Zadanie 2 polegać będzie na odczycie temperatury mikrokontrolera oraz kontroli położenia dwuosiowego manipulatora analogowego (DMA choć bezpieczniej jest zapomnieć w tym przypadku o skrócie i po prostu mówić o joystick'u). Niech składa się z kilku podzadań:

1. Odczyt temperatury MCU – tryb prosty (polling).
    a. Funkcja `HAL_ADC_Start`
    b. Funkcja `HAL_ADC_PollForConversion`
    c. `Funkcja HAL_ADC_GetValue`
2. Odczyt temperatury MCU – tryb przerwań
    a. Funkcja `HAL_ADC_Start_IT`
    b. Funkcja `HAL_ADC_ConvCpltCallback`
3. Odczyt linii analogowych MCU – tryb DMA.
    a. Funkcja `HAL_ADC_Start_DMA`
4. Ustawienie i zmiana parametrów pracy, w zależności od położenia manipulatora.
    a. Z1 – cykl linia 1: 200ms; linia 2: 300 ms; linia 3: 600ms przycisk linia 1: 150ms; linia 2: 100 ms; linia 3: 3s00ms
    b. Z2 - zliczanie binarne przycisk odwrócony cykl
    c. Z3 - wędrująca jedynka przycisk odwrócony cykl
    d. Z4 - wędrująca zero przycisk odwrócony cykl
    e. Z5 - Światła na skrzyżowaniu przycisk odwrócony cykl
    f. Centralne położenie manipulatora – zatrzymanie
        i. Z1 – trzy strefy prędkości sterowane osią X w prawo
        ii. Z2 – trzy strefy prędkości sterowane osią X w lewo
        iii. Z3 – trzy strefy prędkości sterowane osią Y w górę
        iv. Z4 – trzy strefy prędkości sterowane osią Y w dół
        v. Z5 – dwie strefy prędkości sterowane osią X prawo/lewo
        vi. Z6 – dwie strefy prędkości sterowane osią Y góra/dół

Przykładowe wywołania funkcji (niektóre nazwy są własne):

```
MX_ADC1_Init();
HAL_ADC_Start(&hadc1);/*tryb zwykły (polling) hadc1 struktura zawierająca konfigurację przetwornika */
HAL_ADC_Start_IT(&hadc1); /*przerwania*/
HAL_ADC_PollForConversion(&hadc1, 10); /*oczekiwanie na zakończenie przetwarzania;jeśli się zakończyło funkcja zwraca HAL_OK; drugi parametr to timeout*/
wynik = HAL_ADC_GetValue(&hadc1); /* przetwornik jest 12-bitowy więc wynik będzie przechowywany w zmiennej typu uint16_t */
HAL_ADC_Start_DMA(&hadc1, tablica, 2);/* drugi i trzeci parametr opisują miejsce docelowe dla danych, tablica dwuelementowa z elementami uint16_t*/
```

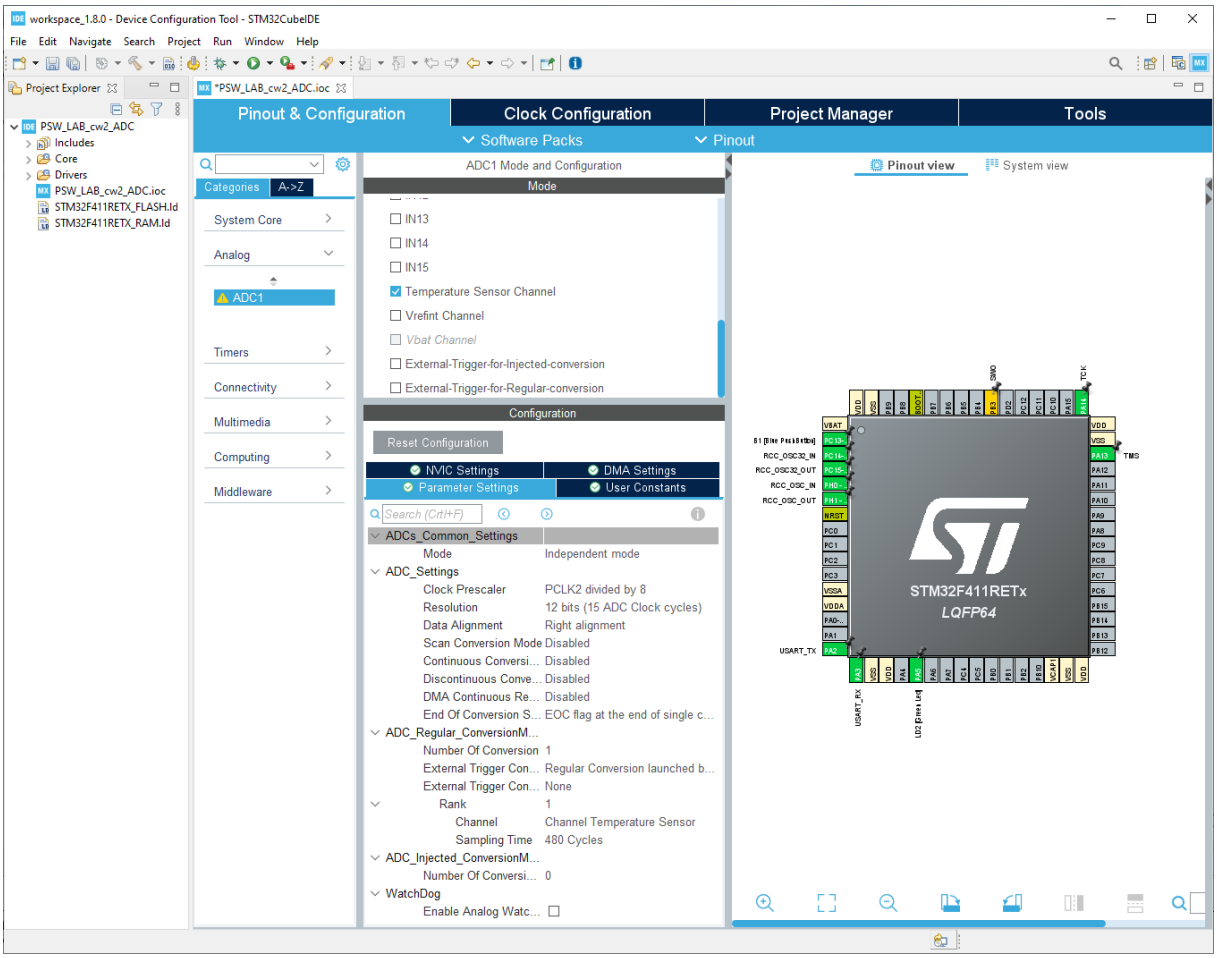
Szczegółowy opis funkcji znajduje się na końcu instrukcji.

Należy stworzyć nowy projekt (**File**->**New**->**STM32 Project**)

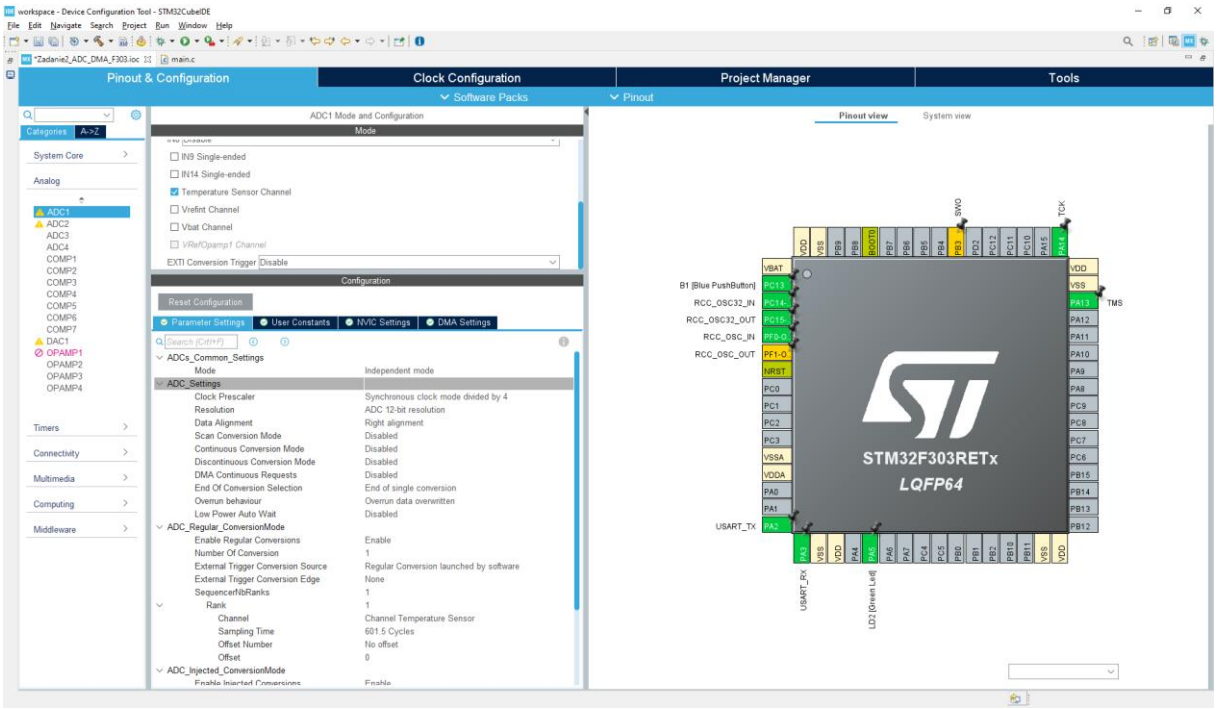
1.Pomiar temperatury MCU.

