Analog to Digital Converter

There are many types of analog devices connected to microcontrollers. These analog device generate a variable voltage in a meaningful range. That voltage level is read and converted to a number through an Analog to Digital Converter.

#arm #stm32 #adc #joystick

Last update: 2021-06-29 18:23:57

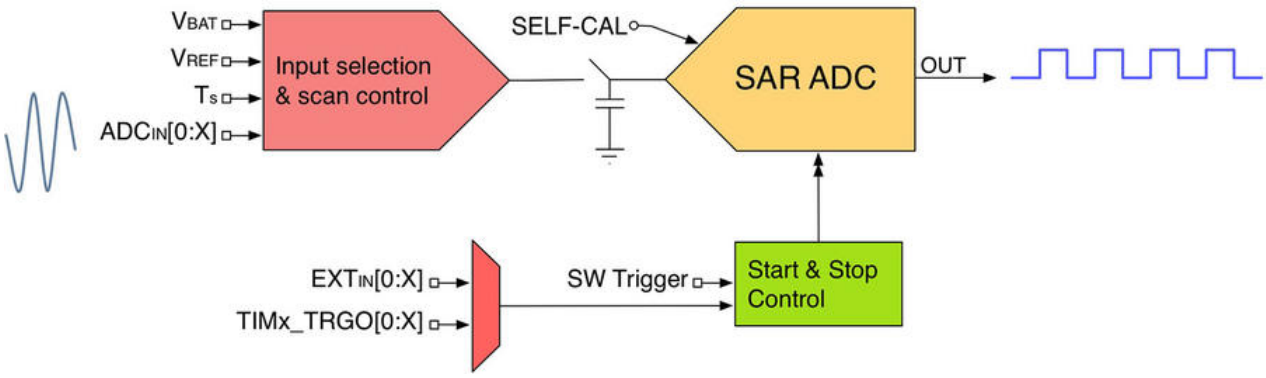
Table of Content

1. Analog to Digital Converter

An Analog-to-Digital Converter (ADC) is able to acquire input voltage level and convert it to a number. The input voltage is compared against a well known and fixed voltage, also known as reference voltage. This reference voltage can be either derived from the V_{DDA} domain or, in MCUs with high pin count, supplied by an external and fixed reference voltage generator V_{REF+}. The majority of STM32 MCUs provide a 12-bit ADC. Some of them from the STM32F3 portfolio even a 16-bit ADC.

In almost all STM32 microcontrollers, the ADC is implemented as a *Successive Approximation Register ADC*. MCUs usually have more than ten channels, allowing to measure signals from external sources. Moreover, some internal channels are also available: a channel for internal temperature sensor (V_{SENSE}), one for internal reference voltage (V_{REFINT}), one for monitoring external V_{BAT} power supply and a channel for monitoring LCD voltage in those MCUs providing a native monochrome passive LCD controller (for example, the STM32L053). ADCs implemented in STM32F3 and in majority of STM32L4 MCUs are also capable of converting fully differential inputs.

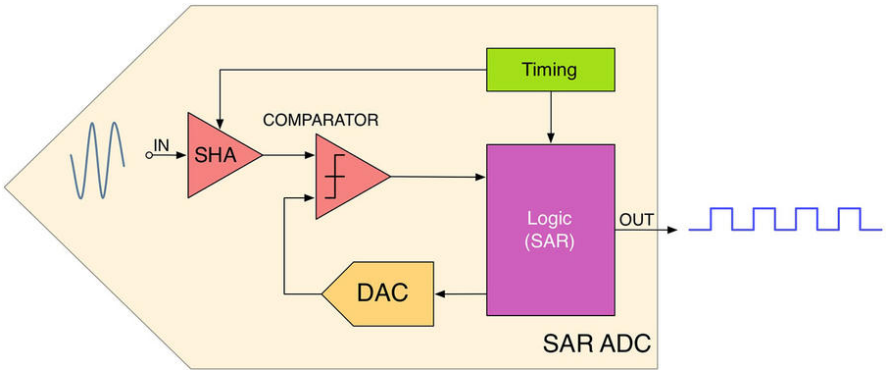
A/D conversion of the various channels can be performed in single, continuous, scan, or discontinuous mode. The result of the ADC is stored in a left- or right-aligned 16-bit data register. Moreover, the ADC also implements the analog watchdog feature, which allows the application to detect if the input voltage goes outside the user-defined higher or lower thresholds: if this happens, a dedicated IRQ fires.



