Fienga Luigi [M63001733] – Roscigno Andrea [M63001778] – Savarese Serena [M63001855]

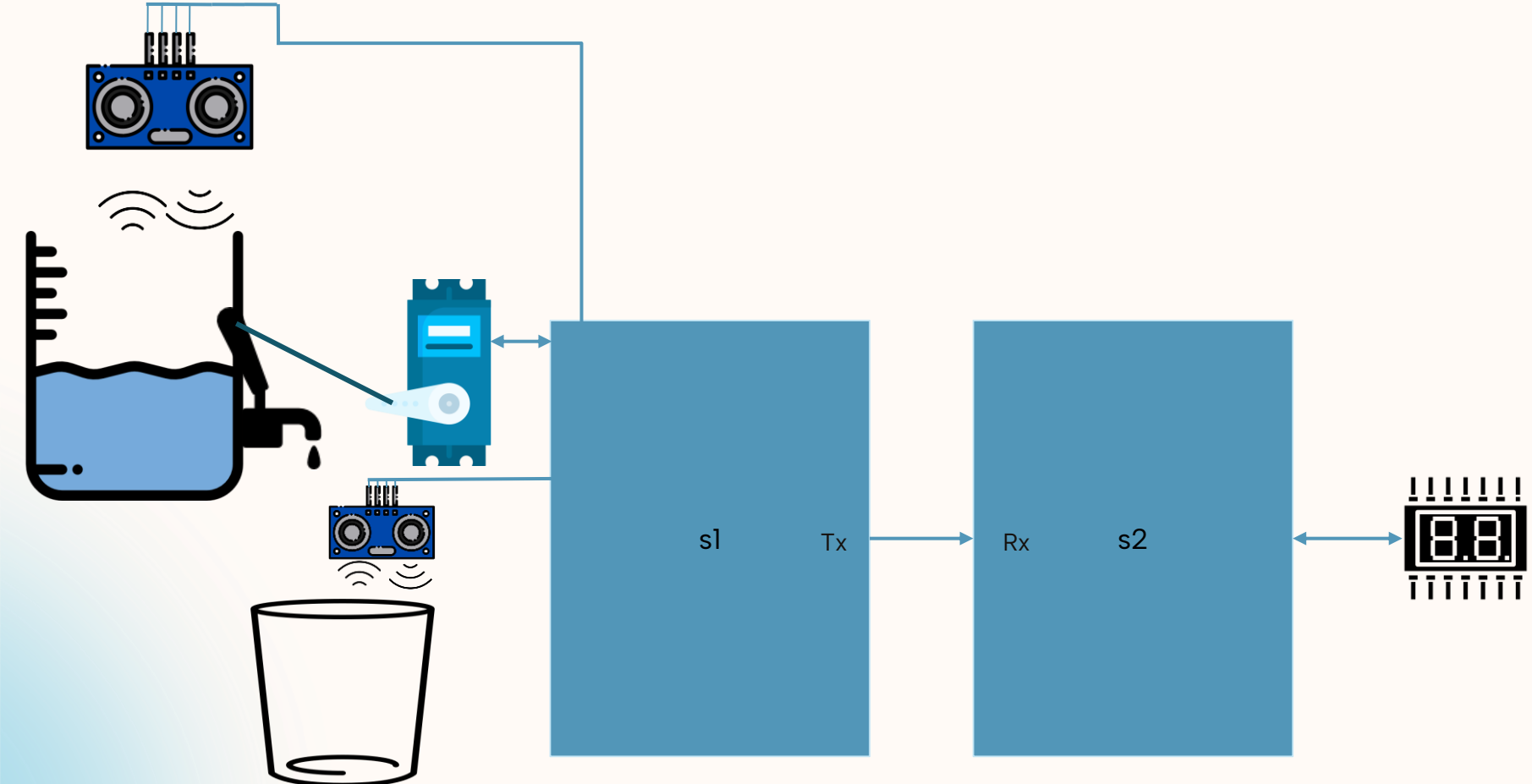
Automatic Water Dispenser

Descrizione e obiettivi del progetto

Il nostro obiettivo è quello di voler realizzare un dispenser automatico d’acqua. Il funzionamento è basato sulla comunicazione tra due sistemi e consiste nel:

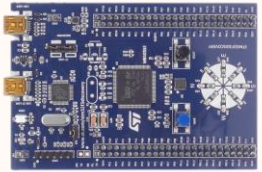
- Fornire automaticamente dell’acqua da un dispenser, facendo aprire un rubinetto mediante un servomotore, attivato tramite il superamento di una soglia, che viene catturata da un sensore di prossimità rispetto al fondo di un recipiente. Entrambi i sensori fanno parte del primo sistema.
- Viene controllato il livello dell’acqua nel dispenser mediante un secondo sensore di prossimità i quali valori vengono mandati dal sistema uno al sistema due. I dati ricevuti sono usati per mostrare su uno schermo OLED il livello dell’acqua, che può essere: alto, medio o basso.
- Il servomotore chiude infine il rubinetto se il primo sensore di prossimità raggiunge una seconda soglia dal recipiente, ad indicare la fine del riempimento di quest’ultimo.

Rappresentazione schematica

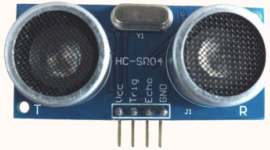


Componenti utilizzati

01 Board
STM32F303Discovery x 2

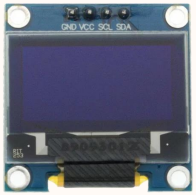


02 Sensore di prossimità
HC-SR04 x 2

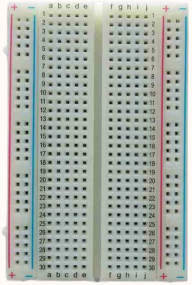


03 Servomotore
Micro servo SG90

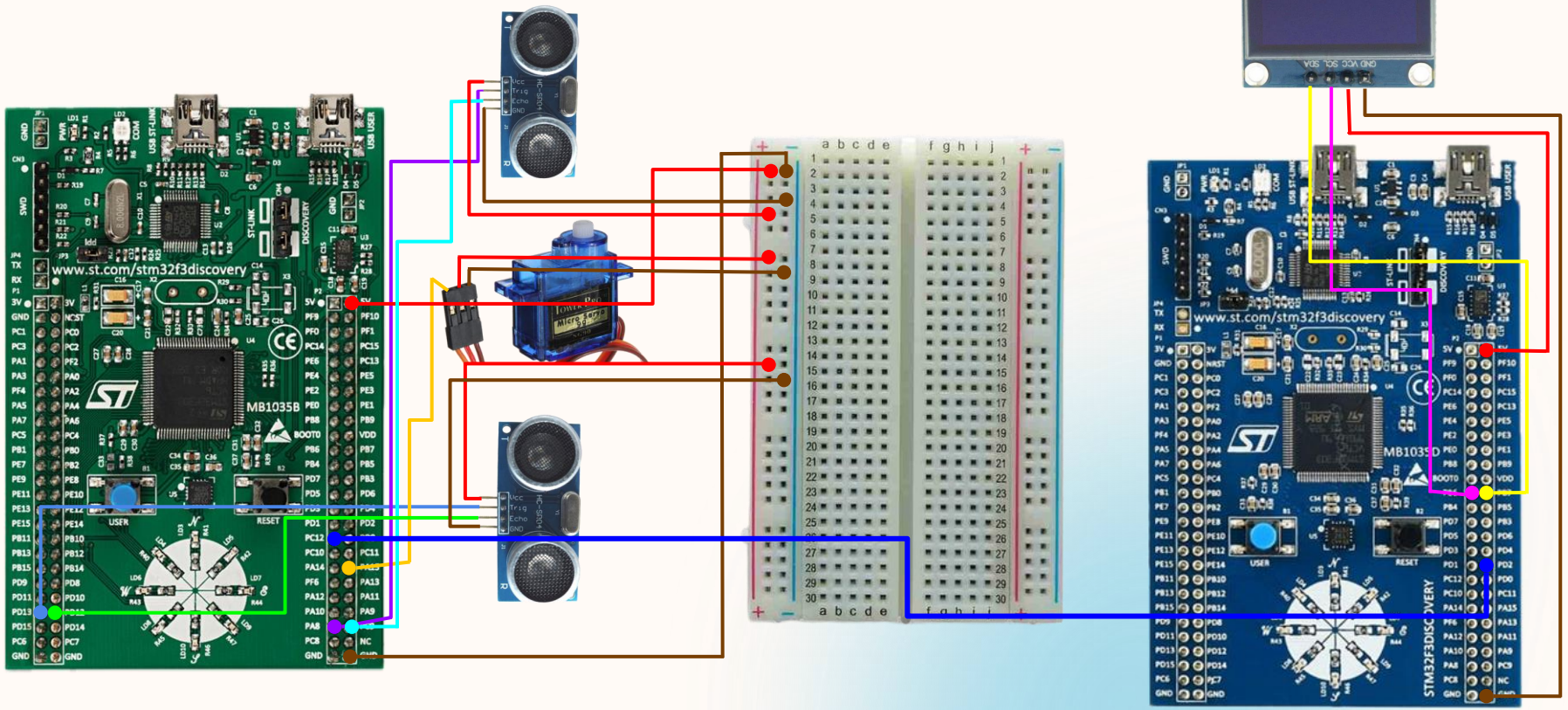
04 Display OLED
ssd1306



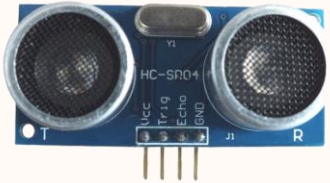
05 Breadboard



Collegamenti Elettronici



Sensore di prossimità HC-SR04

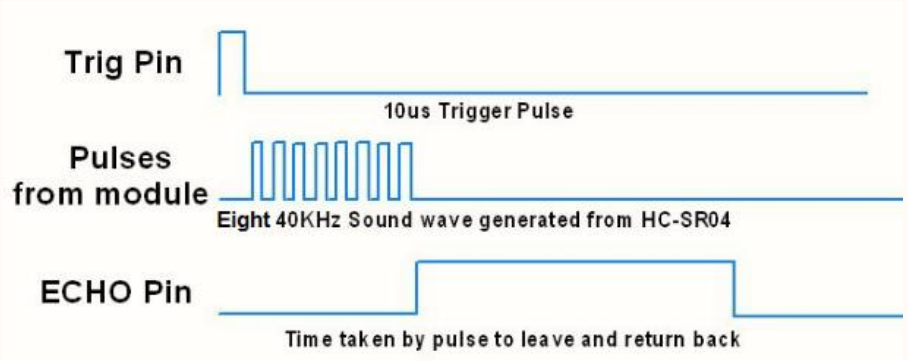


Il sensore di prossimità **HC-SR04** è un sensore a ultrasuoni usato per misurare distanze tra il sensore stesso e un oggetto davanti a lui. I pin del sensore sono:

- Trigger pin di input, che serve per iniziare il processo di misura.
- Echo pin di output, restituisce un segnale alto per una durata proporzionale alla distanza del sensore e ciò che ha davanti (a riposo è LOW).
- VCC, alimentazione a 5V.
- GND, massa (ground)