-tryb prosty (*polling*).

W oknie przedstawionym na rysunku 1 należy wybrać **Temperature Sensor Channel**.



Rysunek 1a. Okno konfiguracyjne projektu (STM32F411) z wybranym przetwornikiem **Analog**->**ADC1** i wybranym sensorem temperatury.



Rysunek 1b . Okno konfiguracyjne projektu (STM32F303) z wybranym przetwornikiem **Analog**->**ADC1** i wybranym sensorem temperatury.

Ustawienia przetwornika dostępne w oknie konfiguracyjnym z rysunku 1 mają następujące znaczenie:

ADC_Settings*:*

- **Clock Prescaler** – rejestr dzielnika zegara, którym wyzwalany jest przetwornik. Większa wartość oznacza mniejszą częstotliwość pracy (próbkowania) przetwornika.
- **Resolution** – liczba bitów wykorzystywana w operacji przetwarzania analogowo/cyfrowego. Większa długość oznacza większą dokładność przetwarzania ale też i dłuższy czas przetwarzania.
- **Data Alignment** – liczba bitów wartości wyjściowej przetwornika nie zawsze stanowi bajtową wielokrotność i dane mogą być wyrównywane (przesuwane) do lewej bądź do prawej strony (domyślnie do prawej).
- **Scan Conversion Mode** – opcja pozwala na zebranie danych (przeskanowanie) większej liczby/grupy kanałów w jednym cyklu/przebiegu zbierania/konwersji danych.
- **Continuous Conversion Mode** – automatyzuje proces przetwarzania A/C danych, w ten sposób, że po zakończeniu jednego cyklu automatycznie rozpoczyna się kolejny bez określenia chwili końcowej. Mechanizm ten wspiera (umożliwia) ciągłą akwizycję danych. Wykorzystanie tego mechanizmu wymaga przynajmniej w trybie przerwań, a najlepiej w trybie DMA, w którym można uzyskać wyższe wartości częstotliwości próbkowania. Założenie wykorzystania tego trybu w zwyczajnym „pollingu" jest praktycznie niemożliwe. Z jednej strony potencjalne prędkości przetwarzania były by bardzo małe ale co istotniejsze trudno w tym przypadku mówić o częstotliwości próbkowania.
- **Discontinuous Conversion mode** – automatyzuje proces przetwarzania A/C danych, w ten sposób, że po zakończeniu jednego cyklu przetwarzania (pomiaru) automatycznie rozpoczyna się kolejny z określeniem liczby powtórzeń - zbierany jest bufor danych o określonej długości.
- **DMA Continuous Requests** - w trybie DMA po każdym cyklu rejestracji przetwarzania danych generowane możliwe jest zainicjowanie procesu w DMA przesłania zmierzonej/zmierzonych wartości do zmiennej docelowej w pamięci.
- **End Of Conversion Selection** - pozwala na ustawienie flagi *EOC* (End OF Conversion) po każdej konwersji osobno czy też po przeprowadzeniu całej grupy konwersji.

ADC_Regular_ConvertionMode*:*

- **Number of Conversion** – określa liczę konwersji w pojedynczej sekwencji pomiarów.
- **External Trigger Conversion Edge** – umożliwia konfigurację zdarzenia zewnętrznego, które zainicjuje pomiar/przetwarzanie sygnału wejściowego.
- **Rank** – określa miejsce/kolejność danego pomiaru w sekwencji.
- **Channel** – określa wejście/numer kanału.

- **Sampling time** – okres próbkowania wyrażony w cyklach zegara. Im większa wartość tym dokładniejsze pomiary.

W bieżącej konfiguracji pomiar będzie dotyczył pojedynczego kanału, z tego względu **Scan Conversion Mode** pozostawiamy wyłączone. W tym przypadku każdorazowo pomiar będzie inicjowany i sprawdzane będzie zakończenie przetwarzania (polling), a więc **Continunous Conversion Mode** również należy wyłączyć.
Aby uzyskać większą dokładność pomiarów, parametry **Clock Prescaler** oraz **Sampling Time** należy nastawić na wartości maksymalne. W ramach eksperymentów warto określić ich wpływ na wyniki końcowe.

Po wygenerowaniu kodu w sekcji **private variables** znajdować się będzie następująca definicja:

```
ADC_HandleTypeDef hadc1;
```

Ta nowa struktura przeznaczona jest do przechowywania parametrów pracy przetwornika. Analogiczne podejście dotyczy konfiguracji również i innych peryferiów.

Struktura programu w trybie *polling* może mieć następującą postać:

```
ADC_inicjacja;
ADC_Start;
while(1)
{
     CzyKonwersjaZakończona?
          OdczytajWynik
     ADC_Start;
}
```

Funkcja `HAL_ADC_GetValue` zwróci wynik stanowiący zawartość rejestru wyjściowego przetwornika ADC. Aby otrzymać wartość temperatury należy dokonać przeliczeń zgodnych z dokumentacją.

Temperatura dla STM32F303 – Źródło: RM0316 Reference manual

Zakres pomiarowy: od -40 do 125 °C
Dokładność: ±2 °C

6. Calculate the actual temperature using the following formula:

   Temperature (in °C) = {(V_{25} – V_{TS}) / Avg_Slope} + 25

   Where:

   – V_{25} = V_{TS} value for 25° C
   – Avg_Slope = average slope of the temperature vs. V_{TS} curve (given in mV/°C or µV/°C)

   Refer to the datasheet electrical characteristics section for the actual values of V_{25} and Avg_Slope.

Charakterystyka Temperaturowa dla STM32F303 – Źródło STM32F303 datasheet.

Table 78. TS characteristics

Symbol	Parameter	Min	Typ	Max	Unit
T_L(1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope(1)	Average slope	4.0	4.3	4.6	mV/°C
V_{25}	Voltage at 25 °C	1.34	1.43	1.52	V
t_{START}(1)	Startup time	4	-	10	µs
T_{S_temp}(1)(2)	ADC sampling time when reading the temperature	2.2	-	-	µs

1. Guaranteed by design.
2. Shortest sampling time can be determined in the application by multiple iterations.

Temperatura dla STM32F411 – Źródło: RM0383 Reference manual

Zakres pomiarowy: od -40 do 125 °C
Dokładność: ±1.5 °C

8. Calculate the temperature using the following formula:

   Temperature (in °C) = {(V_{SENSE} – V_{25}) / Avg_Slope} + 25

   Where:

   – V_{25} = V_{SENSE} value for 25° C
   – Avg_Slope = average slope of the temperature vs. V_{SENSE} curve (given in mV/°C or µV/°C)

   Refer to the datasheet's electrical characteristics section for the actual values of V_{25} and Avg_Slope.

Charakterystyka Temperaturowa dla STM32F411 – Źródło STM32F411 datasheet.