A simplified structure of an ADC

An *input selection and scan control unit* performs the selection of the input source to the ADC. Depending on the conversion mode (single, scan or continuous mode), this unit automatically switches among the input channels, so that every one can be sampled periodically. The output from this unit feeds the ADC.

The *start and stop control unit* is to control the A/D conversion process, and it can be triggered by software or by a variable number of input sources. Moreover, it is internally connected to the `TRGO` line of some timers so that time-driven conversions can be automatically performed in DMA mode.



The internal structure of a SAR ADC

The input signal goes through the *Sample-and-Hold (SHA)* unit, a feature available in all ADCs, to keep the input signal constant during the conversion cycle. To keep the voltage level of the input constant, the SHA is implemented with a network of capacitors: this ensure that source signal is kept at a certain level during the A/D conversion, which is a procedure that requires a given amount of time, depending on the conversion frequency chosen.

The output from the SHA module feeds a comparator that compares it with another signal generated by an internal DAC. The result of comparison is sent to the logic unit, which computes the numerical representation of the input signal according a well-characterized algorithm. This algorithm is what distinguishes SAR ADC from other A/D converters.

The *Successive Approximation* algorithm computes the voltage of the input signal by comparing it with the one generated by the internal DAC, which is a fraction of the V_{REF} voltage: *if the input signal is higher than this internal reference voltage, then this is further increased until the input signal is lower*. The final result will correspond to a number ranging from zero to the maximum 12-bit unsigned integer. The SAR algorithm essentially performs a search in a binary tree. The great advantage of this algorithm is that the conversion is performed in N-cycles, where N corresponds to the ADC resolution. So a 12-bit ADC requires twelve cycles to perform a conversion.

2. Calibration

The ADC has a calibration feature. During the procedure, the ADC calculates a calibration factor which is internally applied to the ADC until the next ADC power-off. The application must not use the ADC during calibration and must wait until it is complete.

Calibration should be performed before starting A/D conversion. It removes the offset error which may vary from chip to chip due to process variation. The calibration is initiated by software by setting bit `ADCAL=1`. Calibration can only be initiated when the ADC is disabled (when `ADEN=0`). `ADCAL` bit stays at 1 during all the calibration sequence. It is then cleared by hardware as soon the calibration completes. After this, the calibration factor can be read from the `ADC_DR` register (from bits 6 to 0).

3. Conversion clock

The ADC clock mainly is derived from the APB clock, however it may have a dual clock-domain architecture on some MCU line:

- The ADC clock can be a specific clock source, named “ADC asynchronous clock” which is independent and asynchronous with the APB clock. It has the advantage of reaching the maximum ADC clock frequency whatever the APB clock scheme selected.
- The ADC clock can be derived from the APB clock of the ADC bus interface, divided by a programmable factor (2/4/n) and it must not exceed 14 MHz. This has the advantage of bypassing the clock domain re-synchronizations. This can be useful when the ADC is triggered by a timer and if the application requires that the ADC is precisely triggered without any uncertainty.

4. Conversion Modes

ADCs implemented in STM32 MCUs provide several conversion modes useful to deal with different application scenarios.

4.1. Single-Channel, Single Conversion Mode

In this mode, the ADC performs the single conversion (single sample) of a single channel, then stops when conversion is finished.

4.2. Multiple-channel, Single Conversion Mode

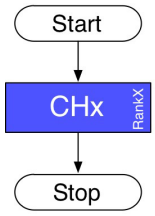
It’s also know as Scan Single Conversion Mode. A sequence can have up-to 16 channels, in a custom order. ADC module do not need to stop for CPU to configure the next channel, therefore, it saves CPU load can run in background. DMA is used to store converted data.

