

April, 2023 Semester

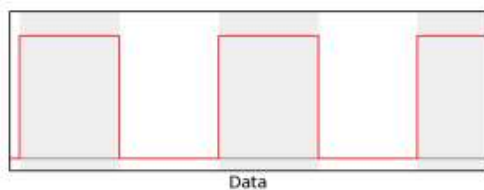
# ICT 6641 Advanced Embedded System Design

## Lecture#6: Application of Input Capture & ADC

S. M. Lutful Kabir, *PhD*  
Professor, BUET

### Motivation for Input Capture: Example 1: Wireless Transmission of Digital Data

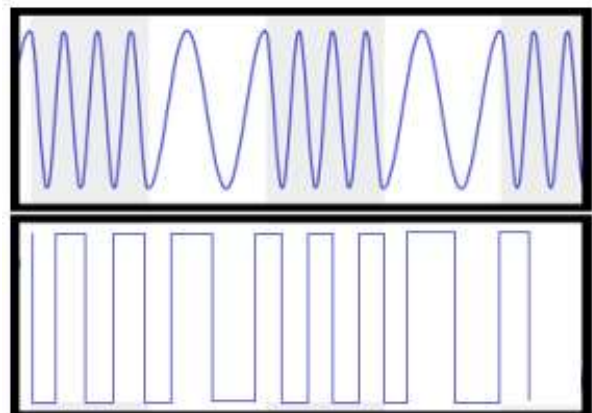
FSK: Transmission



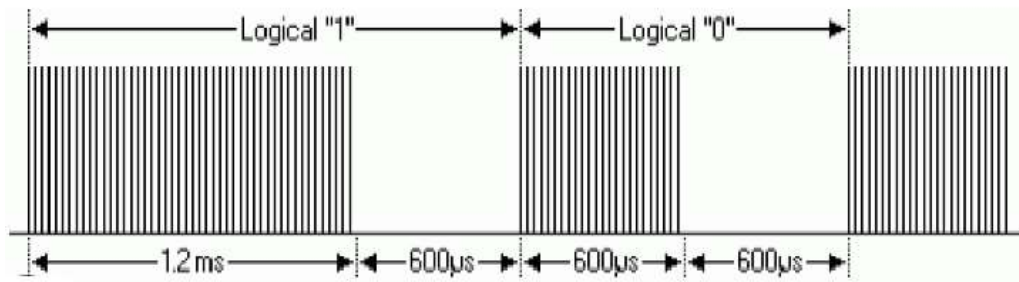
Data

Modulated Signal

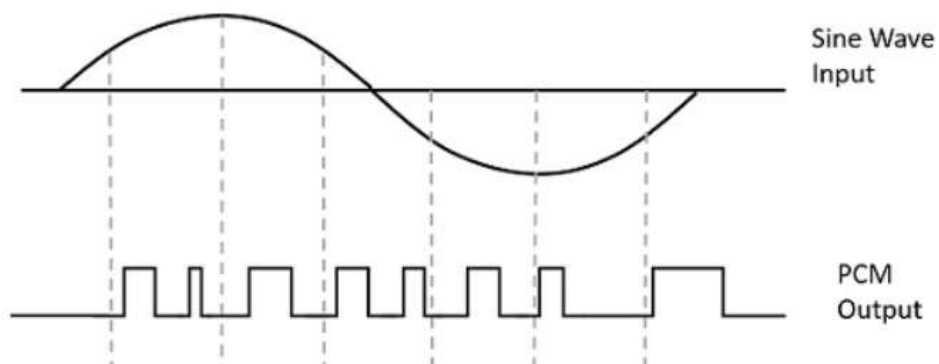
FSK: Receiving



## Signal from IR (remote)



## Pulse Coded Modulation (PCM)



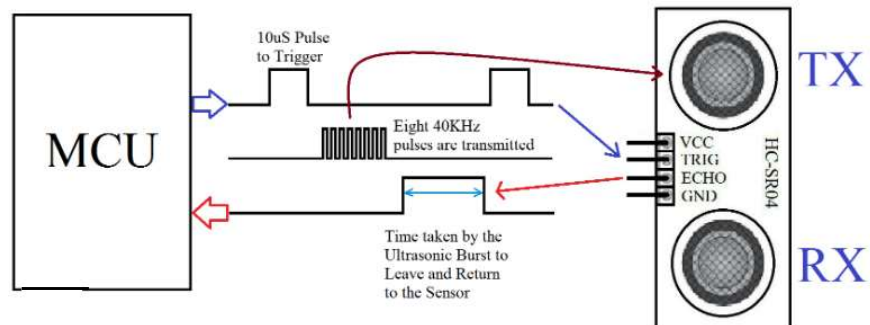
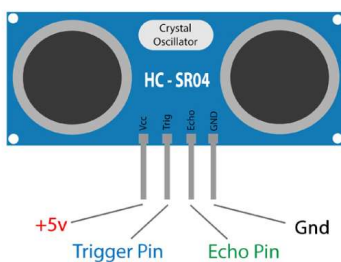
## Ultrasonic Sensor For Distance Measurement



- **Vcc** supplies power to the module.
- **Trigger** pin is used to trigger ultrasonic sound pulses. By setting this pin to HIGH for  $10\mu s$ , the sensor initiates an ultrasonic burst.
- **Echo** pin goes high when the ultrasonic burst is transmitted and remains high until the sensor receives an echo, after which it goes low.
- By measuring the time the Echo pin stays high, the distance can be calculated.
- **GND** is the ground pin.