Table 71. Temperature sensor characteristics

Symbol	Parameter	Min	Typ	Max	Unit
T_L(1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope(1)	Average slope	-	2.5	-	mV/°C
V_{25}(1)	Voltage at 25 °C	-	0.76	-	V
t_{START}(2)	Startup time	-	6	10	µs
T_{S_temp}(2)	ADC sampling time when reading the temperature (1 °C accuracy)	10	-	-	µs

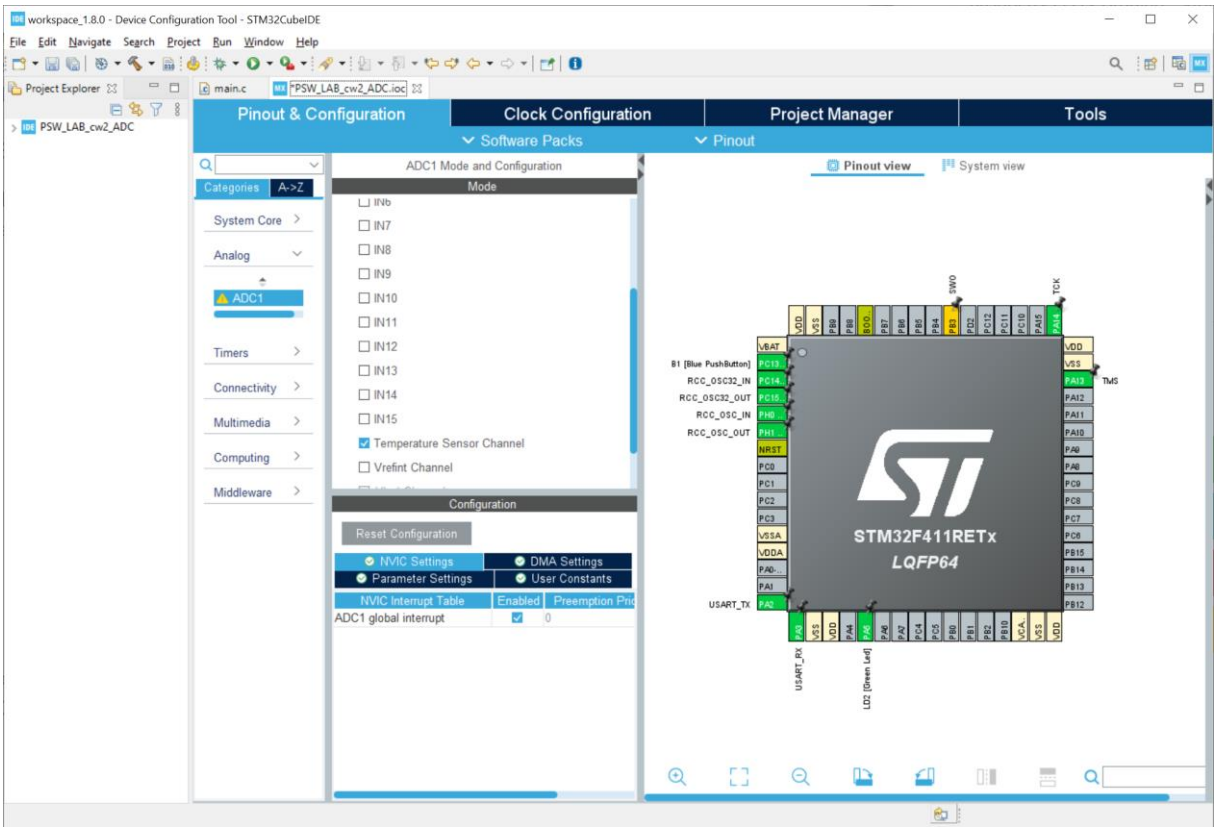
1. Guaranteed by characterization results.
2. Guaranteed by design.

Zatem należy zdefiniować (sekcja **private variables**) stałe opisujące średnie nachylenie - `avr_slope`, napięcie przy 25 °C – V_{25}. Typ stałych można zdefiniować jako `const float`, gdyż standardowo w środowisku nie ma definicji `float32_t` ale można dodać własny `typedef`. To niestety nie wszystko, każdorazowo odczytana z przetwornika wartość to stan wyjściowy jego rejestru. Należy ją przeskalować na jednostki napięcia – wolty. Skalowanie należy przeprowadzić pamiętając, że przetwornik ma 12 bitów (4096 stanów czyli 4095 przedziałów) a napięcie zasilania wynosi 3.3V. Zatem potrzebne będą dodatkowe zmienne (`float`) i stałe (`const float`). Po skompilowaniu programu należy go wgrać do mikrokontrolera. Bieżące wartości temperatury można odczytać za pomocą programu STM Studio (albo STM32 CubeMonitor).

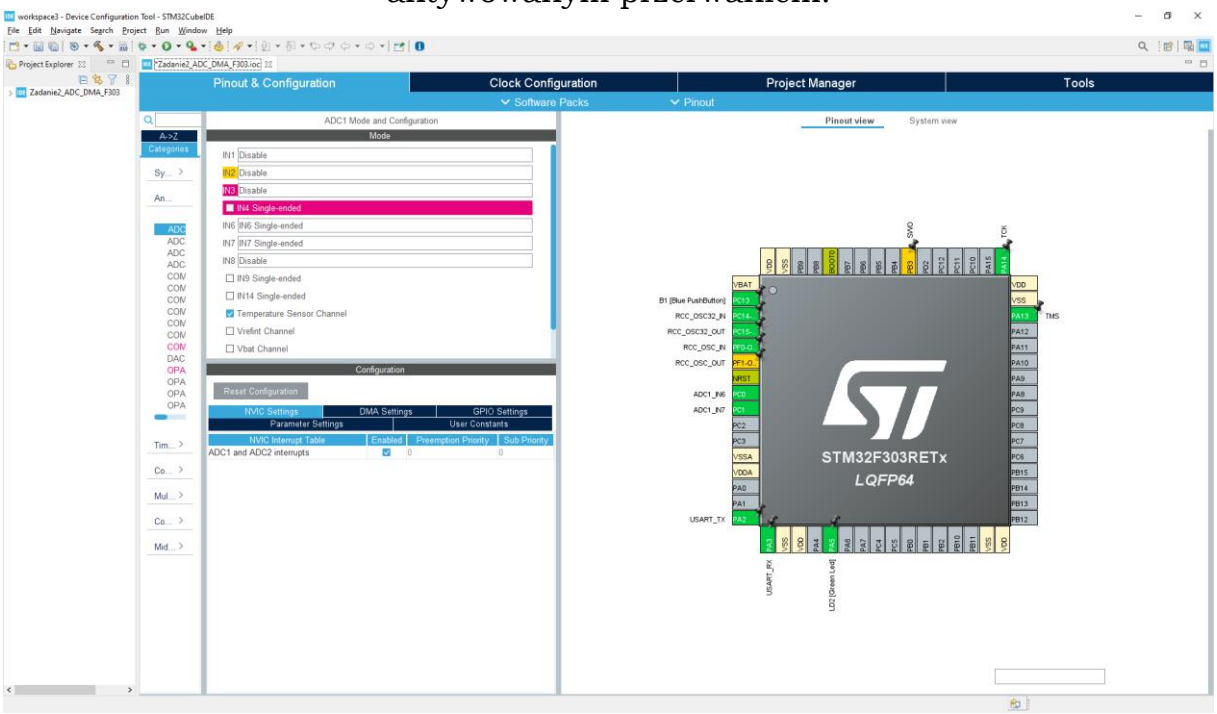
2.Pomiar temperatury MCU.

-tryb przerwań.

W oknie przedstawionym na rysunku 1 należy uaktywnić tryb **Continunous Conversion Mode** i w zakładce **NVIC settings** aktywować przerwanie pochodzące z przetwornika ADC1 – rysunek 2.



Rysunek 2a. Okno konfiguracyjne projektu (STM32F411) z wybranym przetwornikiem **Analog**->**ADC1**, wybranym sensorem temperatury i aktywowanym przerwaniem.



Rysunek 2b. Okno konfiguracyjne projektu (STM32F303) z wybranym przetwornikiem **Analog**->**ADC1**, wybranym sensorem temperatury i aktywowanym przerwaniem.

Nowa struktura programu będzie miała postać:

```
ADC_inicjacja;
ADC_Start_Przerwania;
while(1)
{
}
ObsługaPrzerwaniaADC;
```

Po wprowadzeniu zmian i wygenerowaniu kodu dodana jest funkcja obsługi przerwania. Jest ona umieszczona w pliku źródłowym zawierającym funkcje HAL dla ADC. Można ją odszukać wybierając z meny **Search**->**File...** i następnie w polu **Containing text**: wpisać **weak*callback**, pozostawiając w polu **File** *. Poszukiwana funkcja szablonowo zdefiniowana jest jako **weak**, tzn., że można (nawet zalecane jest) skopiować jej definicję do pliku źródłowego użytkownika (w zadaniu jest to main.c) już bez słowa kluczowego **weak** i tam wstawiać kod wykonawczy. Należy pamiętać o deklaracji i definicji nowej funkcji. W zadaniu nowa funkcja ma nazwę **`HAL_ADC_ConvCpltCallback`** i oryginalnie/szablonowo znajduje się w pliku stm32fxxx_hal_adc.c. Obsługa przerwania jest nową wygenerowaną funkcją. W niej należy umieścić odczyt stanu przetwornika i skalowanie. Nie trzeba już odpytywać o stan przetwornika i uruchamiać kolejnych konwersji. Przed skompilowaniem i uruchomieniem nowego programu należy dokonać jeszcze jednej zmiany. W miejsce funkcji **`HAL_ADC_Start`** należy użyć **`HAL_ADC_Start_IT`**, aktywującej mechanizm obsługi przerwań. Po skompilowaniu programu należy go wgrać do mikrokontrolera. Bieżące wartości temperatury można odczytać za pomocą programu STM Studio (albo STM32 CubeMonitor).

