

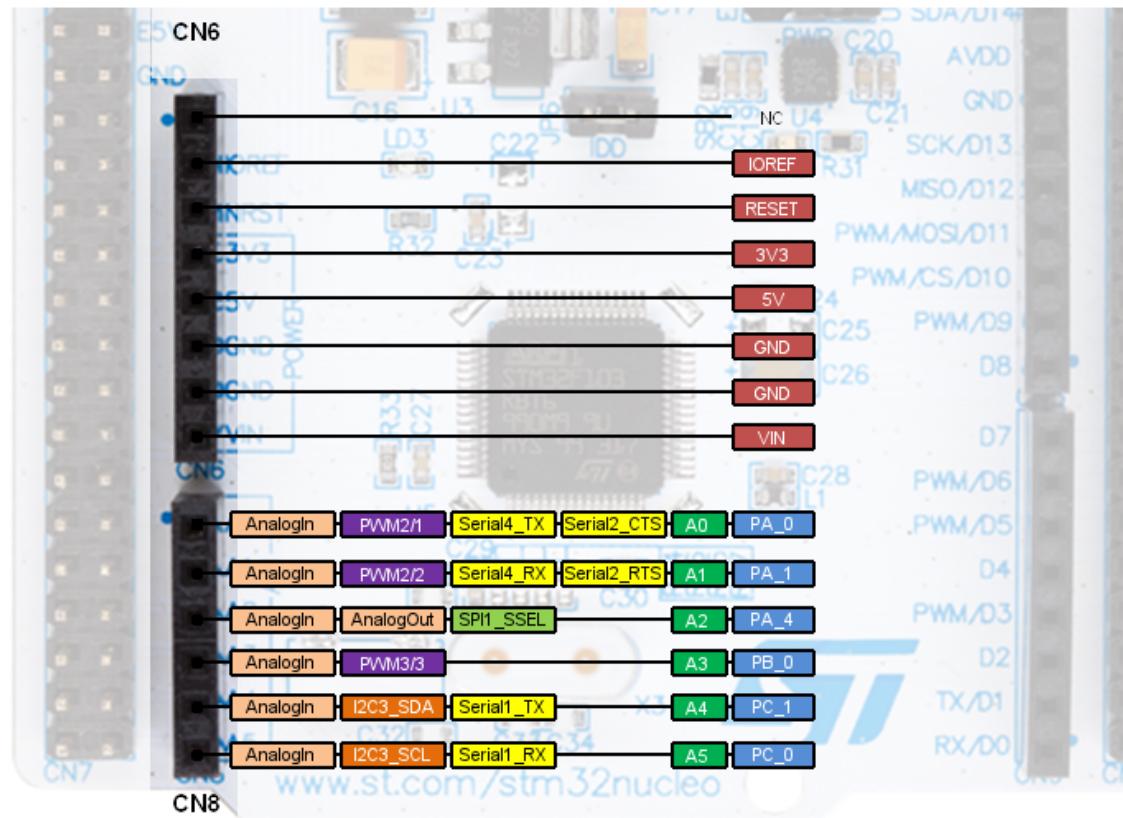
USART

Universal synchronous asynchronous receiver transmitter



life.augmented

NUCLEO-L476RG
ARDUINO HEADER
(top left side)

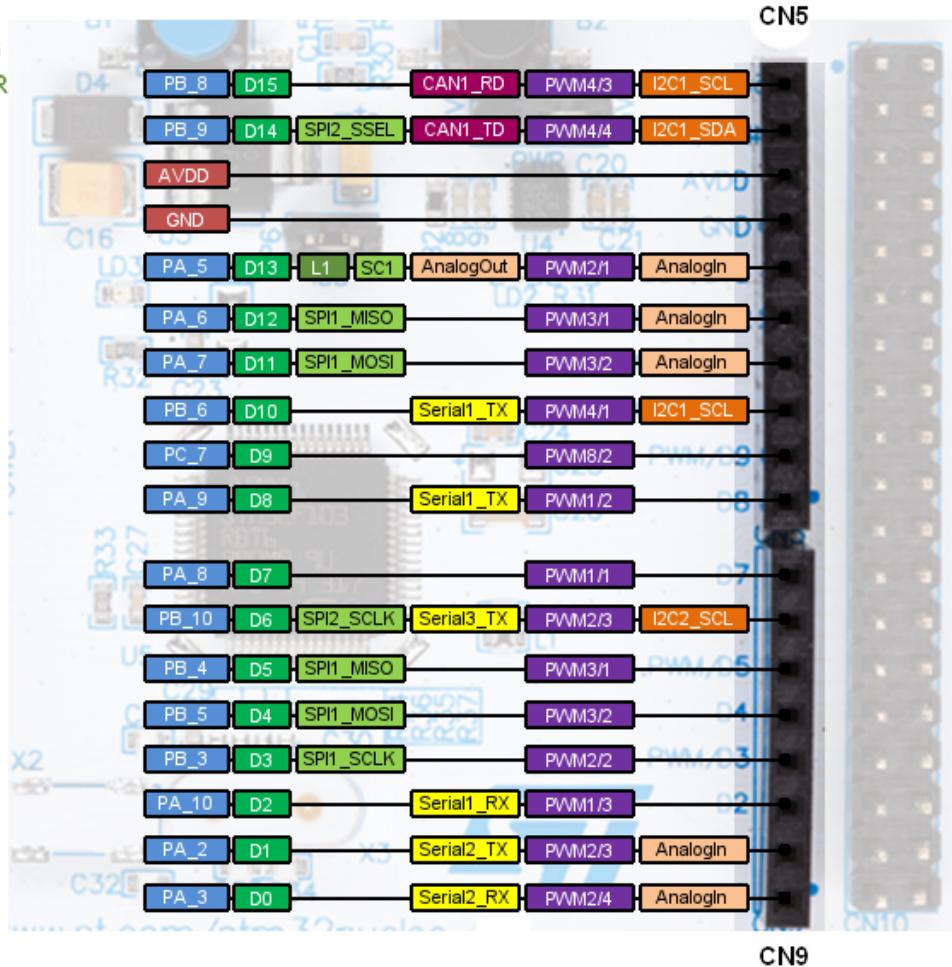




life.augmented

NUCLEO-L476RG

ARDUINO HEADER
(top right side)