4.3. Single-Channel, Continuous Conversion Mode

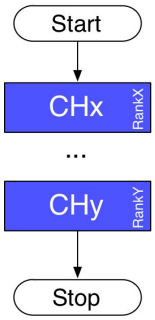
This mode converts a single channel continuously and indefinitely in regular channel conversion. The continuous mode feature allows the ADC to work in the background. The ADC converts the channels continuously without any intervention from the CPU. Additionally, the DMA can be used in circular mode, thus reducing the CPU load.

4.4. Multiple-channel, Continuous Conversion Mode

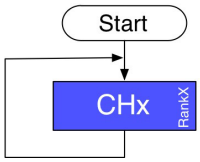
This mode is also called multichannel continuous mode and it can be used to convert some channels successively with the ADC in independent mode. Scan conversions are carried out in DMA mode.



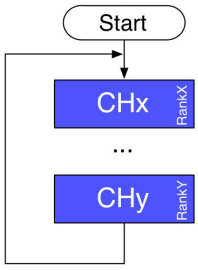
1 Channel, Single Mode



N-Channel, Single Mode



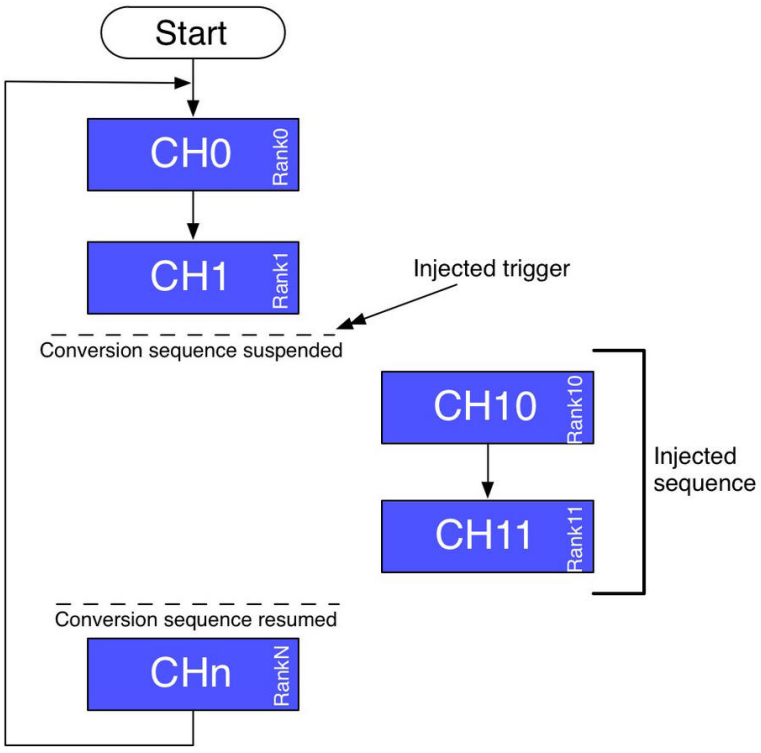
1-Channel, Continuous Mode



N-Channel, Continuous Mode

4.5. Injected Conversion Mode

This mode is intended for use when conversion is triggered by an external event or by software. The injected group has priority over the regular channel group. It interrupts the conversion of the current channel in the regular channel group.



Injected conversion mode

For example, this mode can be used to synchronize the conversion of channels to an event. It is interesting in motor control applications where transistor switching generates noise that impacts ADC measurements and results in wrong conversions. Using a timer, the injected conversion mode can thus be implemented to delay the ADC measurements to after the transistor switching.

5. Channel Selection

Depending on the STM32 family and package used, ADCs in STM32 MCUs can convert signals from a variable number of channels.

5.1. F0 and L0 families

In F0 and L0 families the allocation of channel is fixed: the first one is always IN0, the second IN1 and so on. User can decide only if a channel is enabled or not. This means that in scan mode the first sampled channel will be always IN0, the second IN1 and so on. The sampling speed is applied to all channel in F0 and L0 families.

To read 1-channel, it is needed to disable the previous configured channel. If more than 1 channel is enabled, ADC will work in N-channel modem, causing reading ADC value always returns the value of the first channel. This will be explained more in the Lab 1 which use an STM32**F0**51R8 MCU.