Jeśli funkcja przerwania „obsługuje" jednocześnie kilka przetworników (można to sprawdzić w **NVIC Settings** dla **ADC1**) aby obsłużyć wyłącznie wybrany (w tym przypadku **ADC1**) w funkcji **`HAL_ADC_ConvCpltCallback`** należy dodać instrukcję warunkową np.:

```
if((hadc->Instance) == ADC1) /* ADC1 jest przekazywany jako parametr do funkcji przerwania*/
```

3. Rejestracja wielokanałowa – Dwuosiowy Manipulator Analogowy (rysunek 3), jak sama nazwa wskazuje nie wypada obsługiwać go inaczej ja k tylko w trybie DMA.

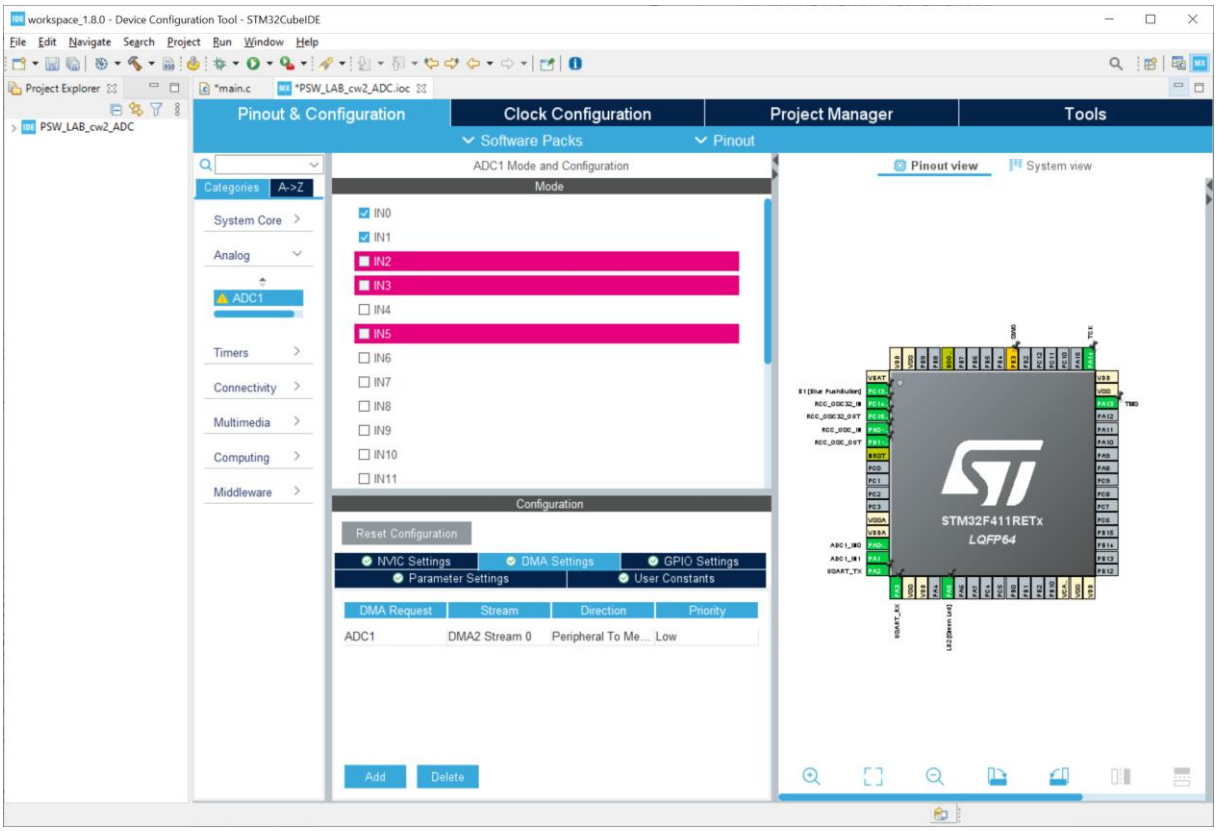
-tryb DMA.

Rysunek 3. Wykorzystany w ćwiczeniu dwuosiowy manipulator analogowy – joystick. Do wejścia 5V podłączyć należy napięcie zasilania MCU czyli **3.3V !**.

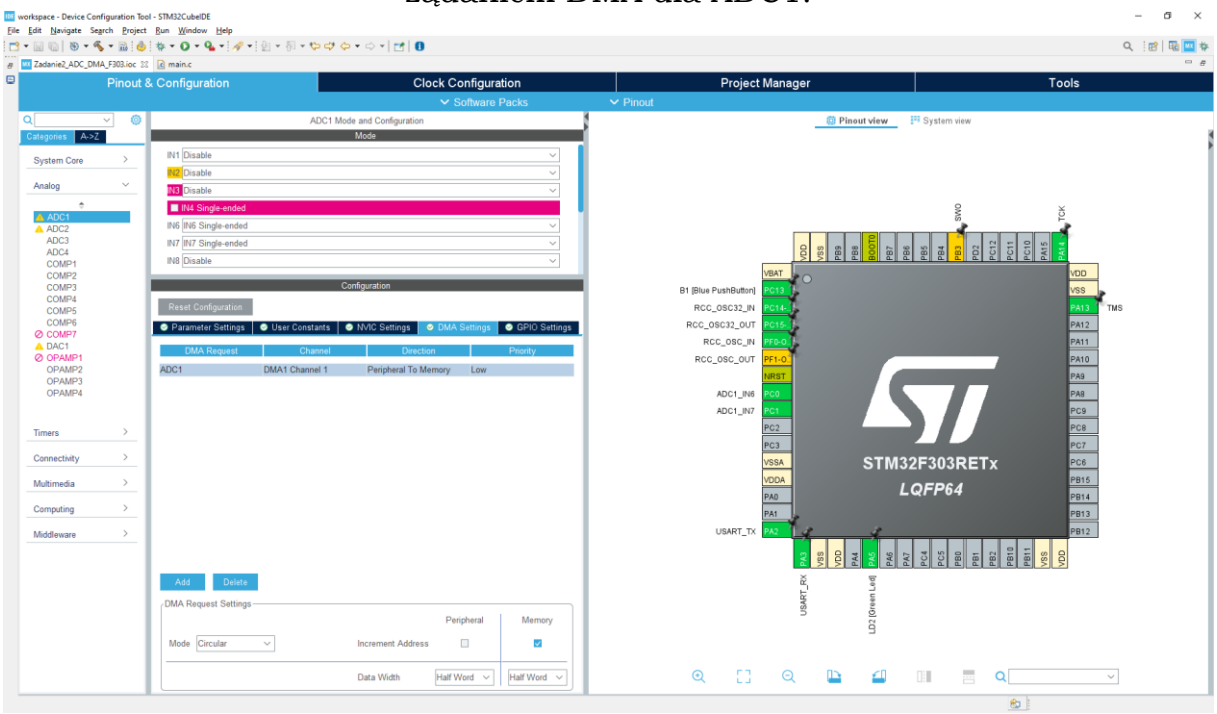
Joystick z rysunku 3 należy podłączyć następująco:

- GND – masa płytki kontrolera
- +5V – 3.3V zasilanie mikrokontrolera
- VRx – oś pozioma manipulatora -> PA0 (IN0) – uwaga na linie zajęte.
- VRy – oś pionowa manipulatora -> PA1 (IN1) – uwaga na linie zajętes
- SW- switch/przełącznik do wolnej (i skonfigurowanej) linii cyfrowej kontrolera