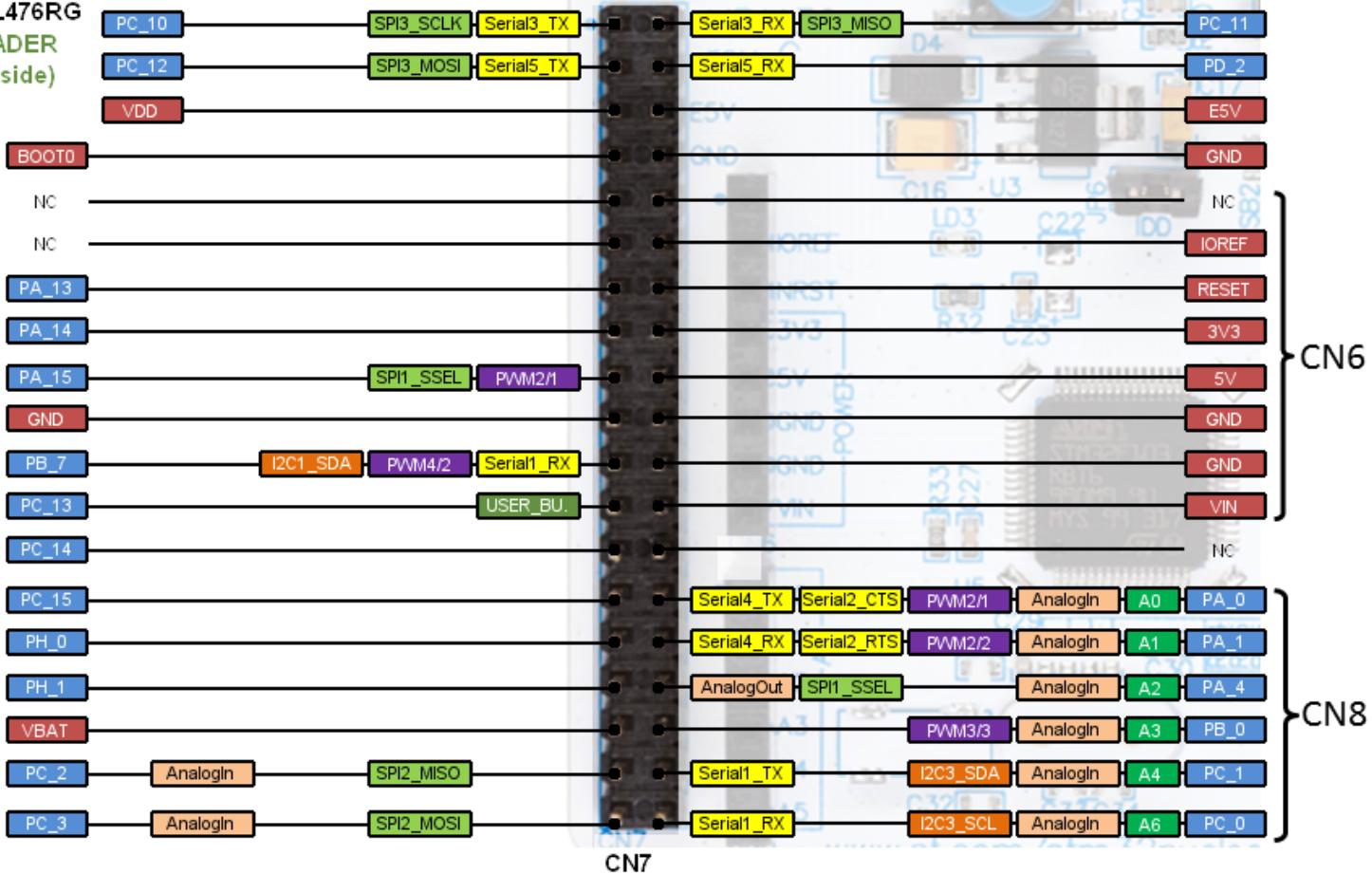




life.augmented

NUCLEO-L476RG

CN7 HEADER
(top left side)