5.2. Other Fx and Lx families

Other STM32 MCUs, instead, offer the notion of *group*. A group consists of a sequence of conversions that can be done on any channel and in any order. The reordering of channels is performed by assigning to them an index (called *rank*) ranging from 1 to 16. Those MCUs offering this possibility also allow to select the sampling speed of each channel individually, differently from F0/L0 MCUs where the configuration is ADC-wide.

There exist two groups for each ADC:

- A regular group, made of up to 16 channels, which corresponds to the sequence of sampled channels during a scan conversion.
- An injected group, made of up to 4 channels, which corresponds to the sequence of injected channel if an injected conversion is performed.

This will be explained more in the Lab 2 which use an STM32**F4**11CE MCU.

6. Lab 1: Joystick in Polling Mode

The PS2 joystick module shown in the figure down below is basically a couple of potentiometers (variable resistors) that moves in 2 axises. This creates a voltage difference that can be read using any ADC peripheral on two analog pins: `VRX` and `VRY`.

A simple joystick

Requirements

- Read Joystick analog pins `VRX` and `VRY` stimulatingly
- Transfer these value in raw number on UART1

Target board Any board which has STM32 MCUs. This tutorial will be using the STM32F0 Discovery board, which features an STM32F051R8 Cortex-M0 MCU.

STM32F051R8	Mode	External peripheral
PA9	Alternate Function	UART1 TX
PA10	Alternate Function	UART1 RX
PC8	GPIO Output	Blue LED
PC9	GPIO Output	Green LED
PA1	Analog Input 1	VRX
PA2	Analog Input 2	VRY

6.1. Start a new project

Open STM32CubeIDE and create a new STM32 with STM32F051R8 MCU by selecting the target board or just the target MCU.

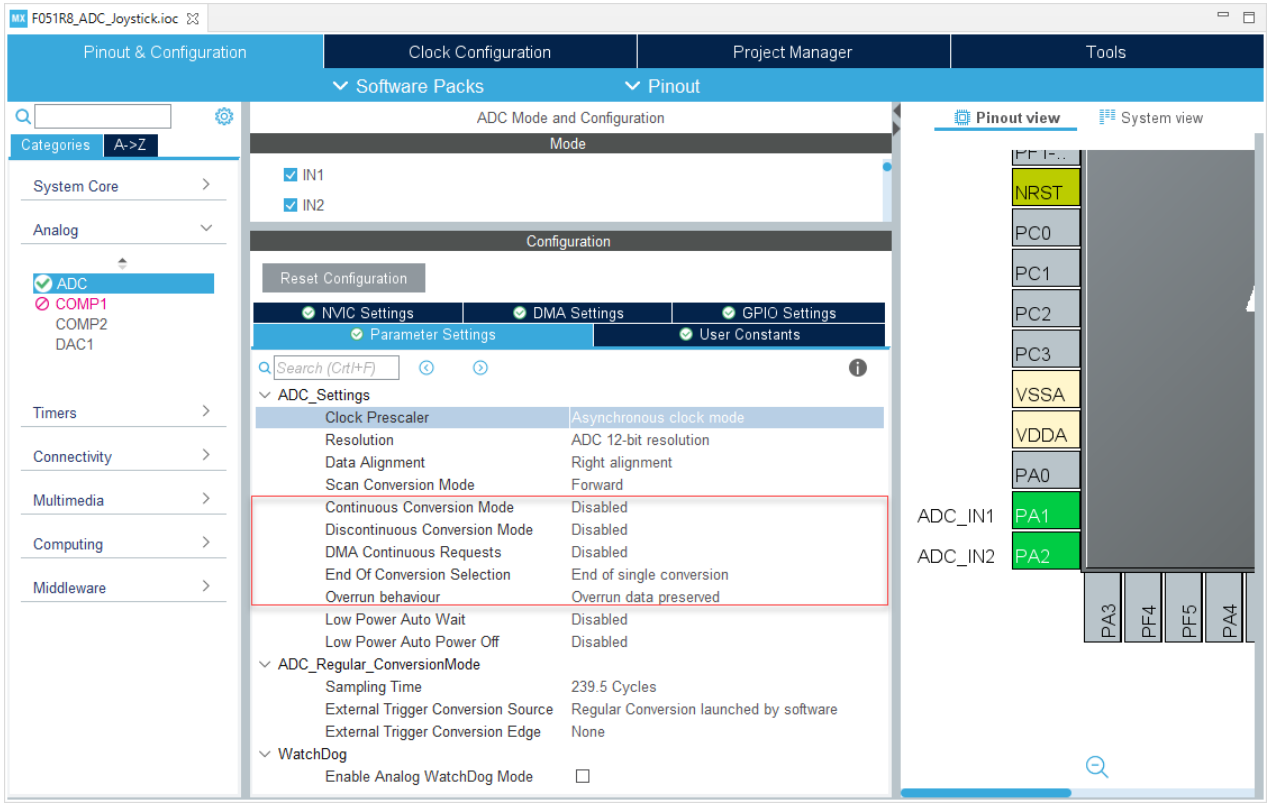
Make sure to configure below settings:

- Set the **HLCK** to *48 MHz*
- Set the **Debug** mode to *Debug Serial Wire*
- Enable **USART1** with settings: *115200 bps, 8 bits, no Parity, and 1-bit stop*
- Set the **PC8** and **PC9** pins to *GPIO output* mode

6.2. Enable ADC Peripheral

In the **ADC** module under the **Analog** category, select below settings:

- Select Input Channel **IN1**, and **IN2**
- Clock Pre-scaler: If there is an dedicated HSI ADC clock source, *Asynchronous clock* can be selected. Otherwise, if the clock source is derived from APB clock, there is synchronous mode available there.
- Continuous/Discontinuous Conversion Mode are disabled by default, as ADC is not in scan mode
- DMA Continuous Request is disabled by default
- End of Conversion flag is selected to be set at *End of single conversion*
- Overrun behavior is set to *Overrun Data Preserved*
- Sampling time: base on the required conversion speed
  For example, select 239.5 cycles will result as 239.5 + 12 * 1/14000000 = 18 us (one conversion time), i.e. 55.5 KHz conversion rate.



Setup ADC in polling mode

6.3. Generated code

An ADC instance `ADC_HandleTypeDef hadc;` is created and the that ADC is initialized in the function `MX_ADC_Init()` in which the common settings will be applied. After that, it enable all selected channel for a converting sequence.

```
static void MX_ADC_Init_Single_Conversion(void) {
  ADC_ChannelConfTypeDef sConfig = {0};
  /** Configure the global features of the ADC
  (Clock, Resolution, Data Alignment and number of conversion) */
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc.Init.Resolution = ADC_RESOLUTION_12B;
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
  hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;
  hadc.Init.LowPowerAutoPowerOff = DISABLE;
  hadc.Init.ContinuousConvMode = DISABLE;
  hadc.Init.DiscontinuousConvMode = DISABLE;
  hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc.Init.DMAContinuousRequests = DISABLE;
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK) { Error_Handler(); }
}
```

6.4. Single-Channel, Single Conversion

To read an ADC channel, it is needed to configure one and only target channel. If there is more than one channel, ADC module will act as scan mode, and if the target channel has its number less than the first enabled channel, it will not be read.

Here is the sequence to read a channel

1. Clear all channel registered in Channel Selection `CHSELR` register by assigning it to 0.
2. Enable the target channel by calling to `HAL_ADC_ConfigChannel()` with a configs having: Channel Number, `ADC_RANK_CHANNEL_NUMBER` rank, and Sample Time
3. Call to `HAL_ADC_Start()`
4. Call to `HAL_ADC_PollForConversion()`
5. Read the returned value with `HAL_ADC_GetValue()`
6. Clear all channel registered in Channel Selection `CHSELR` register by assigning it to 0.

```
void Add_ADC_Channel(ADC_HandleTypeDef* hadc, uint32_t channel, uint32_t
sampling_time) {
  ADC_ChannelConfTypeDef sConfig = {0};
  /** Configure for the selected ADC regular channel to be converted. */
  sConfig.Channel = channel;
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = sampling_time;
  if (HAL_ADC_ConfigChannel(hadc, &sConfig) != HAL_OK) { Error_Handler(); }
}
```

```
uint32_t Read_ADC_Channel(ADC_HandleTypeDef* hadc, uint32_t channel, uint32_t
sampling_time) {
  uint32_t adc_value = 0;
  // clear all channel
  hadc->Instance->CHSELR = 0;
  Add_ADC_Channel(hadc, channel, sampling_time);
  // read
  HAL_ADC_Start(hadc);
  HAL_ADC_PollForConversion(hadc, HAL_MAX_DELAY);
  adc_value = HAL_ADC_GetValue(hadc);
  HAL_ADC_Stop(hadc);
  // clear all channel
  hadc->Instance->CHSELR = 0;
  // return
  return adc_value;
}
```