## How Does Ultrasonic Distance Sensor Work?

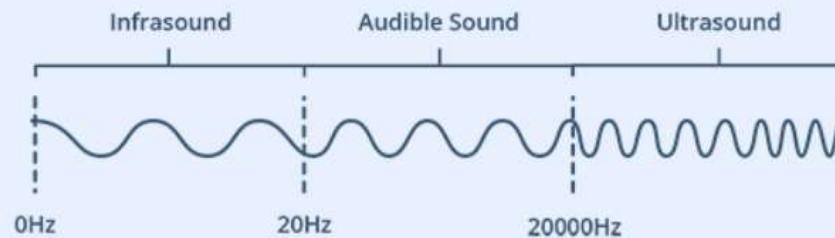
### HC-SR04 Pinout



- An Trigger pulse is sent to the trigger pin.
- The sensor generates, through its transmitter, 8 number of 40 kHz ultrasonic sound burst.
- On facing a obstruction the wave returns back and the receiver receives the signal.
- After receiving the signal, it generates a pulse, which comes out through the Echo pin.
- The width of the pulse is proportional to the distance the sound wave travelled.

## What is Ultrasound?

Ultrasound is a high-pitched sound wave whose frequency exceeds the audible range of human hearing.



Humans can hear sound waves that vibrate in the range of about 20 times a second (a deep rumbling noise) to 20,000 times a second (a high-pitched whistle). However, ultrasound has a frequency of more than 20,000 Hz and is therefore inaudible to humans.

## Features of the Ultrasonic Sensor

- An HC-SR04 ultrasonic distance sensor actually consists of two ultrasonic transducers.
- One acts as a transmitter that converts the electrical signal into 40 KHz ultrasonic sound pulses. The other acts as a receiver and listens for the transmitted pulses.
- When the receiver receives these pulses, it produces an output pulse whose width is proportional to the distance of the object in front.
- This sensor provides excellent non-contact range detection between 2 cm to 400 cm (~13 feet) with an accuracy of 3 mm.
- Since it operates on 5 volts, it can be connected directly to microcontroller pin if that can tolerate the level.

## Proposition of the Experiment

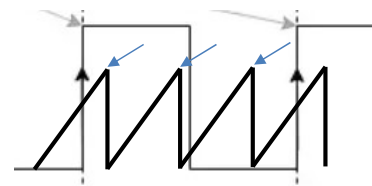
- We want to measure the distance of an object from the point of measurement.
- First of all we have to generate a 10 uS pulse and send this pulse to the trigger pin.
- We shall make a output pin (to be connected to the trigger pin) low and then make it high and maintain this high for 10uS or so. Finally making it low again, a pulse of 10uS will be generated.
- When the pulse through the Echo pin is received, we have to measure the width of the pulse.
- Using standard calculation, we can find out the distance.
- So, now the problem is how to measure the width of a pulse?

## Measurement of the Time Period of a Periodic wave

- In the last class we have learnt how to measure time period of a periodic signal.
- Let us recollect it: At the rising or falling edge at the starting and at the end of a period the value in the CCRx (Compare and Capture Register) is stored and also the number of overflow that might have occurred in between the two edges.
- From a calculation, we can accurately determine the time period of the periodic wave.
- The technique is depicted in the following figure.

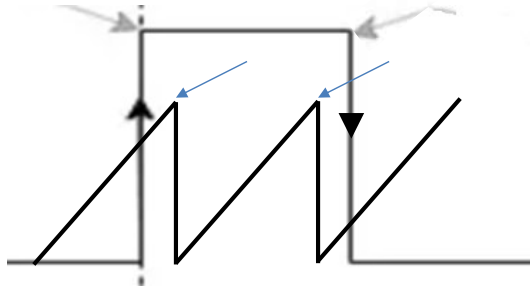
Say,  $n$  = number of overflow occurred within a Period