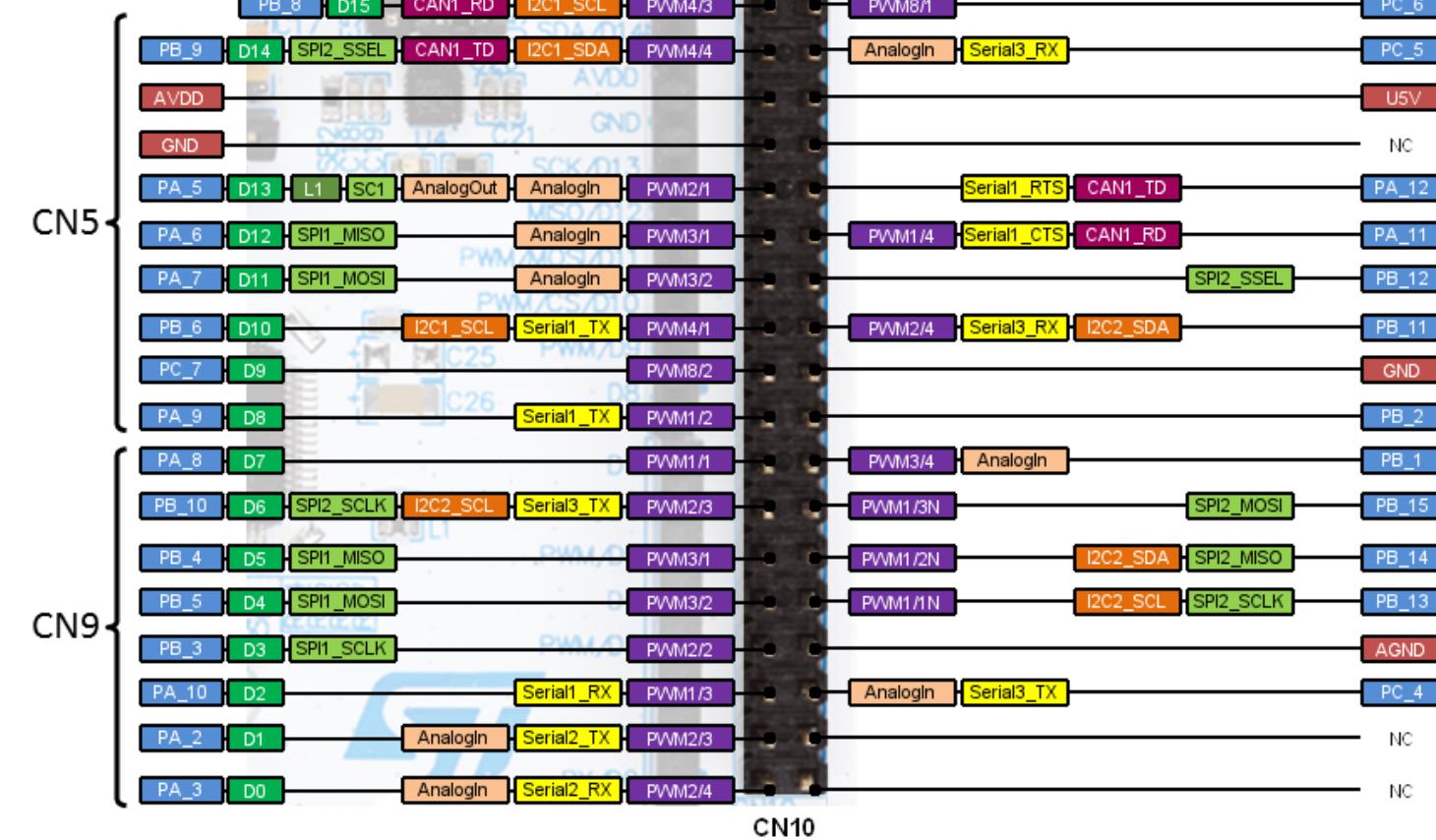


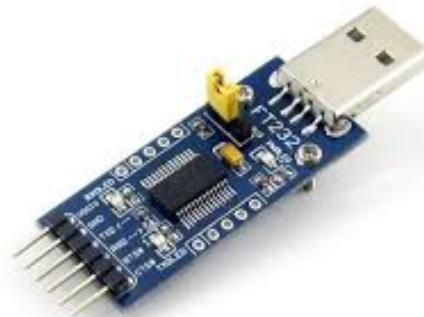
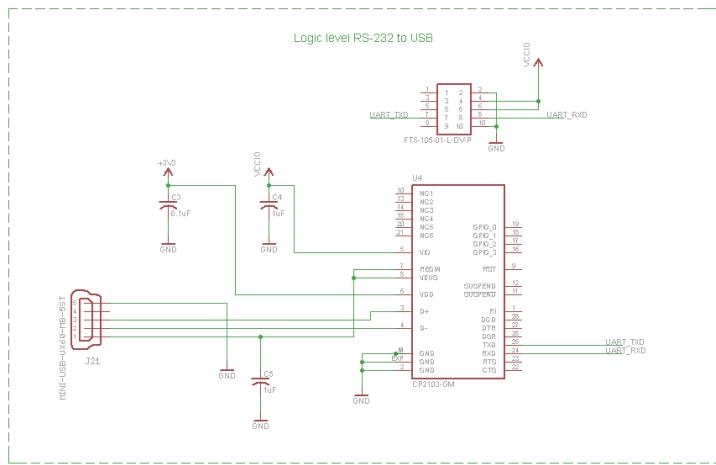


life.augmented

NUCLEO-L476RG

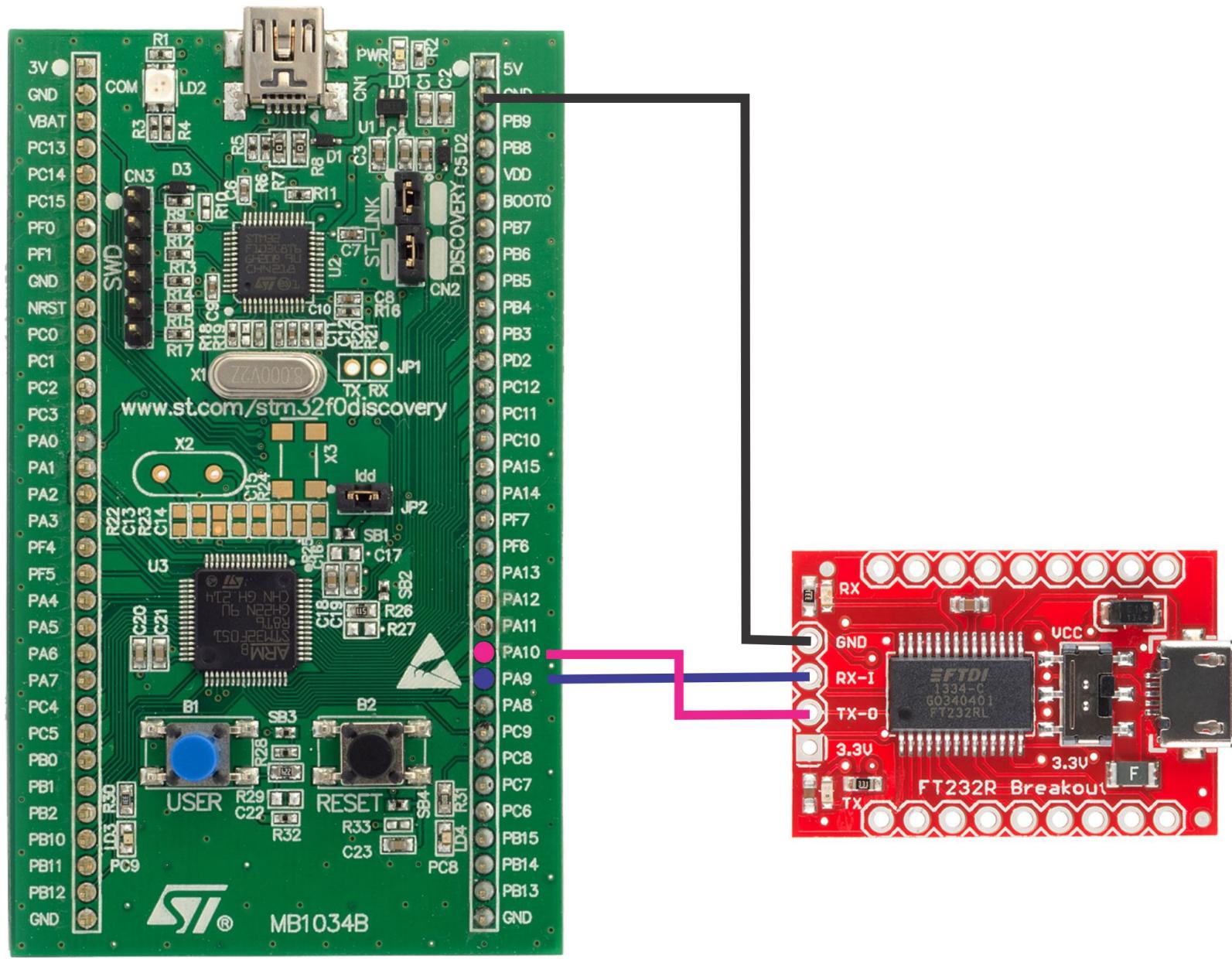
CN10 HEADER
(top right side)