Read Joystick's position

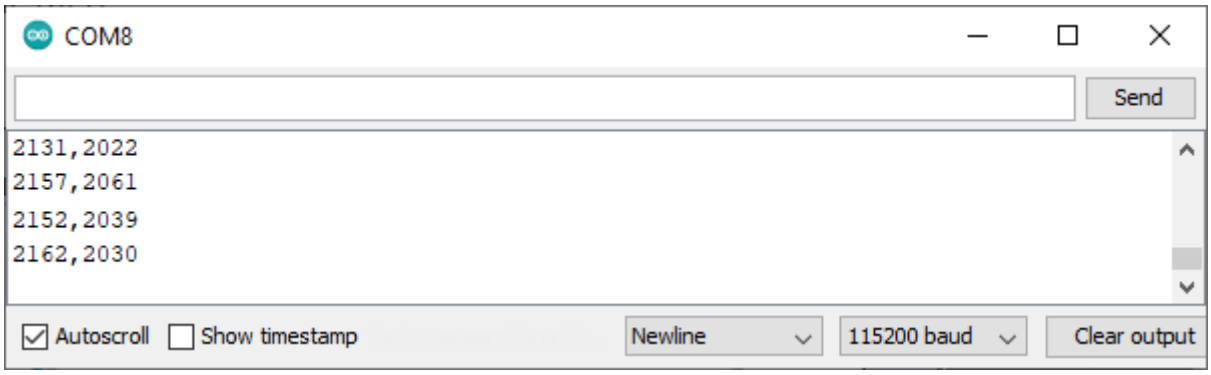
As the ADC value is an 12-bit register, Joystick jas 2 channels, an array will be used to store the Joystick's position. A buffer also is used to hold the message which will be sent on UART1.

```
uint16_t joystick[2] = {0};
char buffer[12] = {0}; // xxxx,yyyy\r\n
```

It is recommended to do ADC Calibration before activating the ADC conversion. Then in the main while loop, call to `Read_ADC()` on `ADC_CHANNEL_1` and `ADC_CHANNEL_2`. The 20 ms delay will help to reduce the conversion rate to about 50 Hz.

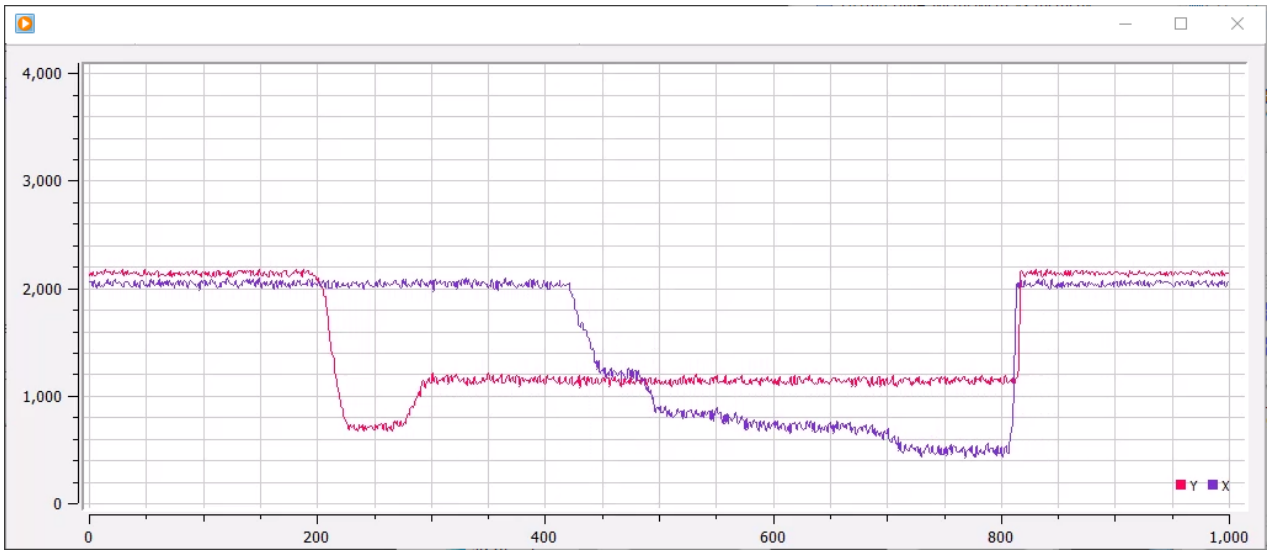
```
int main(void) {
    #if defined(ADC_1_CH_SINGLE_MODE)
    MX_ADC_Init_Single_Conversion();
    HAL_ADCEx_Calibration_Start(&hadc);
    #endif
    while (1) {
        #ifdef ADC_1_CH_SINGLE_MODE
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
        joystick[JS_X] = Read_ADC_Channel(&hadc, ADC_CHANNEL_1,
                                          ADC_SAMPLETIME_239CYCLES_5);
        joystick[JS_Y] = Read_ADC_Channel(&hadc, ADC_CHANNEL_2,
                                          ADC_SAMPLETIME_239CYCLES_5);
        #endif
        HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer),
                                          HAL_MAX_DELAY);
        HAL_Delay(20);
    }
}
```