$$\text{Time Period} = (T_2 + nXARR - T_1)$$



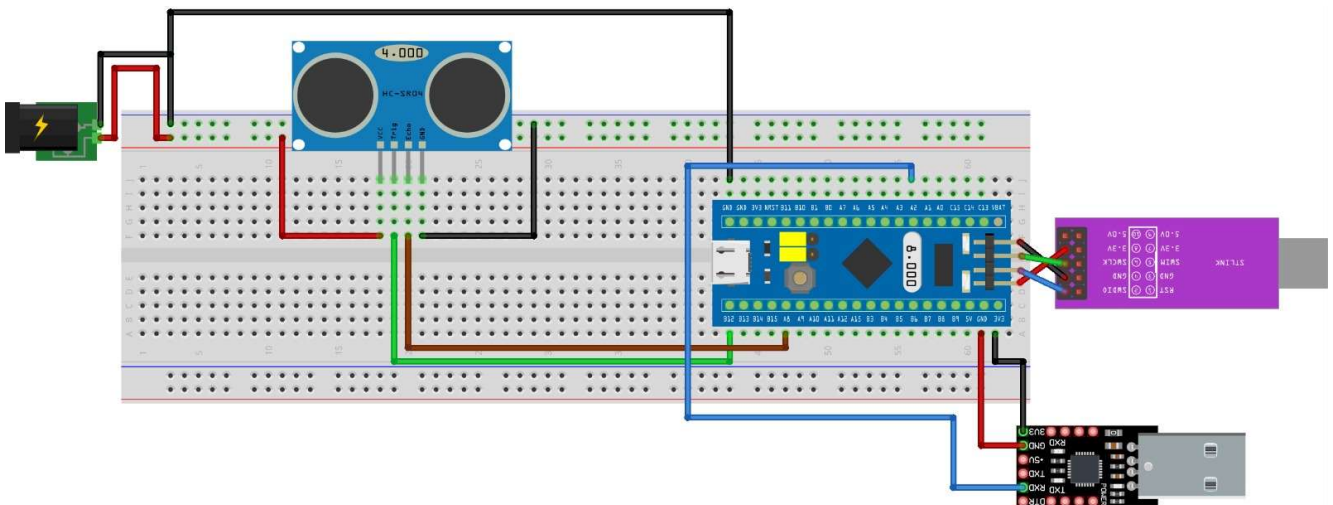



## Measurement of the width of a Pulse



- At the beginning, the trigger to the interrupt will be set at the rising edge.
- Just after crossing the first edge, the detection edge will be modified as falling edge.
- All other calculations will remain same.

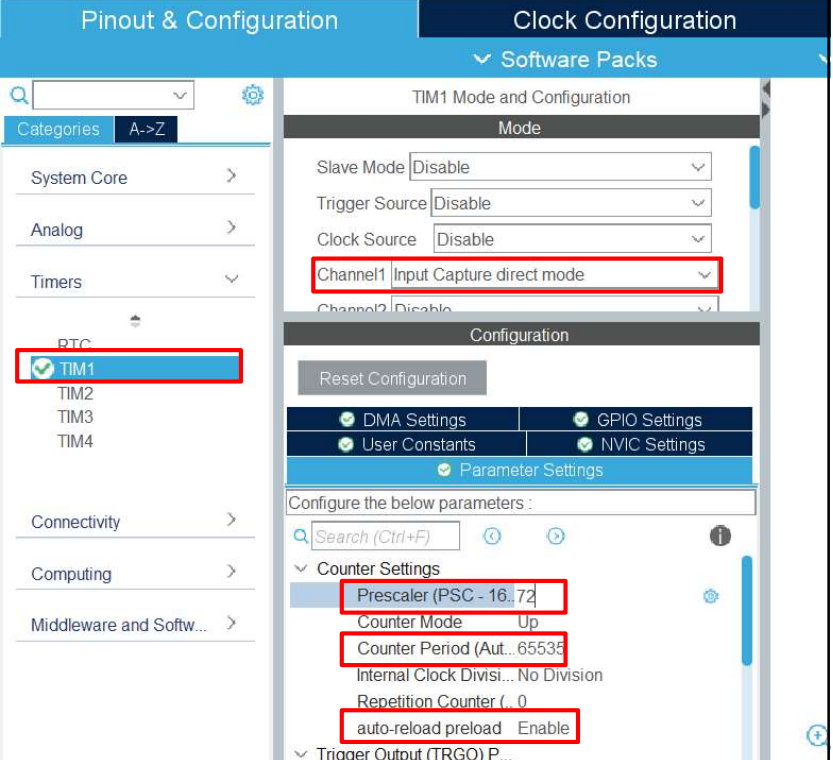
## Experimental Set Up






## Configuration of the Blue Pill Board for the Experiment

- Timer setting





## UART Setting, GPIO and Other Settings

- For serial print, UART2 is used for serial communication.
- The settings are: 9600 baud rate, 8-bit data, no parity, 1 stop bit
- Pin B12 is to be configured as output pin. This will be used as source of 10 uS pulse to be sent to the Trigger pin.
- Note that the Timer1 input pin is A8. This will as receiving the pulse for the distance.

## Addition of Codes

1

```
#include <stdio.h>
#define IDLE 0
#define DONE 1
#define F_CLK 72000000UL
#define Prescalar 72;
#define TRIG_PIN GPIO_PIN_12
#define TRIG_PORT GPIOB
```

2

```
#volatile uint8_t State = IDLE;
volatile uint32_t T1 = 0;
volatile uint32_t T2 = 0;
volatile uint32_t Ticks = 0;
volatile uint16_t TIM1_OVC = 0;
double Freq = 0;
unsigned char MSG[100];
float Distance = 0;
```