Il microcontrollore invia un segnale HIGH di 10 microsecondi al pin TRIG per iniziare il processo di misurazione.
Il sensore emette un ultrasuono (onda sonora a 40 kHz) in avanti per 8 volte.
Dopo aver mandato la raffica di ultrasuoni il pin ECHO dall'essere LOW diventa HIGH.
La raffica viaggiando nell'aria quando troverà un oggetto rimbalzerà dal medesimo e tornerà al sensore.
Quando il sensore riceve tutta la raffica, il pin ECHO diventa LOW, in base alla durata con cui il segnale ECHO è stato alto, è possibile misurare la distanza dal sensore e l'oggetto individuato, con la seguente formula:

$$\Delta s = (\Delta t * 343\ (m/s)) / 2$$



Dove:

- 343 m/s è la velocità del suono nell'aria (che è uguale a quella degli ultrasuoni)
- Δt è la durata del segnale ECHO quando è HIGH, in µs
- La divisione per 2 viene fatta perchè la raffica mandata del sensore compie due viaggi, uno dal sensore verso l'oggetto e uno dall'oggetto verso il sensore, mentre la distanza effettiva è di un solo viaggio.
- Δs è la distanza misurata, in micron

Sensore di prossimità HC-SR04

Per quanto riguarda la corretta integrazione del sensore sul STM32F303, abbiamo realizzato la libreria *"HCSR04.h"*. La necessità nella creazione della libreria nasce perchè dovendo utilizzare due sensori sulla stessa scheda ci servivano delle funzioni che fossero capaci di astrarre e generalizzare il loro comportamento.

```c
#ifndef SRC_HCSR04_H_
#define SRC_HCSR04_H_

#include "stm32f3xx_hal.h"

typedef struct hcsr04_t {

    uint32_t ic_val1;
    uint32_t ic_val2;
    uint8_t is_first_capture;
    uint32_t difference;
    uint32_t distance_cm;

}hcsr04_t;


void HCSR04_Trigger(GPIO_TypeDef * GPIO, uint16_t PIN, TIM_HandleTypeDef * htim);
void delay_us(uint16_t us, TIM_HandleTypeDef * htim);



#endif /* SRC_HCR04_H_ */
```

La struct *hcsr04_t* contiene rispettivamente in ordine di comparsa: la prima lettura temporale (quando si alza ECHO), la seconda lettura temporale (quando si abbassa ECHO), variabile di controllo se la prima lettura è stata fatta, la differenza tra le due lettura e la distanza espressa in cm.

Sensore di prossimità HC-SR04

Per quanto riguarda l'implementazione delle due firme di funzione viste prima:

- *HCSR04_Trigger*, noto il pin su cui mandare il segnale di trigger e noto il timer da usare, scrive sul pin un valore logico HIGH per un tempo pari a 10 µs grazie alla funzione *delay_us*, per poi settarlo nuovamente ad un valore logico LOW
- *delay_us*, noto il tempo e il timer, resetta quest'ultimo come valore di conteggio e rimane bloccato all'interno della funzione in base al tempo fornito

```
void delay_us(uint16_t us, TIM_HandleTypeDef * htim) {
    __HAL_TIM_SET_COUNTER(htim, 0);  // reset counter
    while (__HAL_TIM_GET_COUNTER(htim) < us);
}

void HCSR04_Trigger(GPIO_TypeDef * GPIO, uint16_t PIN, TIM_HandleTypeDef * htim) {
    HAL_GPIO_WritePin(GPIO, PIN, GPIO_PIN_SET);
    delay_us(10, htim);                                    // 10 us
    HAL_GPIO_WritePin(GPIO, PIN, GPIO_PIN_RESET);
}
```

Per il corretto conteggio dei 10 µs, i due timer dei due sensori devono avere un valore di prescaler pari a 47, in quanto il clock di sistema lavora a 48MHz e noi vogliamo che conti un valore ogni µs. La frequenza con cui vorremmo che operasse il timer è 1MHz, di conseguenza seguendo la formula, il valore del Prescaler è ovvio:

Prescaler = ((Frequenza del Clock)/(Frequenza desiderata)) – 1

Oltre a modificare il Prescaler nella configurazione del timer bisogna specificare sul canale a cui è collegato il pin di echo la modalità "Input Capture Direct Mode" e attivare l'interruzione corrispettiva del timer sulla linea del NVIC. Questo serve perchè se su quel pin il timer nota una variazione nel passaggio L->H o H->L (in base a com'è configurato di default Rising Edge), solleva l'interruzione e qui bisogna gestire la logica per effettuare correttamente la misura della distanza.