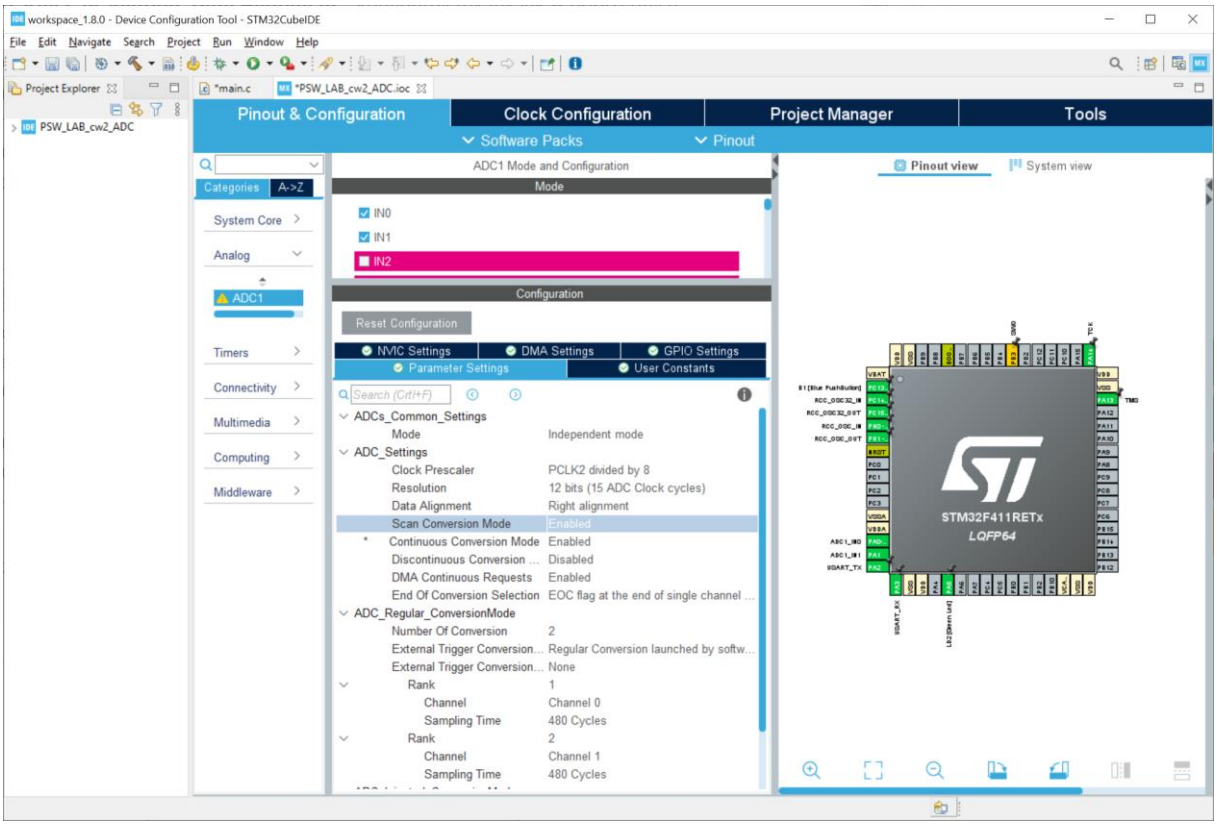
Na rysunku 4 i 5 przedstawiono konfigurację przetwornika dla DMA. W pierwszej kolejności należy uaktywnić żądania DMA dla ADC1 – rysunek 4. Następnie (rysunek 5) należy włączyć tryb **Scan Conversion Mode**, gdyż każdorazowo przetwarzanie będzie dotyczyć więcej niż jednego kanału – pojedynczy „skan" w tym przypadku dotyczy dwóch kanałów. Aktywować należy również tryb **Continuous Conversion Mode**, gdyż odczyt z kanałów ma odbywać się w sposób ciągły a ponieważ przesyłanie danych ma odbywać się w trybie DMA, włączyć należy również **DMA Continuous Request**. Każdorazowo rejestrowane będą dwa kanały i tak należy ustawić parametr **Number of Conversions**. Teraz należy ustalić o które kanały chodzi. W tym celu rozwinąć należy zakładkę **Rank 1** i wybrać **Channel 0**, podobnie w zakładce **Rank 2**, wybrać **Channel 1**. Sampling Time pozostaje jak poprzednio 480 cykli. Należy jeszcze sprawdzić w zakładce aktywację przerwań DMA i dezaktywację przerwań ADC – rysunek 6.



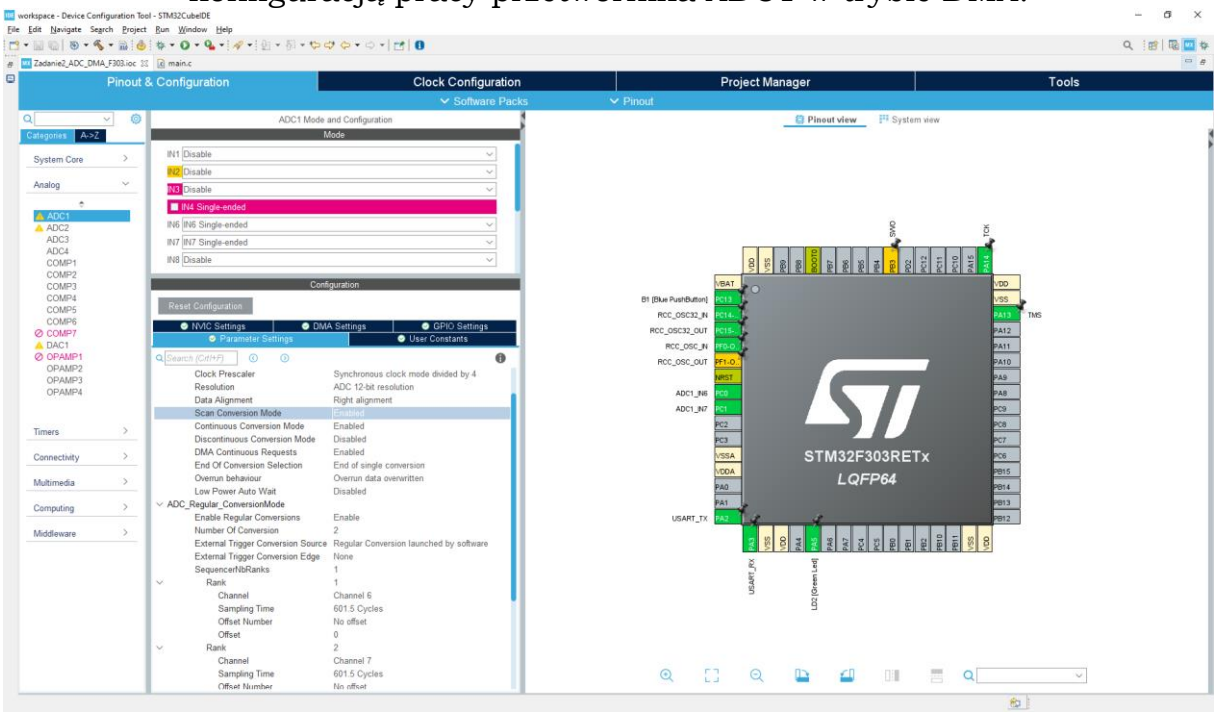
Rysunek 4a. Okno konfiguracyjne projektu (STM32F411) z wybranym przetwornikiem **Analog**->**ADC1,** wybranymi wejściami IN0 i IN1 oraz żądaniem DMA dla ADC1.



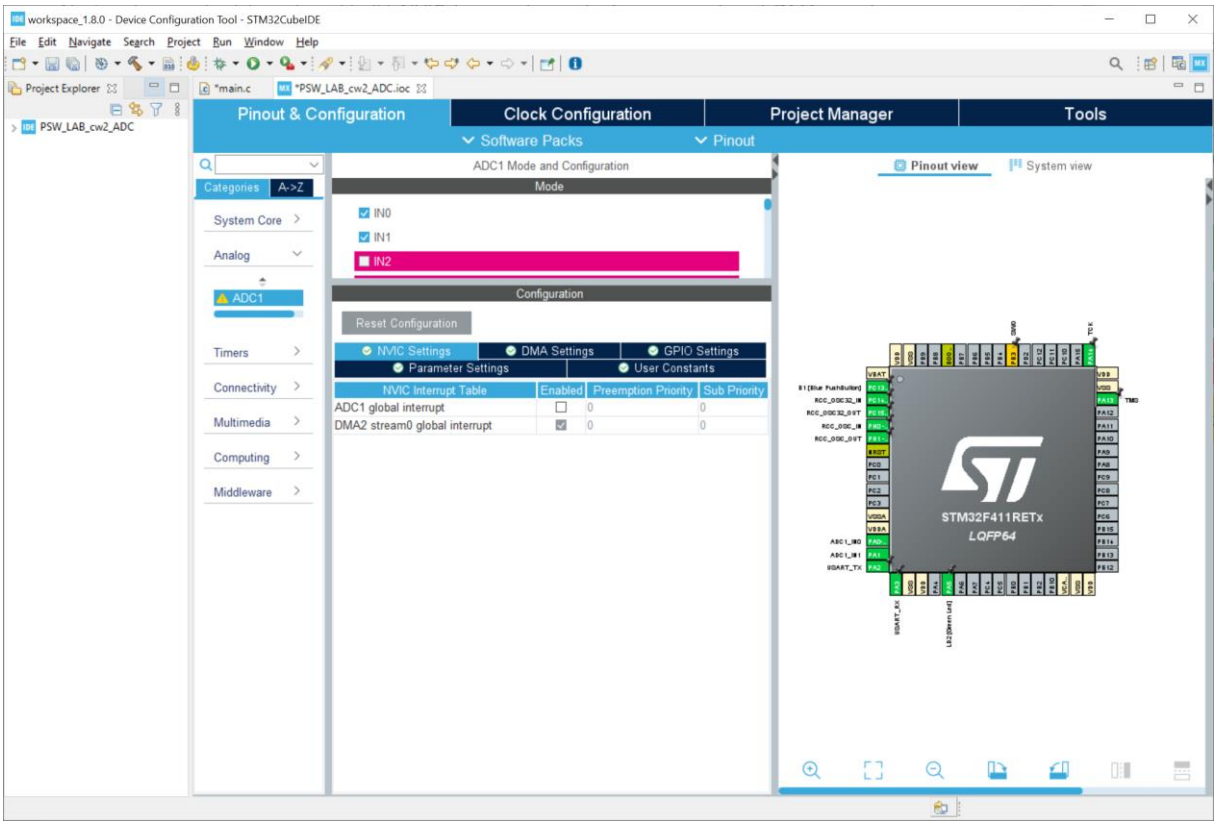
Rysunek 4b. Okno konfiguracyjne projektu (STM32F311) z wybranym przetwornikiem **Analog**->**ADC1,** wybranymi wejściami IN6 i IN7 oraz żądaniem DMA dla ADC1.



