The Analog to Digital Converter (ADC)

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory

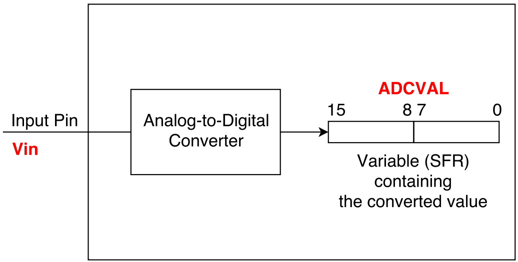
Dipartimento di Matematica e Informatica - Università di Catania, Italy

santoro@dmi.unict.it

L.A.P. 1 Course

What is an ADC?

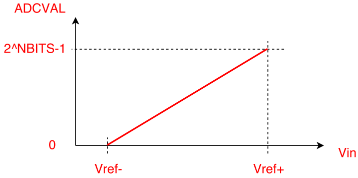
An ADC (Analog-to-Digital-Converter) is a circuit which gets an **analog voltage signal** (as input) and provides (to software) a **integer variable** proportional to the input signal.



ADC Characteristics

An ADC is characterised by:

- The **voltage range** of the input signal, V_{ref-}, V_{ref+}
  - the input signal must always be in the interval $[V_{ref-}, V_{ref+}]$
- The **resolution** in **bits** of the converter, *NBITS*.
- The ADC works by using a **linear law**:
  - If $V_{in} = V_{ref-}$, then $ADCVAL = 0$
  - If $V_{in} = V_{ref+}$, then $ADCVAL = 2^{NBITS} - 1$



$$ADCVAL = \left[(V_{in} - V_{ref-}) \frac{2^{NBITS} - 1}{V_{ref+} - V_{ref-}} \right]$$

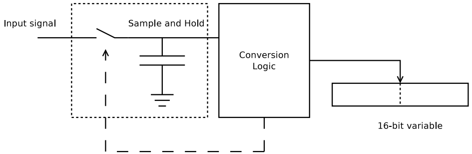
ADC Characteristics

- In general, $V_{ref-} = 0$ (GND) and $V_{ref+} = VDD$ (power supply voltage, i.e. 5 V or 3.3 V)
- In our Nucleo board, $VDD = 3.3\ V$ therefore $V_{ref+} = 3.3\ V$
- In this case, the conversion law becomes:

$$ADCVAL = \left[V_{in} \frac{2^{NBITS} - 1}{3.3} \right]$$

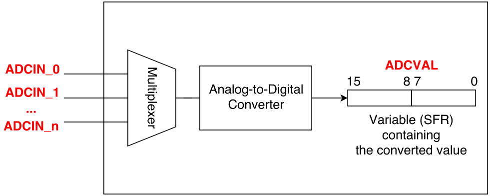
ADC: Basic working scheme

The ADC is a **sequential circuit** that performs conversion using a sequence of steps:



1. **Sample**: the signal is *sampled* by closing the switch and charging the capacitor; the duration of this phase is denoted as T_{samp}
2. **Conversion**: the switch is open and the sampled signal is *converted*; the result is stored in the 16-bit variable. The duration of this phase is denoted as T_{conv}
3. **End-of-conversion**: a proper bit is set to signal that the operation has been done.

ADC inputs



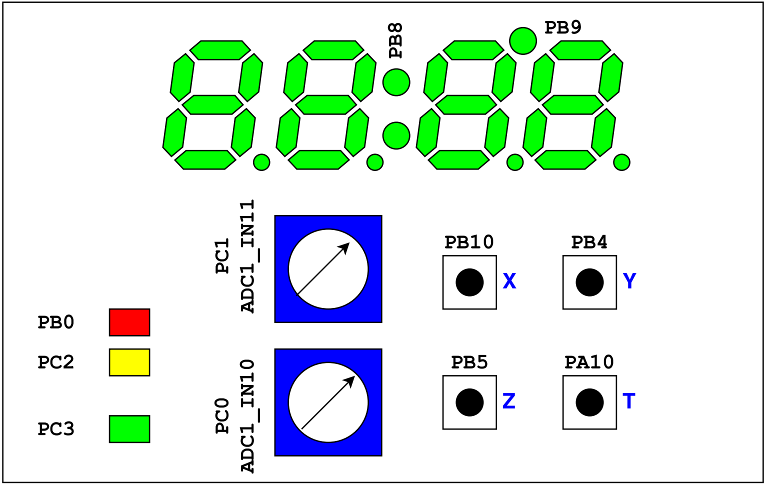
- In general, an ADC has **several inputs**
- But only **one input (channel) at time** can be selected for conversion (through the multiplexer)
- To perform conversion, the software must:
  - Select the input channel to be converted
  - Start the conversion (by setting a proper bit in a SFR)
  - Wait for the end-of-conversion (by checking a proper bit in a SFR), or
  - being notified of the end-of-conversion through an IRQ

ADC inputs on STM32F401

- In the STM32F401 MCU, ADC inputs share the same pin of GPIO ports
- In particular, some GPIO pins can be programmed in order to be served as **analog input channel** (and no more used as digital I/O):