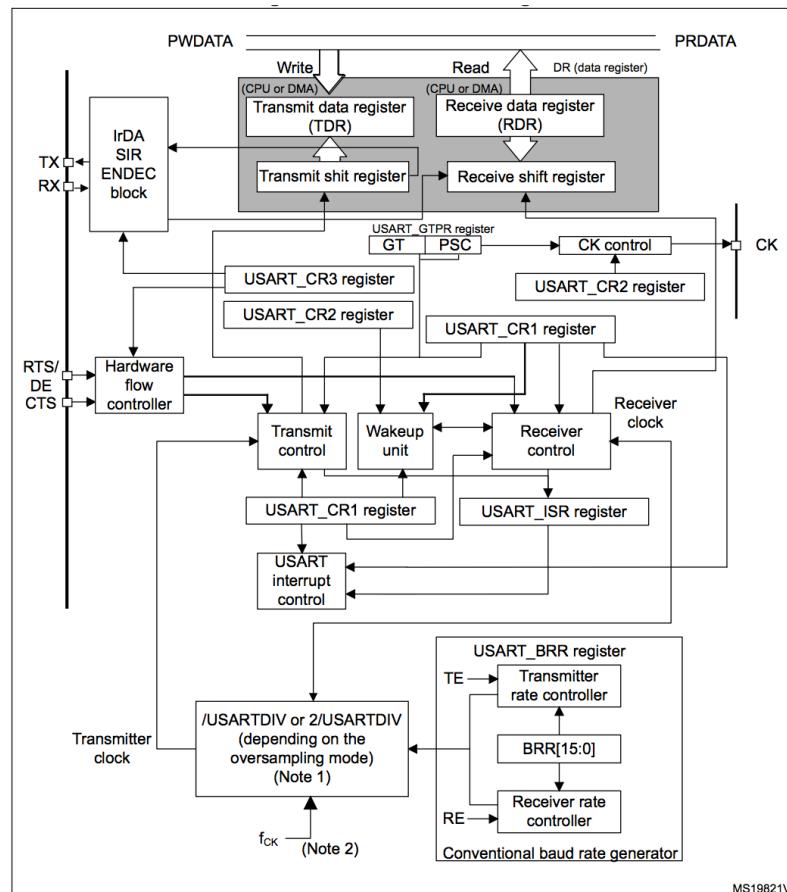


Arrow on UART
Connector Indicates
Pin 1





Block Diagram of USART in STM32L476xx



USART of STM32

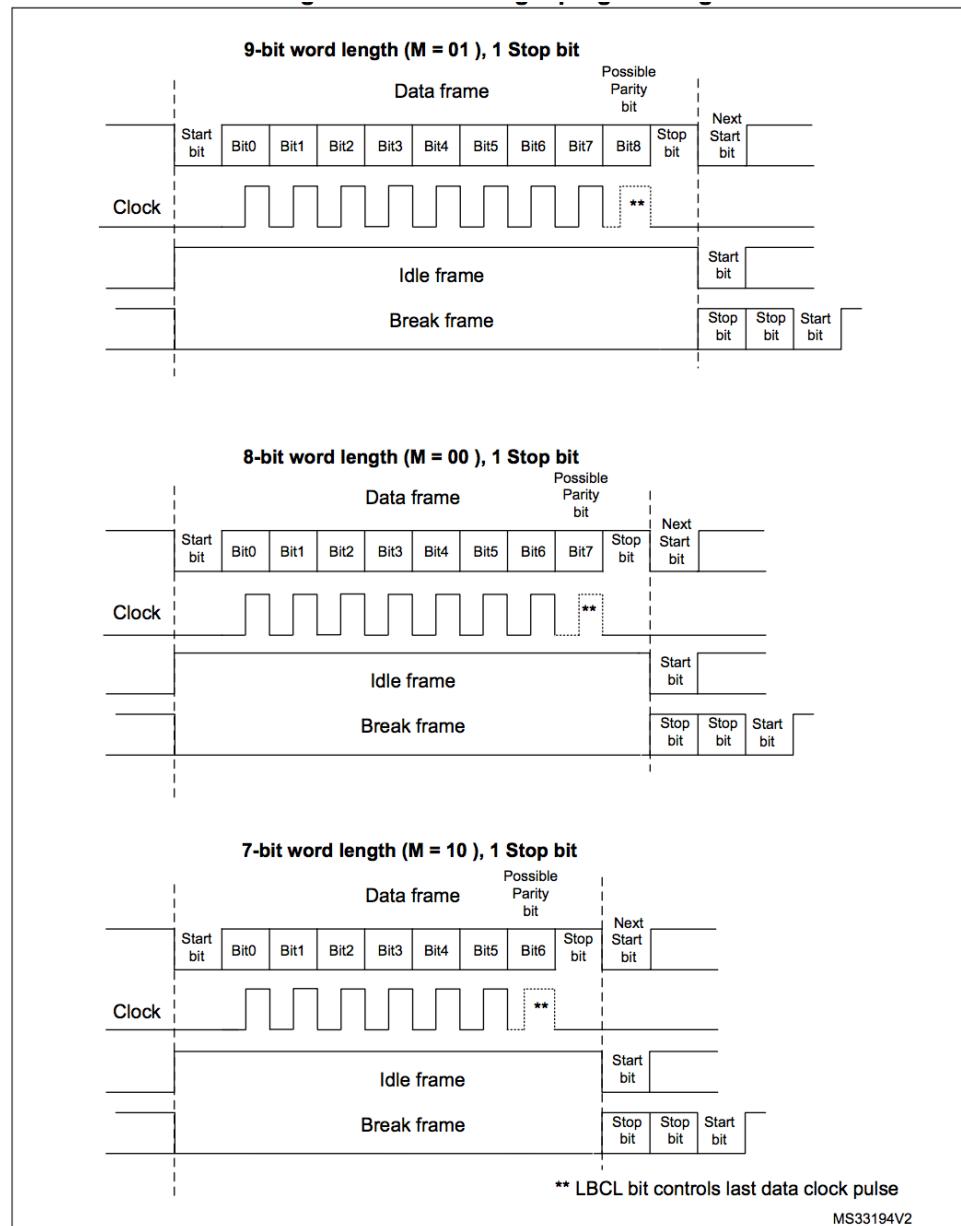
- Configurable oversampling method by 16 or 8 to give flexibility between speed and clock tolerance
- A common programmable transmit and receive baud rate of up to 10 Mbit/s when the clock frequency is 80 MHz and oversampling is by 8
- Programmable data word length (7, 8 or 9 bits)
- Programmable data order with MSB-first or LSB-first shifting
- Configurable stop bits (1 or 2 stop bits)
- Parity control