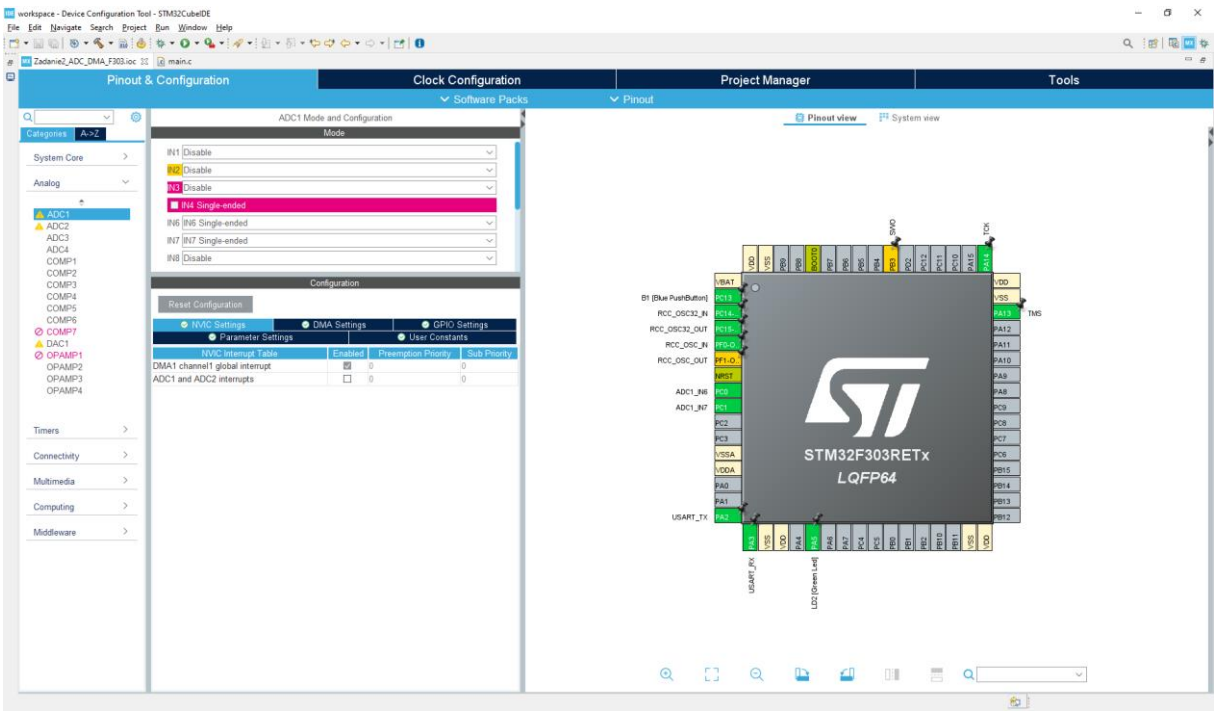
Rysunek 5a. Okno konfiguracyjne projektu (STM32F411) z wybranym przetwornikiem **Analog**->**ADC1,** wybranymi wejściami IN0 i IN1 oraz konfiguracją pracy przetwornika ADC1 w trybie DMA.



Rysunek 5b. Okno konfiguracyjne projektu (STM32F303) z wybranym przetwornikiem **Analog**->**ADC1,** wybranymi wejściami IN6 i IN7 oraz konfiguracją pracy przetwornika ADC1 w trybie DMA.



Rysunek 6a. Okno konfiguracyjne projektu (STM32F411) z wybranym przetwornikiem **Analog**->**ADC1,** wybranymi wejściami IN0 i IN1 oraz aktywnym przerwaniem dla DMA i nieaktywnym dla przetwornika ADC1.



Rysunek 6b. Okno konfiguracyjne projektu z wybranym przetwornikiem **Analog**->**ADC1,** wybranymi wejściami IN6 i IN7 oraz aktywnym przerwaniem dla DMA i nieaktywnym dla przetwornika ADC1.

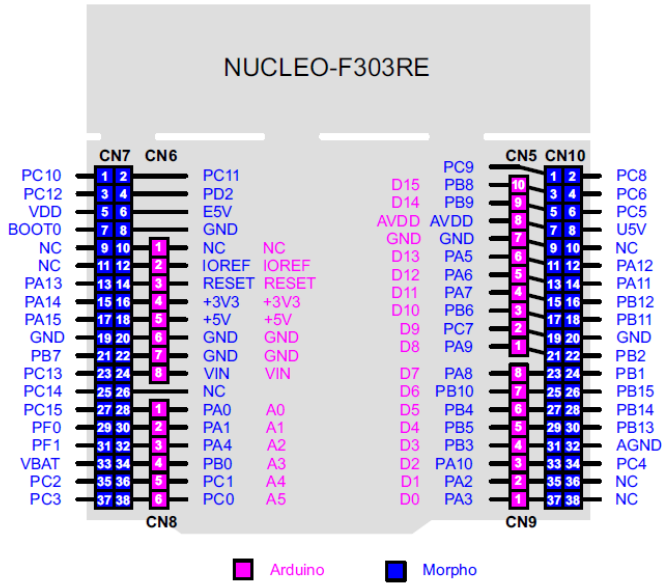
Tym razem struktura programu będzie następująca:

```
ADC_inicjacja_DMA;
ADC_Start_DMA;
while(1)
{
}
```

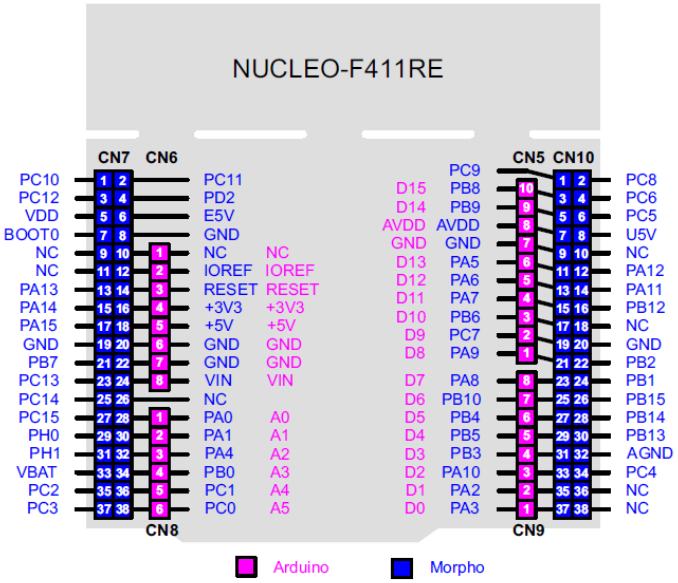
Funkcja uruchamiająca procedurę DMA, która każdorazowo kopiować będzie próbki (w licznie 2) do pamięci (zdefiniowanej tablicy) może wyglądać następująco:

```
HAL_ADC_Start_DMA(&hadc1, Joystick, 2);
```

gdzie `Joystick` jest tablicą dwuelementową typu `uint16_t` (w funkcji może okazać się przydatna konwersja tablicy do typu `(long unsigned int *)`). Definicję tablicy należy umieścić np.: w sekcji USER CODE PV. Po skompilowaniu programu należy go wgrać do mikrokontrolera. Bieżące wartości temperatury można odczytać za pomocą programu STM Studio (albo STM32 CubeMonitor).



Rysunek 7. Wyprowadzenia na płytce NUCLEO-F303RE. Źródło: STM32 Nucleo-64 boards (MB1136.pdf).



