko działało musimy wywołać funkcję *HAL_TIM_Base_Start_IT(&htim10);* uruchamiającą licznik i przerwania.

Na końcu pozostaje w obsłudze przerwania zaświecać i gasić wybrana diodę. Ponieważ przerwanie jest zgłaszane co 1 ms, to trzeba odliczyć 1000 przerwań, aby dioda pulsowała co 1 sekundę. Do tego celu zostanie użyta zmienna globalna *blink*, jak na **listingu 6**.

Tomasz Jabłoński, EP