Functions

- RX: Receive data Input.
 - Oversampling techniques are used for data recovery by discriminating between valid incoming data and noise.
- TX: Transmit data Output.

Character Description

- The word length can be selected as being either 7 or 8 or 9 bits by programming the M[1:0] bits in the USART_CR1 register
 - 7-bit character length: M[1:0] = 10
 - 8-bit character length: M[1:0] = 00
 - 9-bit character length: M[1:0] = 01
- An Idle character is interpreted as an entire frame of “1”s (the number of “1”s includes the number of stop bits).
- A Break character is interpreted on receiving “0”s for a frame period. At the end of the break frame, the transmitter inserts 2 stop bits.



USART Baud Rate

- The baud rate for the receiver and transmitter (Rx and Tx) are both set to the same value as programmed in the USART_BRR register.
- In case of oversampling by 16, the equation is: $\text{Baud} = \frac{f_{CK}}{\text{USARTDIV}}$
- In case of oversampling by 8, the equation is: $\text{Baud} = \frac{2 \times f_{CK}}{\text{USARTDIV}}$
- When OVER8 = 0, BRR = USARTDIV
- When OVER8 = 1
 - BRR[2:0] = USARTDIV[3:0] shifted 1 bit to the right.
 - BRR[3] must be kept cleared.
 - BRR[15:4] = USARTDIV[15:4]

USART Transmitter

- The Transmit Enable bit (TE) must be set in order to activate the transmitter function
- Every character is preceded by a start bit which is a logic level low for one bit period. The character is terminated by a configurable number of stop bits.

Transmission Procedure

- Program the M bits in USART_CR1 to define the word length.
- Select the desired baud rate using the USART_BRR register.
- Program the number of stop bits in USART_CR2.
- Enable the USART
- Write the data to send in the USART_TDR register
- After writing the last data into the USART_TDR register, wait until TC=1. This indicates that the transmission of the last frame is complete

USART Reception

- In the USART, the start bit is detected when a specific sequence of samples is recognized. This sequence is: 1 1 1 0 X 0 X 0 X 0 X 0 X 0.
- The start bit is confirmed (RXNE flag set, interrupt generated if RXNEIE=1) if the 3 sampled bits are at 0 (first sampling on the 3rd, 5th and 7th bits finds the 3 bits at 0 and second sampling on the 8th, 9th and 10th bits also finds the 3 bits at 0).

Reception Procedure

- Program the M bits in USART_CR1 to define the word length.
- Select the desired baud rate using the baud rate register USART_BRR
- Program the number of stop bits in USART_CR2.
- Enable the USART by writing the UE bit in USART_CR1 register to 1.
- Set the RE bit USART_CR1. This enables the receiver which begins searching for a start bit.
- When a character is received, the RXNE bit is set to indicate that the content of the shift register is transferred to the RDR.