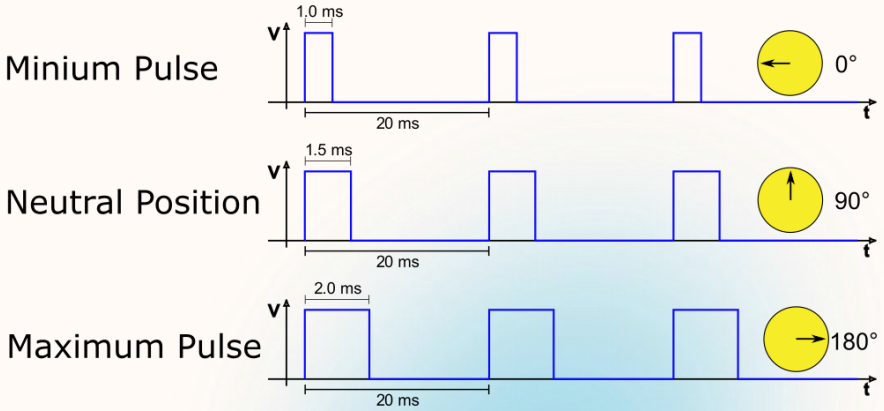
Sensore di prossimità HC-SR04

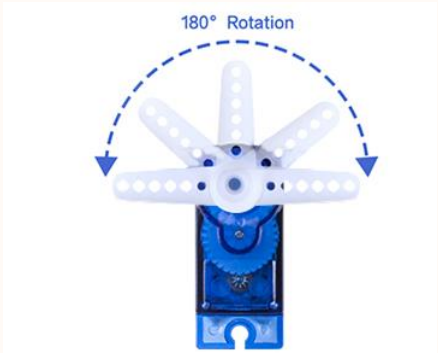
```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {

  if (htim == &htim3){
     if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_4) {
            if (s1.is_first_capture == 0) {
                s1.ic_val1 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_4);
                s1.is_first_capture = 1;

                __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_4, TIM_INPUTCHANNELPOLARITY_FALLING);
            } else {
                s1.ic_val2 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_4);
                __HAL_TIM_SET_COUNTER(htim, 0);

                if (s1.ic_val2 > s1.ic_val1)
                    s1.difference = s1.ic_val2 - s1.ic_val1;
                else
                    s1.difference = (0xFFFF - s1.ic_val1) + s1.ic_val2;

                s1.distance_cm = (s1.difference * 343) / 20000;

                s1.is_first_capture = 0;
                __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_4, TIM_INPUTCHANNELPOLARITY_RISING);
            }
        }
  }


  if (htim == &htim4){
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1) {
                if (s2.is_first_capture == 0) {
                    s2.ic_val1 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
                    s2.is_first_capture = 1;

                    __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_FALLING);
                } else {
                    s2.ic_val2 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
                    __HAL_TIM_SET_COUNTER(htim, 0);

                    if (s2.ic_val2 > s2.ic_val1)
                        s2.difference = s2.ic_val2 - s2.ic_val1;
                    else
                        s2.difference = (0xFFFF - s2.ic_val1) + s2.ic_val2;

                    s2.distance_cm = (s2.difference * 343) / 20000;

                    s2.is_first_capture = 0;
                    __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1, TIM_INPUTCHANNELPOLARITY_RISING);
                }
            }
  }
}
```