Pin	Analog Channel		Pin	Analog Channel
PA0	ADC1_IN0		PA1	ADC1_IN1
PA2	ADC1_IN2		PA3	ADC1_IN3
PA4	ADC1_IN4		PA5	ADC1_IN5
PA6	ADC1_IN6		PA7	ADC1_IN7
PB0	ADC1_IN8		PB1	ADC1_IN9
PC0	ADC1_IN10		PC1	ADC1_IN11
PC2	ADC1_IN12		PC3	ADC1_IN13
PC4	ADC1_IN14		PC5	ADC1_IN15

The Nucleo64 Addon Board (look at ADC settings)



ADC characteristics on STM32F4xx

- In the STM32F4xx MCUs, the ADCs have **configurable resolution**:
  - 6 bits, range $[0, 63]$
  - 8 bits, range $[0, 255]$
  - 10 bits, range $[0, 1023]$
  - 12 bits, range $[0, 4095]$
- The conversion result may be aligned **left** or **right** in the 16 bit result, e.g.:
- 12bit Left-Aligned

b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	0	0	0	0

- 12bit Right-Aligned

0	0	0	0	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

The Software interface of ADCs

- Each ADC has several special function registers
- All of them are accessible by means of global variables called **`ADCx`**, where **`x`** is the number of the adc (our micro has only ADC1) (**`ADC1`**, **`ADC2`**, ...)
- The type of these variables is **`ADC_TypeDef *`**, i.e. pointers to a structure whose field are the SFR of the ADC

stm32_unict_lib Functions for ADCs

- Initialize an ADC:

```
void ADC_init(ADC_TypeDef * adc, int res, int align);
```

  - **adc**, the ADC circuit
  - **res**, the resolution in bits
    - `ADC_RES_6`
    - `ADC_RES_8`
    - `ADC_RES_10`
    - `ADC_RES_12`
  - **align**, the bit alignment
    - `ADC_ALIGN_RIGHT`
    - `ADC_ALIGN_LEFT`

- Configure the input(s):

```
void ADC_channel_config(ADC_TypeDef * adc,
                        GPIO_TypeDef * port,
                        int pin, int chan);
```

  - **adc**, the ADC circuit
  - **port**, the GPIO port of the input
  - **pin**, the GPIO pin of the input
  - **chan**, the ADC channel associated to the input

- Start an ADC circuit:

```
void ADC_on(ADC_TypeDef * adc);
```

- Stop an ADC circuit:

```
void ADC_off(ADC_TypeDef * adc);
```

stm32_unict_lib Functions for ADCs

- Select a channel to convert:
  `void ADC_sample_channel(ADC_TypeDef * adc, int chan);`
  - **adc**, the ADC circuit
  - **chan**, the ADC channel to be converted
- Start a sample+conversion of the selected channel:
  `void ADC_start(ADC_TypeDef * adc);`
- Check if a conversion has been completed:
  `int ADC_completed(ADC_TypeDef * adc);`
- Read the converted value:
  `int ADC_read(ADC_TypeDef * adc);`

Sampling the ADC and showing the value

```c
#include <stdio.h>
#include "stm32_unict_lib.h"

void main(void)
{
    DISPLAY_init();

    ADC_init(ADC1, ADC_RES_8, ADC_ALIGN_RIGHT);
    ADC_channel_config(ADC1, GPIOC, 0, 10);
    ADC_on(ADC1);
    ADC_sample_channel(ADC1, 10);

    for (;;) {
        ADC_start(ADC1);
        while (!ADC_completed(ADC1)) {}

        int value = ADC_read(ADC1);
        char s[4];
        sprintf(s, "%4d", value);
        DISPLAY_puts(0,s);
    }
}
```

Exercise: Let's flash a LED with a variable period

- We want to make a LED flash (with a timer) with a period ranging from 50 to 500 ms
- The period must be set using the trimmer in PC0/ADC1_IN10

- Let's initialize the timebase of a timer to 0.5 ms
- The auto-reload value must be in the range [100, 1000]
- If we set the ADC to 8 bit, we can use the formula:

$$ARR = ADCVAL \frac{1000 - 100}{255} + 100$$

LED flash with variable period

```c
#include <stdio.h>
#include "stm32_unict_lib.h"

int new_arr_value = 100;

void main(void)
{
    DISPLAY_init();
    GPIO_init(GPIOB);   GPIO_config_output(GPIOB, 0);

    ADC_init(ADC1, ADC_RES_8, ADC_ALIGN_RIGHT);
    ADC_channel_config(ADC1, GPIOC, 0, 10);
    ADC_on(ADC1); ADC_sample_channel(ADC1, 10);

    TIM_init(TIM2);
    TIM_config_timebase(TIM2, 42000, 100);
    TIM_set(TIM2, 0); TIM_enable_irq(TIM2, IRQ_UPDATE);
    TIM_on(TIM2);

    for (;;) {
        ADC_start(ADC1);
        while (!ADC_completed(ADC1)) {}
        int value = ADC_read(ADC1);
        new_arr_value = value * 900/255 + 100;
        char s[4];
        sprintf(s, "%4d", new_arr_value / 2); // we will display the milliseconds
        DISPLAY_puts(0,s);
    }
}
```

LED flash with variable period (II)

```
void TIM2_IRQHandler(void)
{
    if (TIM_update_check(TIM2)) {
        GPIO_toggle(GPIOB, 0);
        TIM_update_clear(TIM2);
        TIM2->ARR = new_arr_value;
        // update the autoreload register with new value
    }
}
```

The Analog to Digital Converter (ADC)

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
`santoro@dmi.unict.it`

L.A.P. 1 Course