3

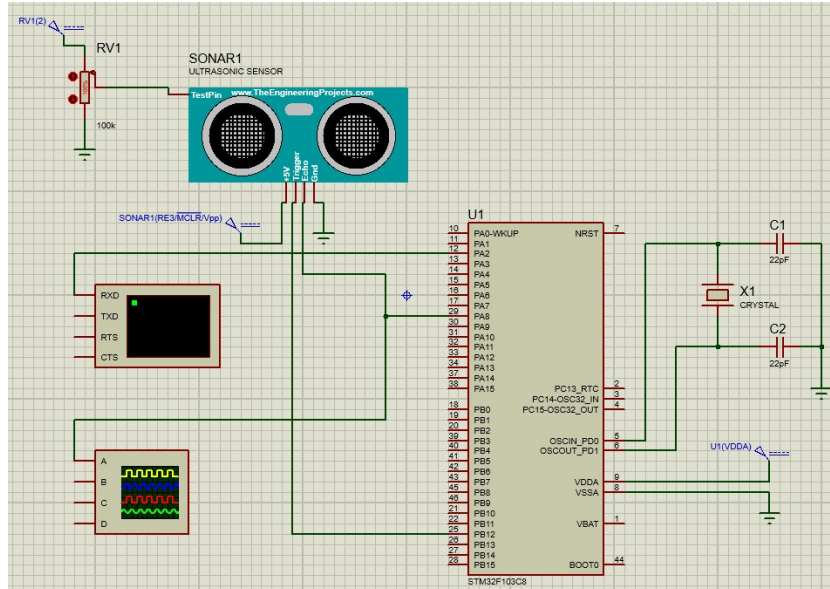
```
void HCSR04_Triger(void)
{
    HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_SET);
    __HAL_TIM_SET_COUNTER(&htim1, 0);
    // wait for 10 us
    while (__HAL_TIM_GET_COUNTER (&htim1) < 10);
    HAL_GPIO_WritePin(TRIG_PORT, TRIG_PIN, GPIO_PIN_RESET);
}
```

### 2nd Callback function

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef* htim)
{
    if(State==IDLE) // if the first edge is not captured
    {
        T1 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1); // read the first value
        TIM1_OVC = 0;
        State = DONE; // set the status as 1, first edge of the pulse is captured
        // Now change the polarity to falling edge
        __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_FALLING);
    }
    else if(State==DONE) // if the first has already already captured
    {
        T2 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1); // read the first value
        Ticks = T2 + (TIM1_OVC * 65536) - T1;
        State = IDLE; // set it back to 0, second edge of the pulse is captured
        // set polarity to rising edge