Questa a sinistra è l'interruzione per gestire le commutazioni del segnale echo e quindi calcolare la distanza.
Per il primo sensore si usa il *timer3_ch4* e per il secondo sensore il *timer4_ch1*. Il comportamento delle due ISR è identico, si differenziano solo perchè la prima lavora sulla struct s1 e il secondo con s2.
Si inizia controllando se la variabile *is_firs_capture* è pari a 0, se sì, significa che la commutazione è L->H, quindi viene salvato il primo valore di lettura temporale, si aggiorna *is_first_capture* con 1 e viene cambiata la polarità con cui l'interruzione si deve sollevare nel timer mettendo una transizione H->L.
Nel caso di tale commutazione l'interruzione si rialza e cade nel ramo dell'else dove cattura la seconda lettura, resetta il timer; si controlla se val2>val1 per calcolare la differenza giusta e infine si calcola la distanza in cm.
Sulla formula usata notiamo che è leggermente diversa rispetto a quella di prima, in quanto la differenza è in µs, la velocità è in m/s e vogliamo convertirla in cm/µs, per fare ciò si divide per un fattore 10000. Dopo il calcolo della distanza si resetta il *is_first_capture* e si inverte la polarità del timer per essere sensibile alle transizioni L->H.

Servomotore Micro servo SG90

Il **SG90** è un servomotore a rotazione limitata a max 180°. Esso contiene al suo interno un piccolo motore DC, un sistema di ingranaggi e un potenziometro collegato all'albero dell'uscita. Il servo SG90 riceve un segnale PWM (Pulse Width Modulation), ovvero un'onda quadra con duty cycle variabile con un periodo di 20 ms (50Hz), che ne controlla la posizione.
Un potenziometro interno permette al motore di sapere dove si trova, e il circuito interno regola automaticamente il movimento per raggiungere e mantenere l'angolo richiesto. Il circuito interno del servo confronta continuamente la posizione attuale, letta dal potenziometro, con quella desiderata, derivata dal PWM in ingresso. Se la posizione reale è diversa da quella target, il circuito attiva il motore nella direzione necessaria, finché i due valori coincidono. Non gira liberamente come un motore normale, ma si ferma esattamente dove richiesto dal segnale. Inoltre, mantiene attivamente quella posizione applicando coppia se necessario, ad esempio contro un carico esterno. È per questo che anche quando è fermo, può consumare corrente se deve opporsi a una forza che cerca di spostarlo.





Servomotore Micro servo SG90

Per il corretto funzionamento del servomotore con la scheda STM32F300, è stato necessario, in primis, configurare correttamente un timer di sistema soddisfacendo i requisiti del servo.
Il timer deve avere per un canale attiva la generazione di un segnale PWM a periodo 20ms (50Hz), e duty cycle variabile in base all'angolo di apertura del servo.
Introduciamo queste tre formule:

(1) Ttick = 1 / [Clock / (Prescaler + 1)]

(2) Tpwm = (ARR + 1) * Ttick

(3) Thigh = CCR * Ttick

- La prima permette di conoscere il periodo di conteggio del timer, noto il Clock e noto il Prescaler.
- La seconda relazione lega il periodo del PWM con l'ARR (registro che indica il conteggio max del timer) e il periodo di conteggio.
- La terza esprime il tempo con cui la finestra quadrata è alta in un periodo; essa dipende dal contenuto del registro CCR, interno al timer, e dal periodo di conteggio di un tick.

Il valore CCR viene calcolato dal microcontrollore in base all'angolo che si vuole ottenere. Considerando il range dei possibili CCR che va da 210 a 1050, l'ampiezza 0° (1.0 ms) è associata a 210, e 180° (2.0 ms) a 1050. Si deve cercare i valori di: Ttick, ARR e Prescaler che soddisfino il range di valori precedenti.
Risolvendo il problema di natura matematica, ipotizzando che il clock sia 48MHz, il Prescaler deve valere 95 e ARR 9999, di conseguenza il Ttick vale circa 2 µs

Servomotore Micro servo SG90

Oltre alla configurazione del timer, è stata implementata la funzione *Set_Servo_Angle* per poter calcolare il valore *pulse_length,* sulla base dell'angolo fornito, e andare a modificare il contenuto del registro CCR del timer con la pulsazione voluta. Si parte sempre da un incremento di 210 per rientrare in un range valido.