Rysunek 8. Wyprowadzenia na płytce NUCLEO-F411RE. Źródło: STM32 Nucleo-64 boards (MB1136.pdf).

Rysunki 7 i 8 przedstawiają wyprowadzenia dla płytek NUCLEO dla dwóch mikrokontrolerów STM23F411RE i STM23F4303RE. Dla wymienionych mikrokontrolerów przykładowe dostępne wejściowe linie analogowe to: PA0 (IN0), PA1 (IN1) dla Nucleo-STM32F411 albo PC0 (IN6), PC1 (IN7) dla Nucleo-STM32F303.

Przydatne skróty:
Ctrl+Spacja – parametry funkcji
Ctrl+/ - komentarz
Ctrl+s – zapis
Ctrl+Shift+f – autoformatowanie
Kod głównego pliku źródłowego wygenerowany automatycznie. Należy zwracać uwagę na komentarze i własny kod wstawiać jedynie w sekcjach USER pomiędzy BEGIN a END. Nigdy odwrotnie i nigdzie indziej !

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2022 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
```

```c
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

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

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
```

```c
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_ADC1_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

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

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

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 16;
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 4;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
```

```
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|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_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */
  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment
and number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure for the selected ADC regular channel its corresponding rank in the
sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
```

```
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
```

```
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

Dokumentacja funkcji użytych w programach

```
/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
```

```c
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */
  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment
and number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure for the selected ADC regular channel its corresponding rank in the
sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}


/**
  * @brief  Enables ADC and starts conversion of the regular channels.
  * @param  hadc pointer to a ADC_HandleTypeDef structure that contains
  *         the configuration information for the specified ADC.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc)
{
  __IO uint32_t counter = 0U;
  ADC_Common_TypeDef *tmpADC_Common;
```

```
  /* Check the parameters */
  assert_param(IS_FUNCTIONAL_STATE(hadc->Init.ContinuousConvMode));
  assert_param(IS_ADC_EXT_TRIG_EDGE(hadc->Init.ExternalTrigConvEdge));

  /* Process locked */
  __HAL_LOCK(hadc);

  /* Enable the ADC peripheral */
  /* Check if ADC peripheral is disabled in order to enable it and wait during
  Tstab time the ADC's stabilization */
  if((hadc->Instance->CR2 & ADC_CR2_ADON) != ADC_CR2_ADON)
  {
    /* Enable the Peripheral */
    __HAL_ADC_ENABLE(hadc);

    /* Delay for ADC stabilization time */
    /* Compute number of CPU cycles to wait for */
    counter = (ADC_STAB_DELAY_US * (SystemCoreClock / 1000000U));
    while(counter != 0U)
    {
      counter--;
    }
  }

  /* Start conversion if ADC is effectively enabled */
  if(HAL_IS_BIT_SET(hadc->Instance->CR2, ADC_CR2_ADON))
  {
    /* Set ADC state                                                        */
    /* - Clear state bitfield related to regular group conversion results     */
    /* - Set state bitfield related to regular group operation                */
    ADC_STATE_CLR_SET(hadc->State,
                      HAL_ADC_STATE_READY | HAL_ADC_STATE_REG_EOC |
HAL_ADC_STATE_REG_OVR,
                      HAL_ADC_STATE_REG_BUSY);

    /* If conversions on group regular are also triggering group injected,    */
    /* update ADC state.                                                      */
    if (READ_BIT(hadc->Instance->CR1, ADC_CR1_JAUTO) != RESET)
    {
      ADC_STATE_CLR_SET(hadc->State, HAL_ADC_STATE_INJ_EOC,
HAL_ADC_STATE_INJ_BUSY);
    }

    /* State machine update: Check if an injected conversion is ongoing */
    if (HAL_IS_BIT_SET(hadc->State, HAL_ADC_STATE_INJ_BUSY))
    {
      /* Reset ADC error code fields related to conversions on group regular */
      CLEAR_BIT(hadc->ErrorCode, (HAL_ADC_ERROR_OVR | HAL_ADC_ERROR_DMA));
    }
    else
    {
      /* Reset ADC all error code fields */
      ADC_CLEAR_ERRORCODE(hadc);
    }

    /* Process unlocked */
    /* Unlock before starting ADC conversions: in case of potential           */
    /* interruption, to let the process to ADC IRQ Handler.                   */
```

```
    __HAL_UNLOCK(hadc);

    /* Pointer to the common control register to which is belonging hadc    */
    /* (Depending on STM32F4 product, there may be up to 3 ADCs and 1 common */
    /* control register)                                                    */
    tmpADC_Common = ADC_COMMON_REGISTER(hadc);

    /* Clear regular group conversion flag and overrun flag */
    /* (To ensure of no unknown state from potential previous ADC operations) */
    __HAL_ADC_CLEAR_FLAG(hadc, ADC_FLAG_EOC | ADC_FLAG_OVR);

    /* Check if Multimode enabled */
    if(HAL_IS_BIT_CLR(tmpADC_Common->CCR, ADC_CCR_MULTI))
    {
#if defined(ADC2) && defined(ADC3)
      if((hadc->Instance == ADC1) || ((hadc->Instance == ADC2) && ((ADC->CCR &
ADC_CCR_MULTI_Msk) < ADC_CCR_MULTI_0)) \
                                    || ((hadc->Instance == ADC3) && ((ADC->CCR &
ADC_CCR_MULTI_Msk) < ADC_CCR_MULTI_4)))
      {
#endif /* ADC2 || ADC3 */
        /* if no external trigger present enable software conversion of regular
channels */
        if((hadc->Instance->CR2 & ADC_CR2_EXTEN) == RESET)
        {
          /* Enable the selected ADC software conversion for regular group */
          hadc->Instance->CR2 |= (uint32_t)ADC_CR2_SWSTART;
        }
#if defined(ADC2) && defined(ADC3)
      }
#endif /* ADC2 || ADC3 */
    }
    else
    {
      /* if instance of handle correspond to ADC1 and  no external trigger present
enable software conversion of regular channels */
      if((hadc->Instance == ADC1) && ((hadc->Instance->CR2 & ADC_CR2_EXTEN) ==
RESET))
      {
        /* Enable the selected ADC software conversion for regular group */
          hadc->Instance->CR2 |= (uint32_t)ADC_CR2_SWSTART;
      }
    }
  }
  else
  {
    /* Update ADC state machine to error */
    SET_BIT(hadc->State, HAL_ADC_STATE_ERROR_INTERNAL);

    /* Set ADC error code to ADC IP internal error */
    SET_BIT(hadc->ErrorCode, HAL_ADC_ERROR_INTERNAL);
  }

  /* Return function status */
  return HAL_OK;
}

/**
  * @brief  Poll for regular conversion complete
```

```
  * @note   ADC conversion flags EOS (end of sequence) and EOC (end of
  *         conversion) are cleared by this function.
  * @note   This function cannot be used in a particular setup: ADC configured
  *         in DMA mode and polling for end of each conversion (ADC init
  *         parameter "EOCSelection" set to ADC_EOC_SINGLE_CONV).
  *         In this case, DMA resets the flag EOC and polling cannot be
  *         performed on each conversion. Nevertheless, polling can still
  *         be performed on the complete sequence.
  * @param  hadc pointer to a ADC_HandleTypeDef structure that contains
  *         the configuration information for the specified ADC.
  * @param  Timeout Timeout value in millisecond.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc, uint32_t
Timeout)
{
  uint32_t tickstart = 0U;

  /* Verification that ADC configuration is compliant with polling for      */
  /* each conversion:                                                       */
  /* Particular case is ADC configured in DMA mode and ADC sequencer with   */
  /* several ranks and polling for end of each conversion.                  */
  /* For code simplicity sake, this particular case is generalized to       */
  /* ADC configured in DMA mode and polling for end of each conversion.     */
  if (HAL_IS_BIT_SET(hadc->Instance->CR2, ADC_CR2_EOCS) &&