        __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_RISING);
        __HAL_TIM_DISABLE_IT(&htim1, TIM_IT_CC1);
    }
}
```



## Proteus Simulation

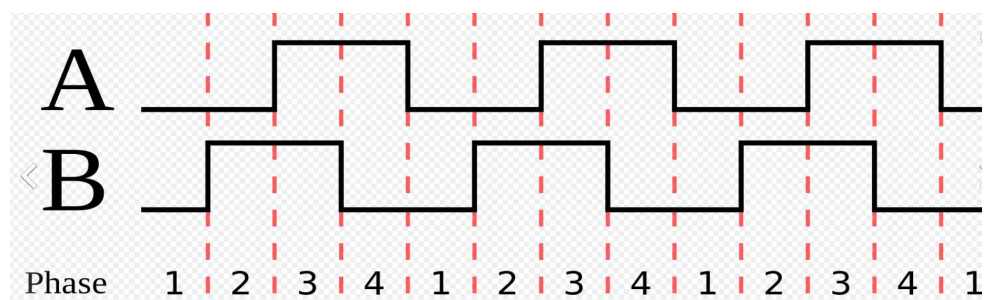


## Rotary Encoder: Incremental Type

## Incremental Encoder

- An **incremental encoder** is a linear or rotary electromechanical device that has two output signals, *A* and *B*, which issue pulses when the device is moved.
- Together, the *A* and *B* signals indicate both the occurrence of and direction of movement.
- Unlike an absolute encoder, an incremental encoder does not indicate absolute position.
- it only reports changes in position and, for each reported position change, the direction of movement.
- Incremental encoders report position changes nearly instantaneously, which allows them to monitor the movements of high speed mechanisms in near real-time.

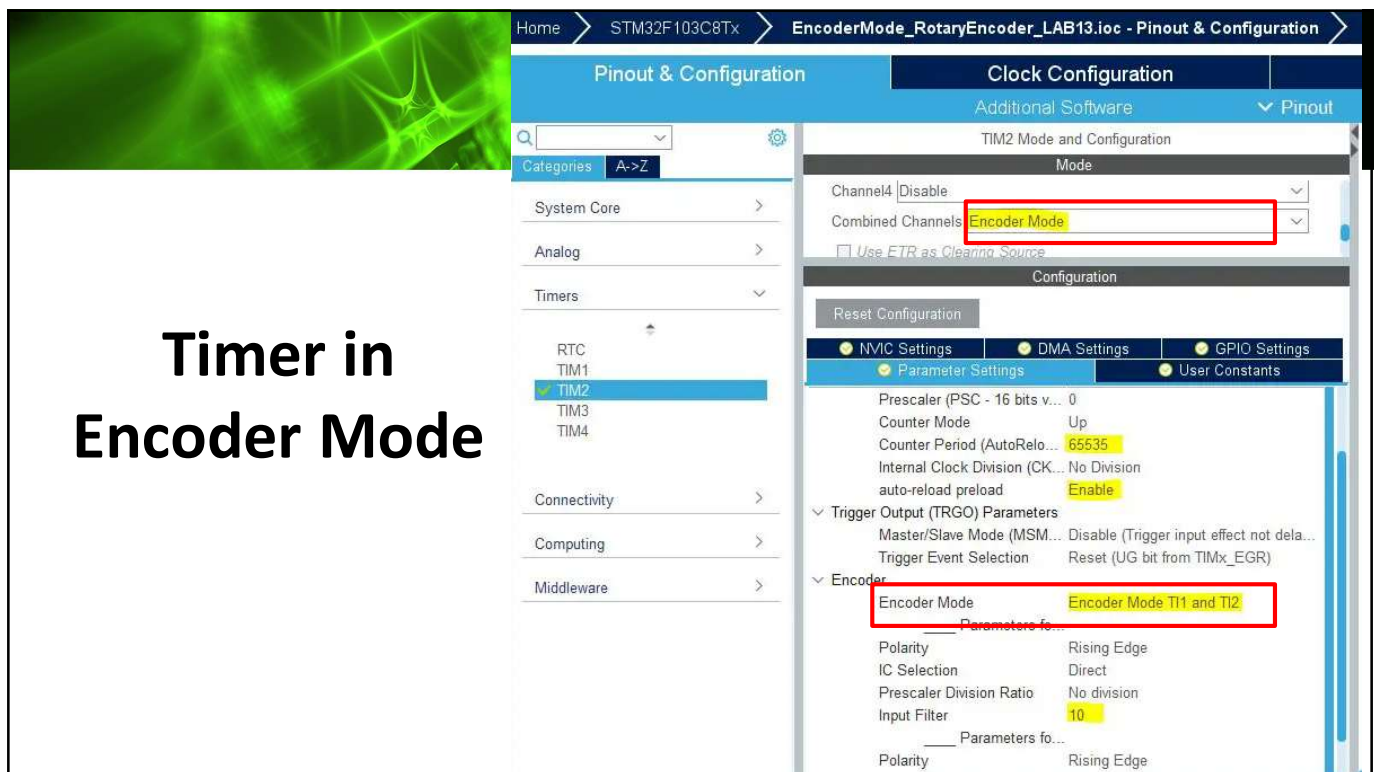
## Quadrature Phase Difference



For better understanding let us watch the animation in the link

<https://lastminuteengineers.b-cdn.net/wp-content/uploads/arduino/rotary-encoder-working-animation.gif>

# Timer in Encoder Mode



## Code for the Timer in Encoder Mode