Run the application and connect to the UART1 terminal, value of `VRX` and `VRY` will be printed out, like below:



Joystick's position is printed in raw string

To visualize the data, use Arduino Serial Plotter feature or SerialPlot program.



Joystick's value in a timeline graph

6.5. Multiple-Channel, Single Conversion

In the **Single Conversion** mode, if multiple channels are registered, ADC module will automatically run int Multiple-Channel scan mode.

> ⚠ **F0/L0 families use the channel numbers as the scan order in Multiple-Channel mode**

In the **Multiple-Channel mode**, ADC module starts with the first activated channel which has smallest channel number when the function `HAL_ADC_Start()` is called.

For each subsequence call of `HAL_ADC_PollForConversion()`, ADC module read the channel then move to the next channel, and wait for next polling call.

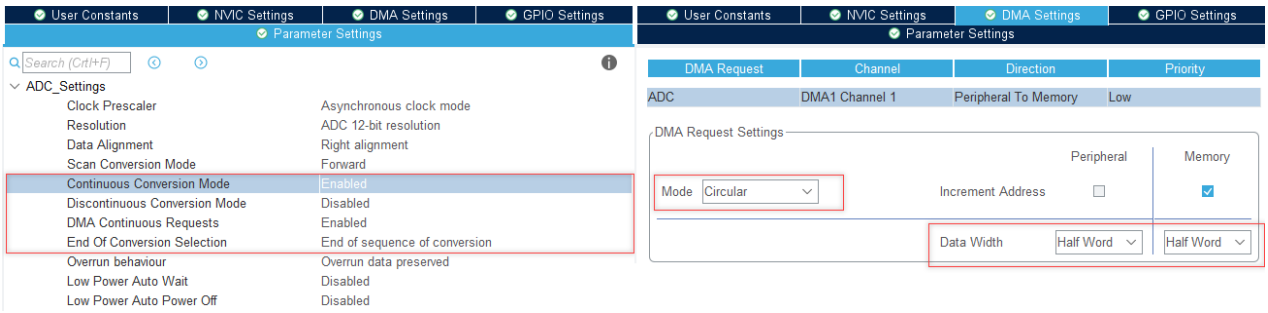
```
int main(void) {
    #if defined(ADC_1_CH_SINGLE_MODE) || defined(ADC_N_CH_SINGLE_MODE)
    MX_ADC_Init_Single_Conversion();
    HAL_ADCEx_Calibration_Start(&hadc);
    #endif
    while (1) {
        #ifdef ADC_N_CH_SINGLE_MODE
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
        HAL_ADC_Start(&hadc);
        HAL_ADC_PollForConversion(&hadc, HAL_MAX_DELAY);
        joystick[JS_X] = HAL_ADC_GetValue(&hadc);
        HAL_ADC_PollForConversion(&hadc, HAL_MAX_DELAY);
        joystick[JS_Y] = HAL_ADC_GetValue(&hadc);
        HAL_ADC_Stop(&hadc);
        #endif
        HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer),
                                            HAL_MAX_DELAY);
        HAL_Delay(20);
    }
}
```

6.6. Continuous Conversion

ADC module will not stop in this mode. When it finishes converting the last channel, it go back to the first channel and then do conversion again.