Procedures

- Enable UART
- Sending Data

```
GPIO_Init();
USART???_Init();
UART _Transmit((uint8_t*)"???", 3);
```

```
void GPIO_Init(void) {
    RCC->AHB2ENR |= (RCC_AHB2ENR_GPIOAEN | RCC_AHB2ENR_GPIOBEN | RCC_AHB2ENR_GPIOCEN);

    // UART
    TM_GPIO_Init(???, ?, TM_GPIO_Mode_AF, TM_GPIO_OType_PP, TM_GPIO_PuPd_NOPULL, TM_GPIO_Speed_Low);
    TM_GPIO_Init(???, ?, TM_GPIO_Mode_AF, TM_GPIO_OType_PP, TM_GPIO_PuPd_NOPULL, TM_GPIO_Speed_Low);
    GPIOB->AFR[0] = (GPIOB->AFR[0] & ~0xFF000000) | 0x77000000; // AF7 for pin

    // BUTTON
    TM_GPIO_Init(GPIOC, 13, TM_GPIO_Mode_IN, TM_GPIO_OType_PP, TM_GPIO_PuPd_NOPULL, TM_GPIO_Speed_Medium);
}
```

```
huart2.Instance = USART2; huart2.Init.BaudRate = 9600;
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;
```

```
void USART1_Init(void) {
/* Enable clock for USART??? */
    RCC->APB2ENR |= RCC_APB2ENR_USART???EN;

    // CR1
    MODIFY_REG(USART???->CR1, USART_CR1_M | USART_CR1_PS | USART_CR1_PCE | USART_CR1_TE | USART_CR1_RE |
    USART_CR1_OVER8, USART_CR1_TE | USART_CR1_RE);

    // CR2
    MODIFY_REG(USART???->CR2, USART_CR2_STOP, 0x0); // 1-bit stop

    // CR3
    MODIFY_REG(USART???->CR3, (USART_CR3_RTSE | USART_CR3_CTSE | USART_CR3_ONEBIT), 0x0); // none hwflowctl
    MODIFY_REG(USART???->BRR, 0xFF, 4000000L/???? L);

/* In asynchronous mode, the following bits must be kept cleared:
- LINEN and CLKEN bits in the USART_CR2 register,
- SCEN, HDSEL and IREN bits in the USART_CR3 register.*/
    USART???->CR2 &= ~(USART_CR2_LINEN | USART_CR2_CLKEN);
    USART???->CR3 &= ~(USART_CR3_SCEN | USART_CR3_HDSEL | USART_CR3_IREN);

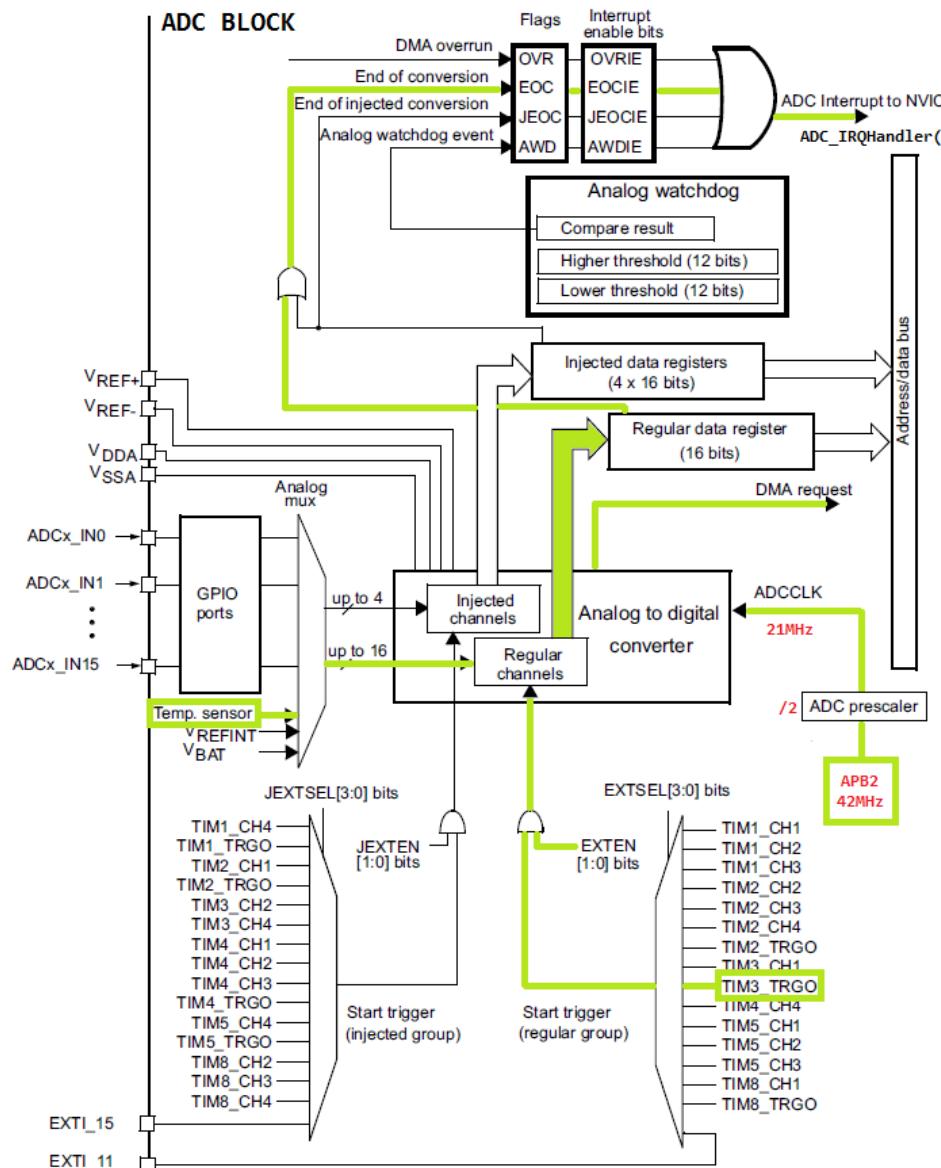
    // Enable USART
    USART???->CR1 |= (USART_CR1_UE);
}
```

```
int UART_Transmit(uint8_t *arr, uint32_t size) {  
    for (int i = 0; i < ???; i++) {  
        while (!IS_UART_TRANS_READY);  
        USART???->TDR = ???;  
    }  
    while (!IS_UART_TRANS_DONE);  
    return ???;  
}
```

Hello World!

- 在按下板子上藍色按鈕時(PC13)，請利用UART將”Hello World!”字串傳送出去，並且可以在電腦端接受並顯示出來。
 - Init需要使用到的GPIO。
 - 瞭解UART的暫存器以及使用方式
 - 利用UART的傳出(TX)來實作此功能

ADC



This is what hardware is doing in the diagram:

- Temp. sensor is ON
- ADC is ON
- TIM3 triggers at 2kHz
- ADC converts the voltage and stores it in the regular data register
- ADC makes a DMA request
- DMA moves data to memory
- ADC raises EOC flag which generates an ADC interrupt
- ADC IRQ handler is called at 2kHz toggling a pin at 1kHz (indirectly testing my calculations)

Software will do the following:

- Calibrating the sensor
 - Averaging N ADC samples (VSENSE is the average value)
 - Calculating the absolute temperature
 - Calculating temperature variations
- $$\text{Temp (in } ^\circ\text{C)} = \frac{(VSENSE - V25)}{\text{Avg_Slope}} + 25$$
- (The rest is just for fun)
- Setting some threshold temperatures
 - If the measured temp. is below/above the threshold values, some LEDs light up :)

STM32L476

- Up to 3x ADCs, out of which two of them can operate in dual mode
- 12, 10, 8 or 6-bit configurable resolution
- Channel-wise programmable sampling time
- Start-of-conversion can be initiated:
 - by software
 - by hardware triggers with configurable polarity
- Conversion modes
 - Each ADC can convert a single channel or can scan a sequence of channels
 - Single mode converts selected inputs once per trigger
 - Continuous mode converts selected inputs continuously

Channel Selection (SQRx)

- There are up to 19 multiplexed channels per ADC:
- Before any conversion of an input channel coming from GPIO pads, it is necessary to configure the corresponding GPIO_x_ASCR register in the GPIO, in addition to the I/O configuration in analog mode.
- The regular channels and their order in the conversion sequence must be selected in the ADC_x_SQR registers. The total number of conversions in the regular group must be written in the L[3:0] bits in the ADC_x_SQR1 register.