```
#include "main.h"
TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
int main(void)
{
    uint16_t LED_DutyCycle = 0;
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM2_Init();
```

```
MX_TIM3_Init();
HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_ALL);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);

while (1)
{
    if(TIM2->CNT < 1023)
    {
        LED_DutyCycle = ((TIM2->CNT)<<6);
    }
    TIM3->CCR1 = LED_DutyCycle;
    HAL_Delay(10);
}
```



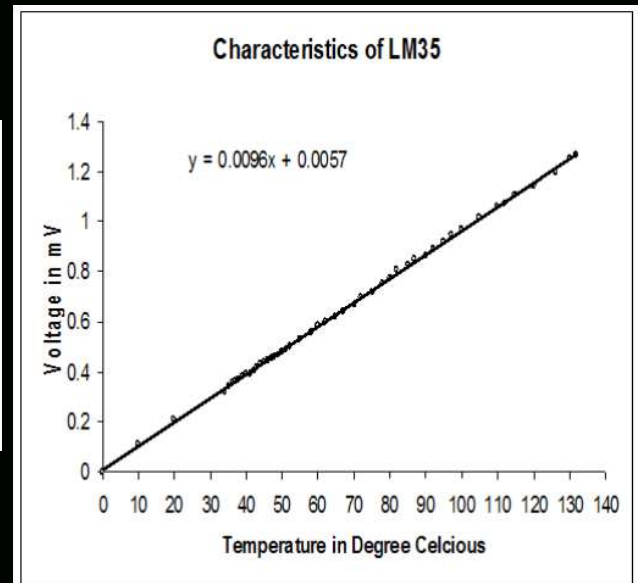
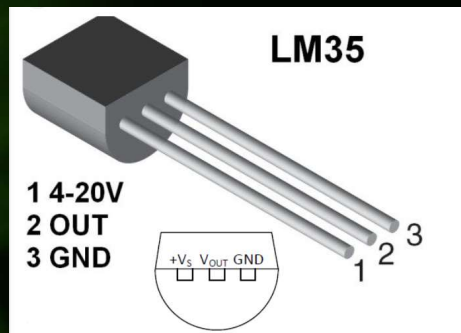
## Analog to Digital Conversion



### Why We Need Analog to Digital Converters?

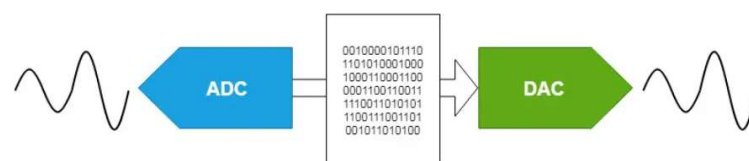
- The need of converting analog signals to digital data stems from the fact that our computers/ controllers are digital ones.
- They just can't handle the analog signal and therefore, there should be a device which converts the signal from analog to digital domain (ADC).
- Most signals are analog in nature and the electronic sensors which we're using for capturing these phenomena are also analog.
- For example, the temperature sensor converts temperature in °C to an analog voltage that's proportional to the value of temperature.
- So do the microphone, pressure sensor, light sensor and so on.
- Hence, we need a way to read analog voltage and convert it to digital values which we can program our computers/ controllers to manipulate it.

## A Temperature Sensor, LM35



## Analog-To-Digital Converters (ADC) Preface

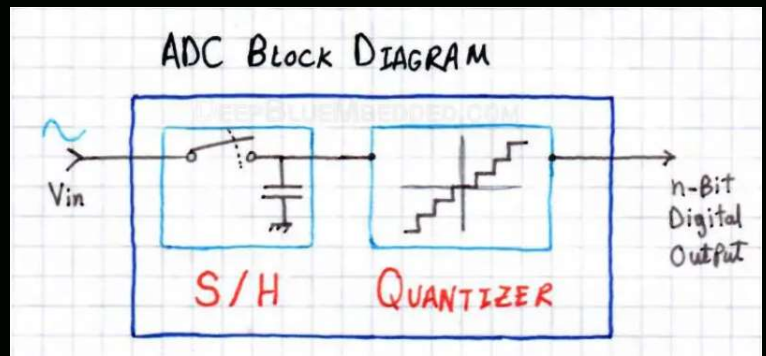
- An ADC (Analog-To-Digital) converter is an electronic circuit that takes in an analog voltage as input and converts it into digital data, a value that represents the voltage level in binary code.
- The ADC samples the analog input whenever you trigger it to start conversion.
- And it performs a process called quantization so as to decide on the voltage level and its binary code that gets pushed in the output register.
- The ADC does the counter operation that of a DAC, while an ADC (A/D) converts analog voltage to digital data the DAC (D/A) converts digital numbers to the analog voltage on the output pin.





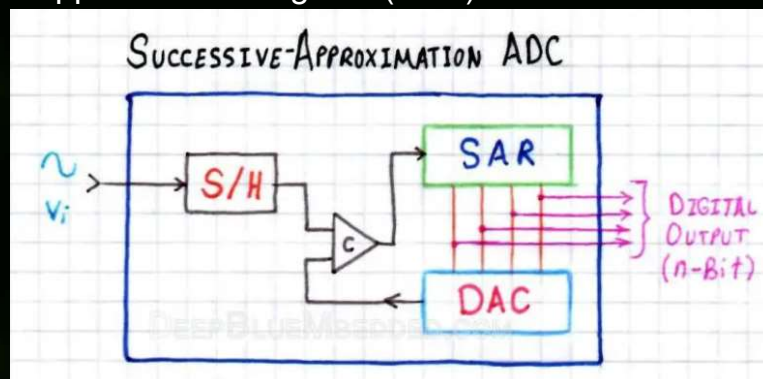
## How ADC Works?

- The basic structure of an ADC consists of an S/H circuit.
- Followed by a quantizer which is actually the working horse for the analog to the digital conversion process.
- The type of ADC depends on how it's performing the quantization process, it can be analog integration, digital counter, successive approximation, or even direct conversion as in Flash ADC types.
- Finally, the digital output data is served to the CPU or gets directly stored in memory.



## Successive Approximation ADC

- The Successive Approximation ADC uses a comparator to successively narrow a range that contains the analog input voltage.
- At each successive step, the converter compares the input voltage to the output of an internal DAC which might represent the midpoint of a selected voltage range.
- At each step in this process, the approximation is stored in the successive approximation register (SAR).



## ADC Resolution

- The Resolution of an ADC indicates the number of discrete values it can produce over the range of analog values.
- The resolution **Q** depends on both the number of bits "**n**" generated by the quantizer and also the **FSR** (full-scale range) for the analog reference voltage line.
- FSR indicates the range of voltage the ADC can convert.