The modified points are:

- **Continuous Conversion** Mode is *enabled*
- **DMA Continuous Request** is *enabled*
- **End of Conversion flag** is selected to be set at *End of sequence of conversion*
- **DMA Settings** is set to *Circular* mode, Data *increment for Memory* with *Half-Word* width



Setup ADC in continuous mode with DMA

```
void MX_ADC_Init_Continuous_Conversion(void) {
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
```

```
    hadc.Init.Resolution = ADC_RESOLUTION_12B;
    hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
    hadc.Init.EOCSelection = ADC_EOC_SEQ_CONV;
    hadc.Init.LowPowerAutoWait = DISABLE;
    hadc.Init.LowPowerAutoPowerOff = DISABLE;
    hadc.Init.ContinuousConvMode = ENABLE;
    hadc.Init.DiscontinuousConvMode = DISABLE;
    hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
    hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc.Init.DMAContinuousRequests = ENABLE;
    hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
    if (HAL_ADC_Init(&hadc) != HAL_OK) {
      Error_Handler();
    }
  }
```

Then DMA moduel is linked to the ADC module in the function `HAL_ADC_MspInit()` :

```
  void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc) {
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    if(hadc->Instance==ADC1) {
      __HAL_RCC_ADC1_CLK_ENABLE();
      __HAL_RCC_GPIOA_CLK_ENABLE();
      /**ADC GPIO Configuration
      PA1     ------> ADC_IN1
      PA2     ------> ADC_IN2 */
      GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_2;
      GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
      GPIO_InitStruct.Pull = GPIO_NOPULL;
      HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
      /* ADC1 DMA Init */
      /* ADC Init */
      hdma_adc.Instance = DMA1_Channel1;
      hdma_adc.Init.Direction = DMA_PERIPH_TO_MEMORY;
      hdma_adc.Init.PeriphInc = DMA_PINC_DISABLE;
      hdma_adc.Init.MemInc = DMA_MINC_ENABLE;
      hdma_adc.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
      hdma_adc.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
      hdma_adc.Init.Mode = DMA_CIRCULAR;
      hdma_adc.Init.Priority = DMA_PRIORITY_LOW;
      if (HAL_DMA_Init(&hdma_adc) != HAL_OK) {
        Error_Handler();
      }
      __HAL_LINKDMA(hadc,DMA_Handle,hdma_adc);
    }
  }
```

After that, add channel want to be read, and start the ADC in DMA mode. Make sure to set the buffer and the length for DMA.

```
int main() {
#if defined(ADC_1_CH_CONTINUOUS_MODE)
    Add_ADC_Channel(&hadc, ADC_CHANNEL_1, ADC_SAMPLETIME_239CYCLES_5);
#endif
#if defined(ADC_N_CH_SINGLE_MODE) || defined(ADC_N_CH_CONTINUOUS_MODE)
    Add_ADC_Channel(&hadc, ADC_CHANNEL_1, ADC_SAMPLETIME_239CYCLES_5);
    Add_ADC_Channel(&hadc, ADC_CHANNEL_2, ADC_SAMPLETIME_239CYCLES_5);
#endif

#if defined(ADC_1_CH_CONTINUOUS_MODE)
    HAL_ADC_Start_DMA(&hadc, (uint32_t*)joystick, 1);
#endif

#if defined(ADC_N_CH_CONTINUOUS_MODE)
    HAL_ADC_Start_DMA(&hadc, (uint32_t*)joystick, 2);
#endif
    while (1)
    {
        HAL_Delay(20);
    }
}
```

That's all for this tutorial. The Discontinuous and Injected Scan mode will be covered in other guides.