Channel-wise Programmable Sampling Time (SMPR1, SMPR2)

- Before starting a conversion, the ADC must establish a direct connection between the voltage source under measurement and the embedded sampling capacitor of the ADC.
- Each channel can be sampled with a different sampling time which is programmable using the SMP[2:0] bits in the ADCx_SMPR1 and ADCx_SMPR2 registers.
 - SMP = 000: 2.5 ADC clock cycles
 - SMP = 001: 6.5 ADC clock cycles
 - SMP = 010: 12.5 ADC clock cycles
 - SMP = 011: 24.5 ADC clock cycles
 - SMP = 100: 47.5 ADC clock cycles
 - SMP = 101: 92.5 ADC clock cycles
 - SMP = 110: 247.5 ADC clock cycles
 - SMP = 111: 640.5 ADC clock cycles

Conversion Mode

- In single conversion mode, the ADC performs once all the conversions of the channels.
- In continuous conversion mode, when a software or hardware regular trigger event occurs, the ADC performs once all the regular conversions of the channels and then automatically re-starts and continuously converts each conversions of the sequence.

Programmable Resolution

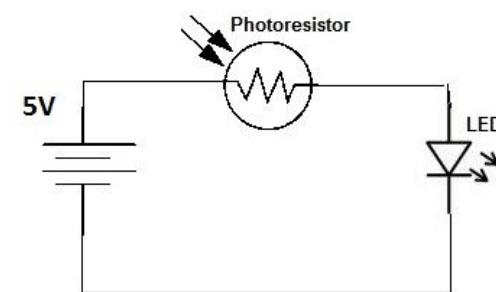
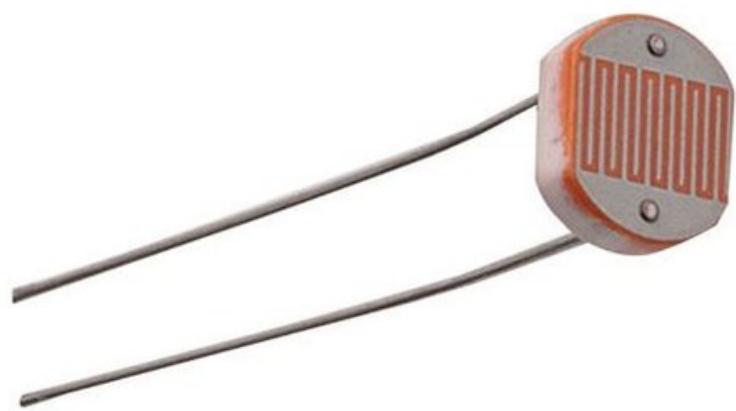
- It is possible to perform faster conversion by reducing the ADC resolution.
- The resolution can be configured to be either 12, 10, 8, or 6 bits by programming the control bits RES[1:0].

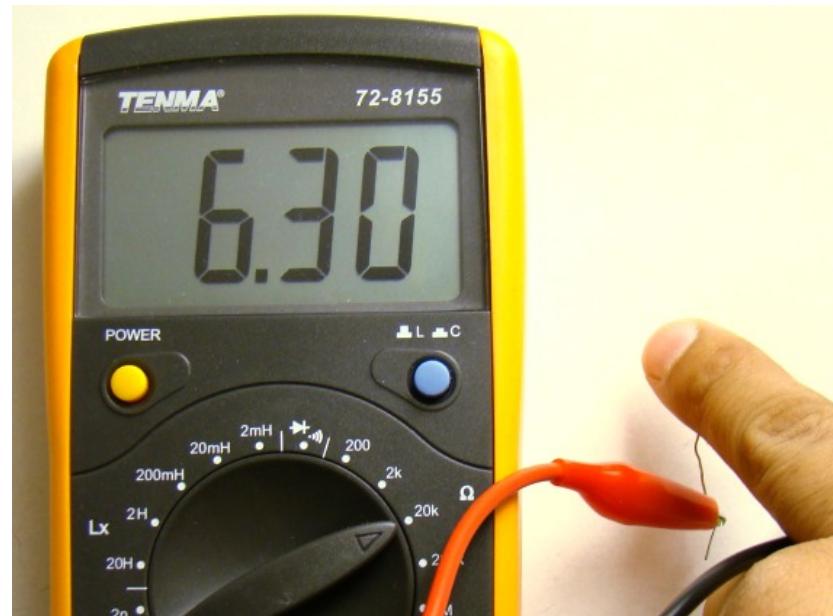
End of Conversion

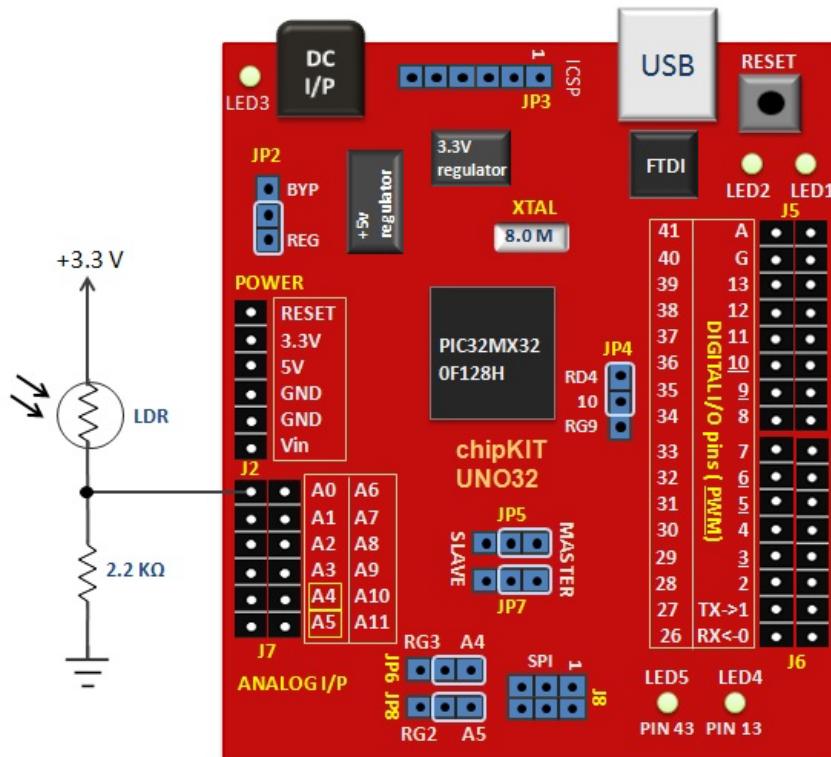
- The ADC sets the EOC flag as soon as a new regular conversion data is available in the ADCx_DR register. An interrupt can be generated if bit EOIE is set. EOC flag is cleared by the software either by writing 1 to it or by reading ADCx_DR.