- Here is the formula to calculate the resolution of quantization **Q** which is basically a division for the FSR voltage by the number of levels  $2^n$ .
- Hence,  
$$\text{Resolution} = \text{FSR} / 2^n$$
- For a 12-bit ADC and FSR=3.3V, Resolution =  $3.3\text{V} / 2^{12} = 0.8057\text{mV}$

## STM32 ADC Brief

- The STM32F103C8 (Blue Pill) has a 12-bit ADC which is a successive approximation analog-to-digital converter.
- It has up to 18 multiplexed channels allowing it to measure signals from sixteen external and two internal sources.
- The result of the ADC is stored in a left-aligned or right-aligned 16-bit data register.
- The ADC input clock is generated from the PCLK2 clock divided by a Prescaler and it must not exceed 14 MHz.

## ADC Features

- 12-bit resolution
- Interrupt generation at End of Conversion, End of Injected conversion and Analog watchdog event
- Single and continuous conversion modes
- Scan mode for automatic conversion of channel 0 to channel 'n'
- Self-calibration
- Data alignment with in-built data coherency
- Channel by channel programmable sampling time.
- External trigger option for both regular and injected conversion
- Discontinuous mode
- Dual-mode (on devices with 2 ADCs or more)
- ADC conversion time: 1  $\mu$ s at 56 MHz (1.17  $\mu$ s at 72 MHz)
- ADC supply requirement: 2.4 V to 3.6 V
- ADC input range:  $V_{REF-} \leq V_{IN} \leq V_{REF+}$
- DMA request generation during regular channel conversion.

## STM32 ADC Read Methods

### 1 Polling

- In the polling method, we start an ADC conversion and stop the CPU at this point to wait for the ADC conversion completion.

### 2 Interrupts

- We can trigger the ADC in order to start a conversion and the CPU continues executing the main code routine. Upon conversion completion, the ADC fires an interrupt.
- Despite being an efficient way, the interrupt method can add so much overhead to the CPU and cause very high CPU loading. Especially when you're doing so many conversions per second.

### 3 DMA

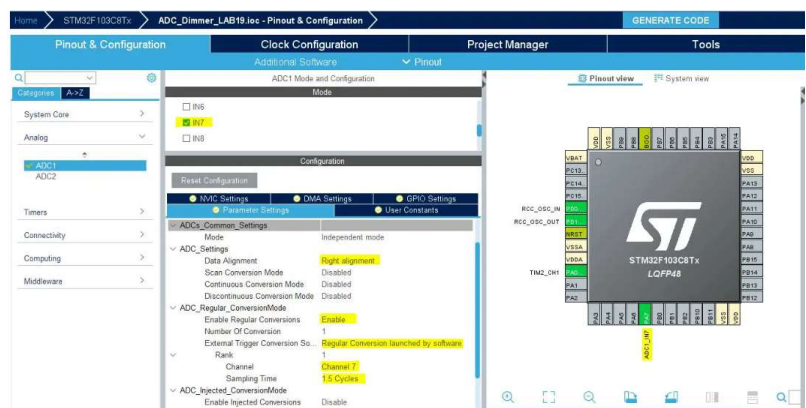
- The third method is using the DMA unit that can directly transfer the ADC result from the peripheral to the memory without any CPU intervention.

## An Example of Polling Method

- The objective of the first exercise is to control the brightness of an LED in accordance with an input.
- The input is a voltage across a potentiometer.
- The output is a PWM signal to be applied across the LED.
- We have to build a system that initializes the ADC with an analog input pin.
- And also configure a timer module to operate in PWM mode with output on the pin (LED pin).
- Therefore, we can start an ADC conversion and map the result to the PWM duty cycle and repeat the whole process over and over again.

## ADC Settings

- Open STM32CubeIDE.
- Choose The Target MCU & Save with a Project Name (maybe ADC\_POLL)
- Configure The ADC1 Peripheral, Enable Channel7 & Set it to be triggered by software. (as shown, these are default configurations which happens to be ok for us).
- The settings are for Data Alignment, Enable Regular Conversion, External Trigger Conversion Source. Channel and Sampling time.



- 

# Clock Setting

- [illegible]



## ADC Polling

## Code Generation and Addition of the following Portions

- **Add the following lines at the beginning**  

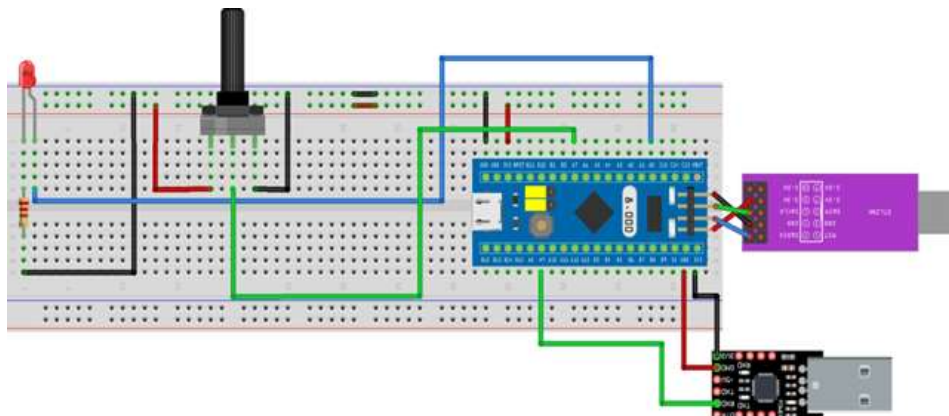
```
#include <stdio.h>
uint16_t AD_RES = 0;
uint8_t MSG[35] = {'\0'};
uint8_t X=0;
```
- **Before while statement, start the Timer and the ADC as shown below**  