      HAL_IS_BIT_SET(hadc->Instance->CR2, ADC_CR2_DMA)    )
  {
    /* Update ADC state machine to error */
    SET_BIT(hadc->State, HAL_ADC_STATE_ERROR_CONFIG);

    /* Process unlocked */
    __HAL_UNLOCK(hadc);

    return HAL_ERROR;
  }

  /* Get tick */
  tickstart = HAL_GetTick();

  /* Check End of conversion flag */
  while(!(__HAL_ADC_GET_FLAG(hadc, ADC_FLAG_EOC)))
  {
    /* Check if timeout is disabled (set to infinite wait) */
    if(Timeout != HAL_MAX_DELAY)
    {
      if((Timeout == 0U) || ((HAL_GetTick() - tickstart ) > Timeout))
      {
        /* New check to avoid false timeout detection in case of preemption */
        if(!(__HAL_ADC_GET_FLAG(hadc, ADC_FLAG_EOC)))
        {
          /* Update ADC state machine to timeout */
          SET_BIT(hadc->State, HAL_ADC_STATE_TIMEOUT);

          /* Process unlocked */
          __HAL_UNLOCK(hadc);

          return HAL_TIMEOUT;
        }
      }
```

```
    }
  }

  /* Clear regular group conversion flag */
  __HAL_ADC_CLEAR_FLAG(hadc, ADC_FLAG_STRT | ADC_FLAG_EOC);

  /* Update ADC state machine */
  SET_BIT(hadc->State, HAL_ADC_STATE_REG_EOC);

  /* Determine whether any further conversion upcoming on group regular       */
  /* by external trigger, continuous mode or scan sequence on going.          */
  /* Note: On STM32F4, there is no independent flag of end of sequence.       */
  /*       The test of scan sequence on going is done either with scan        */
  /*       sequence disabled or with end of conversion flag set to            */
  /*       of end of sequence.                                                */
  if(ADC_IS_SOFTWARE_START_REGULAR(hadc)                   &&
     (hadc->Init.ContinuousConvMode == DISABLE)            &&
     (HAL_IS_BIT_CLR(hadc->Instance->SQR1, ADC_SQR1_L) ||
      HAL_IS_BIT_CLR(hadc->Instance->CR2, ADC_CR2_EOCS)  )   )
  {
    /* Set ADC state */
    CLEAR_BIT(hadc->State, HAL_ADC_STATE_REG_BUSY);

    if (HAL_IS_BIT_CLR(hadc->State, HAL_ADC_STATE_INJ_BUSY))
    {
      SET_BIT(hadc->State, HAL_ADC_STATE_READY);
    }
  }

  /* Return ADC state */
  return HAL_OK;
}

/**
  * @brief  Gets the converted value from data register of regular channel.
  * @param  hadc pointer to a ADC_HandleTypeDef structure that contains
  *         the configuration information for the specified ADC.
  * @retval Converted value
  */
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc)
{
  /* Return the selected ADC converted value */
  return hadc->Instance->DR;
}

/**
  * @brief  Enables ADC DMA request after last transfer (Single-ADC mode) and
enables ADC peripheral
  * @param  hadc pointer to a ADC_HandleTypeDef structure that contains
  *         the configuration information for the specified ADC.
  * @param  pData The destination Buffer address.
  * @param  Length The length of data to be transferred from ADC peripheral to
memory.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc, uint32_t* pData,
uint32_t Length)
{
  __IO uint32_t counter = 0U;
```

```
  ADC_Common_TypeDef *tmpADC_Common;

  /* Check the parameters */
  assert_param(IS_FUNCTIONAL_STATE(hadc->Init.ContinuousConvMode));
  assert_param(IS_ADC_EXT_TRIG_EDGE(hadc->Init.ExternalTrigConvEdge));

  /* Process locked */
  __HAL_LOCK(hadc);

  /* Enable the ADC peripheral */
  /* Check if ADC peripheral is disabled in order to enable it and wait during
  Tstab time the ADC's stabilization */
  if((hadc->Instance->CR2 & ADC_CR2_ADON) != ADC_CR2_ADON)
  {
    /* Enable the Peripheral */
    __HAL_ADC_ENABLE(hadc);

    /* Delay for ADC stabilization time */
    /* Compute number of CPU cycles to wait for */
    counter = (ADC_STAB_DELAY_US * (SystemCoreClock / 1000000U));
    while(counter != 0U)
    {
      counter--;
    }
  }

  /* Check ADC DMA Mode                                                    */
  /* - disable the DMA Mode if it is already enabled                       */
  if((hadc->Instance->CR2 & ADC_CR2_DMA) == ADC_CR2_DMA)
  {
    CLEAR_BIT(hadc->Instance->CR2, ADC_CR2_DMA);
  }

  /* Start conversion if ADC is effectively enabled */
  if(HAL_IS_BIT_SET(hadc->Instance->CR2, ADC_CR2_ADON))
  {
    /* Set ADC state                                                       */
    /* - Clear state bitfield related to regular group conversion results  */
    /* - Set state bitfield related to regular group operation             */
    ADC_STATE_CLR_SET(hadc->State,
                      HAL_ADC_STATE_READY | HAL_ADC_STATE_REG_EOC |
HAL_ADC_STATE_REG_OVR,
                      HAL_ADC_STATE_REG_BUSY);

    /* If conversions on group regular are also triggering group injected, */
    /* update ADC state.                                                   */
    if (READ_BIT(hadc->Instance->CR1, ADC_CR1_JAUTO) != RESET)
    {
      ADC_STATE_CLR_SET(hadc->State, HAL_ADC_STATE_INJ_EOC,
HAL_ADC_STATE_INJ_BUSY);
    }

    /* State machine update: Check if an injected conversion is ongoing */
    if (HAL_IS_BIT_SET(hadc->State, HAL_ADC_STATE_INJ_BUSY))
    {
      /* Reset ADC error code fields related to conversions on group regular */
      CLEAR_BIT(hadc->ErrorCode, (HAL_ADC_ERROR_OVR | HAL_ADC_ERROR_DMA));
    }
    else
```

```
    {
      /* Reset ADC all error code fields */
      ADC_CLEAR_ERRORCODE(hadc);
    }

    /* Process unlocked */
    /* Unlock before starting ADC conversions: in case of potential           */
    /* interruption, to let the process to ADC IRQ Handler.                   */
    __HAL_UNLOCK(hadc);

    /* Pointer to the common control register to which is belonging hadc    */
    /* (Depending on STM32F4 product, there may be up to 3 ADCs and 1 common */
    /* control register)                                                      */
    tmpADC_Common = ADC_COMMON_REGISTER(hadc);

    /* Set the DMA transfer complete callback */
    hadc->DMA_Handle->XferCpltCallback = ADC_DMAConvCplt;

    /* Set the DMA half transfer complete callback */
    hadc->DMA_Handle->XferHalfCpltCallback = ADC_DMAHalfConvCplt;

    /* Set the DMA error callback */
    hadc->DMA_Handle->XferErrorCallback = ADC_DMAError;


    /* Manage ADC and DMA start: ADC overrun interruption, DMA start, ADC     */
    /* start (in case of SW start):                                           */

    /* Clear regular group conversion flag and overrun flag */
    /* (To ensure of no unknown state from potential previous ADC operations) */
    __HAL_ADC_CLEAR_FLAG(hadc, ADC_FLAG_EOC | ADC_FLAG_OVR);

    /* Enable ADC overrun interrupt */
    __HAL_ADC_ENABLE_IT(hadc, ADC_IT_OVR);

    /* Enable ADC DMA mode */
    hadc->Instance->CR2 |= ADC_CR2_DMA;

    /* Start the DMA channel */
    HAL_DMA_Start_IT(hadc->DMA_Handle, (uint32_t)&hadc->Instance->DR,
(uint32_t)pData, Length);

    /* Check if Multimode enabled */
    if(HAL_IS_BIT_CLR(tmpADC_Common->CCR, ADC_CCR_MULTI))
    {
#if defined(ADC2) && defined(ADC3)
      if((hadc->Instance == ADC1) || ((hadc->Instance == ADC2) && ((ADC->CCR &
ADC_CCR_MULTI_Msk) < ADC_CCR_MULTI_0)) \
                                    || ((hadc->Instance == ADC3) && ((ADC->CCR &
ADC_CCR_MULTI_Msk) < ADC_CCR_MULTI_4)))
      {
#endif /* ADC2 || ADC3 */
        /* if no external trigger present enable software conversion of regular
channels */
        if((hadc->Instance->CR2 & ADC_CR2_EXTEN) == RESET)
        {
          /* Enable the selected ADC software conversion for regular group */
          hadc->Instance->CR2 |= (uint32_t)ADC_CR2_SWSTART;
        }
```

```
#if defined(ADC2) && defined(ADC3)
      }
#endif /* ADC2 || ADC3 */
    }
    else
    {
      /* if instance of handle correspond to ADC1 and  no external trigger present
enable software conversion of regular channels */
      if((hadc->Instance == ADC1) && ((hadc->Instance->CR2 & ADC_CR2_EXTEN) ==
RESET))
      {
        /* Enable the selected ADC software conversion for regular group */
          hadc->Instance->CR2 |= (uint32_t)ADC_CR2_SWSTART;
      }
    }
  }
  else
  {
    /* Update ADC state machine to error */
    SET_BIT(hadc->State, HAL_ADC_STATE_ERROR_INTERNAL);

    /* Set ADC error code to ADC IP internal error */
    SET_BIT(hadc->ErrorCode, HAL_ADC_ERROR_INTERNAL);
  }

  /* Return function status */
  return HAL_OK;
}
```