Procedure

- To start ADC operations, it is first needed to exit deep-power-down mode by setting bit DEEPPWD=0.
- It is mandatory to enable the ADC internal voltage regulator by setting the bit ADVREGEN=1 into ADCx_CR register.
- Once DEEPPWD=0 and ADVREGEN=1, the ADC can be enabled
 - Clear the ADRDY bit in the ADC_ISR register by writing ‘1’.
 - Set ADEN=1.
 - Regular conversion can then start by setting ADSTART=1







```
int main()
{
    TIM3_Config();
    ADC_Config();

    while (1)
    {
        R = Get_R();
        ...
    }
}
```

```
void TIM3_Config(void)
{
    TIM_TimeBaseInitTypeDef TIM3_TimeBase;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

    TIM_TimeBaseStructInit(&TIM3_TimeBase);
    TIM3_TimeBase.TIM_Period      = (uint16_t)49; // Trigger = CK_CNT/(49+1) = 2kHz
    TIM3_TimeBase.TIM_Prescaler   = 420;           // CK_CNT = 42MHz/420 = 100kHz
    TIM3_TimeBase.TIM_ClockDivision = 0;
    TIM3_TimeBase.TIM_CounterMode  = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM3_TimeBase);
    TIM_SelectOutputTrigger(TIM3, TIM_TriggerSource_Update);

    TIM_Cmd(TIM3, ENABLE);
}
```

```
void ADC_Config(void)
{
    ADC_InitTypeDef      ADC_INIT;
    ADC_CommonInitTypeDef ADC_COMMON;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    ADC_COMMON.ADC_Mode          = ADC_Mode_Independent;
    ADC_COMMON.ADC_Prescaler     = ADC_Prescaler_Div2;
    ADC_COMMON.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
    ADC_COMMON.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;
    ADC_CommonInit(&ADC_COMMON);

    ADC_INIT.ADC_Resolution      = ADC_Resolution_12b;
    ADC_INIT.ADC_ScanConvMode    = DISABLE;
    ADC_INIT.ADC_ContinuousConvMode = DISABLE; // ENABLE for max ADC sampling frequency
    ADC_INIT.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_Rising;
    ADC_INIT.ADC_ExternalTrigConv   = ADC_ExternalTrigConv_T3_TRGO;
    ADC_INIT.ADC_DataAlign        = ADC_DataAlign_Right;
    ADC_INIT.ADC_NbrOfConversion  = 1;
    ADC_Init(ADC1, &ADC_INIT);

    ADC-RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_3Cycles);
    ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);
    ADC_Cmd(ADC1, ENABLE);
}
```

```
float Get_R(void)
{
    uint8_t i;

    for(i = 0; i < NS; i++)
    {
        Sample_ADC_Raw[i] = ADC_Raw[i];
    }

    return R;
}
```

```
    /**Configure the global features of the ADC (Clock, Resolution, Data
Alignment and number of conversion)
 */
hadc1.Instance = ADC1;
hadc1.Init.ClockPrescaler = ADC_CLOCKPRESCALER_PCLK_DIV2;
hadc1.Init.Resolution = ADC_RESOLUTION12b;
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_0;
sConfig.Rank = 1;
sConfig.SamplingTime = ADC_SAMPLETIME_15CYCLES;
//sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);
```

```

void ConfigureADC() {
    HAL_RCC_ADC_CLK_ENABLE();
    // Clear Deep Sleep
    CLEAR_BIT(ADC1->CR, ADC_CR_DEEPPWD);
    // Turn on Voltage Regulator
    SET_BIT(ADC1->CR, ADC_CR_ADVREGEN);
    delay_us(200);
    // Prescaler
    MODIFY_REG(ADC123_COMMON->CCR, ADC_CCR_PRESC|ADC_CCR_CKMODE, ADC_CCR_CKMODE_0);
    MODIFY_REG(ADC1->CFGGR, ADC_CFGGR_FIELDS_1, ADC_CFGGR_CONT);
    CLEAR_BIT(ADC1->CFGGR2, ADC_CFGGR2_ROVSE);
    CLEAR_BIT(ADC1->SQR1, ADC_SQR1_L);
    MODIFY_REG(ADC1->SQR1, (0xF) | (0x1F<<24) | (0x1F<<18) | (0x1F<<12) | (0x1F<<6), (0x1<<6)); // Channel 1, Rank 1
    MODIFY_REG(ADC1->SMPR1, (0x3FFFFFFF), (0x6<<3)); // Channel 1, Sampling Time: 247.5 ADC cycles
}

void startADC() {
    while (!(ADC1->ISR & ADC_ISR_ADRDY)) ADC1->CR |= ADC_CR_ADEN; // TURN ON
    delay_us(5000);
    ADC1->ISR = ADC_ISR_EOC | ADC_ISR_EOS | ADC_ISR_OVR; // Clear flags
    SET_BIT(ADC1->CR, ADC_CR_ADSTART); // START CONV
}

void light() {
    startADC();
    while (!(ADC1->ISR & ADC_ISR_EOC));
    adcVal = ADC1->DR;
}

```

References

- RM0351 Reference manual, STM32L4x6 advanced ARM®-based 32-bit MCUs
- STM32L476xx, Datasheet