```
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
// Calibrate The ADC On Power-Up For Better Accuracy
HAL_ADCEx_Calibration_Start(&hadc1);
```
- **while loop**  

```
while (1)
{
    // Start ADC Conversion
    HAL_ADC_Start(&hadc1);
    // Poll ADC1 Peripheral
    HAL_ADC_PollForConversion(&hadc1, 1);
    // Read The ADC Conversion Result
    // & Map It To PWM DutyCycle
    AD_RES = HAL_ADC_GetValue(&hadc1);
    TIM2->CCR1 = (AD_RES<<4);
    X=(int)((TIM2->CCR1)*100/65535));
    sprintf(MSG, "Duty Cycle (percent) = %d\r\n", X);
    HAL_UART_Transmit(&huart1, MSG, sizeof(MSG), 100);
    HAL_Delay(1);
}
```

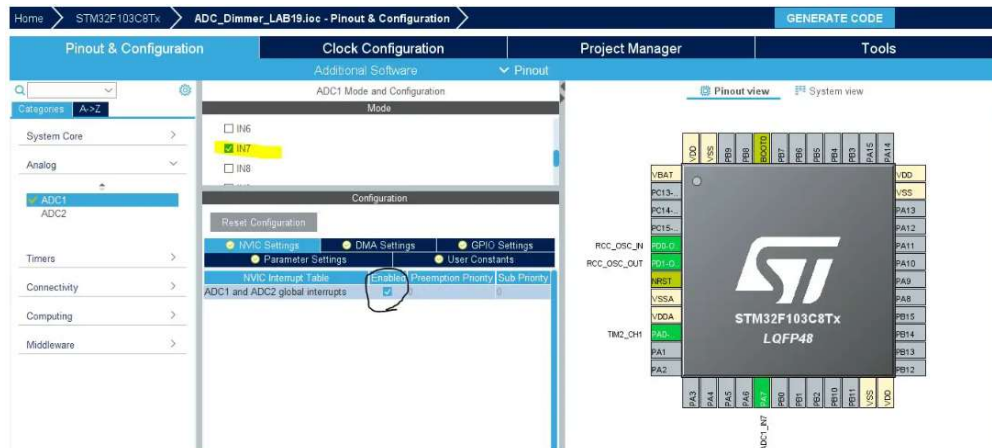
## Circuit Connection

- Establish the circuit shown below:



# STM32 ADC Interrupt Example

- The Exact Same Steps As The Previous Example. Set the ADC Configuration as below.
- All ADC settings will remain the same but we'll need to enable the interrupt from the NVIC controller tab.



## ADC Interrupt

## Code Generation and Change in the previous addition

- Add the following at the beginning
 

```
#include <stdio.h>
uint16_t AD_RES = 0;
uint8_t MSG[35] = {'\0'};
uint8_t X = 0;
```
- Before while statement, start the Timer and the ADC as shown below
 

```
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
// Calibrate The ADC On Power-Up For Better Accuracy
HAL_ADCEx_Calibration_Start(&hadc1);
```

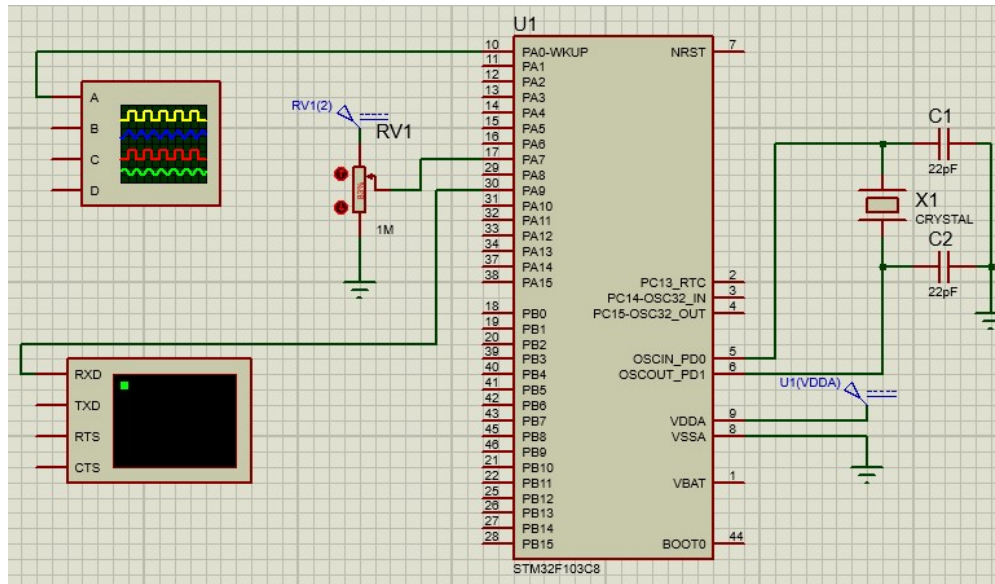
### • while loop

```
while (1) {
    // Start ADC Conversion
    // Pass (The ADC Instance, Result Buffer Address, Buffer Length)
    HAL_ADC_Start_IT(&hadc1, &AD_RES, 1);
    TIM2->CCR1 = (AD_RES<<4);
    X = (int)((TIM2->CCR1)*100.0/65535.0);
    sprintf(MSG, "Duty Cycle = %d\r\n", X);
    HAL_UART_Transmit(&huart1, MSG, sizeof(MSG), 100);
    HAL_Delay(1);
}
```

### • Callback function [after main() function]

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    // Read & Update The ADC Result
    AD_RES = HAL_ADC_GetValue(&hadc1);
}
```

## Proteus Simulation



# Thanks