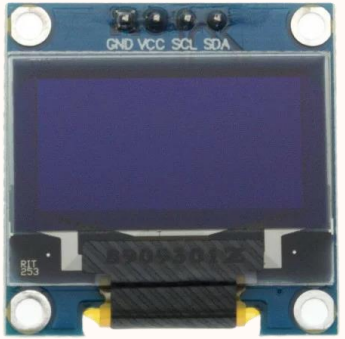
```
void Set_Servo_Angle(TIM_HandleTypeDef *htim, uint32_t channel, uint8_t angle) {

  uint32_t pulse_length = 210 + (angle * (1050 - 210)/180);
  __HAL_TIM_SET_COMPARE(htim, channel, pulse_length);
}
```

Questo di seguito è un esempio su come muovere il servomotore da un angolo di partenza (0°) ad uno finale (180°).
È implementato con il ciclo for in quanto la PWM da generare deve avere un duty cycle variabile con iterazioni di ogni 5°

```
for (uint8_t angle = theta0; angle <= thetaf; angle += 5) {
  Set_Servo_Angle(&htim2, TIM_CHANNEL_1, angle);
  HAL_Delay(50);
}
Set_Servo_Angle(&htim2, TIM_CHANNEL_1, thetaf);
```

Display OLED ssd1306

Il display OLED con controller **SSD1306** è un modulo grafico a matrice di punti. Il controller SSD1306 integrato nel modulo gestisce una griglia di 128x64 pixel monocromatici e ognuno può essere acceso o spento individualmente.
Il controller include sia una memoria RAM interna dedicata alla rappresentazione grafica, sia un sistema di indirizzamento dei pixel organizzato in una struttura a pagine.
Nel funzionamento in modalità I²C, il display si comporta come uno slave con indirizzo fisso. Ogni comunicazione I²C è composta da un byte di controllo iniziale, che specifica se i byte successivi sono comandi o dati, seguito da una sequenza di byte che verranno interpretati dal controller.
il display comunica con il microcontrollore usando solo due linee: SDA (Serial Data), trasporta i dati e i comandi; SCL (Serial Clock),fornisce il segnale di clock per sincronizzare la trasmissione.
Questi byte vengono memorizzati nella RAM interna del display, strutturata in pagine da 8 righe ciascuna.
Ogni byte scritto controlla 8 pixel verticali di una pagina. L'intero contenuto grafico del display è quindi definito da una mappa di byte, che può essere costruita in RAM della scheda e poi trasferita via SDA, sincronizzata dal clock SCL.



Display OLED ssd1306

per l'utilizzo del display , abbiamo utilizzato la libreria fornita da *GitHub - afiskon/stm32-ssd1306: STM32 library for working with OLEDs based on SSD1306, SH1106, SH1107 and SSD1309, supports I2C and SPI*. Di seguito le primitive principali fornite per poter scrivere e lavorare a piacimento sul display.

```
// Procedure definitions
void ssd1306_Init(void);
void ssd1306_Fill(SSD1306_COLOR color);
void ssd1306_UpdateScreen(void);
void ssd1306_DrawPixel(uint8_t x, uint8_t y, SSD1306_COLOR color);
char ssd1306_WriteChar(char ch, SSD1306_Font_t Font, SSD1306_COLOR color);
char ssd1306_WriteString(char* str, SSD1306_Font_t Font, SSD1306_COLOR color);
void ssd1306_SetCursor(uint8_t x, uint8_t y);
void ssd1306_Line(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color);
void ssd1306_DrawArc(uint8_t x, uint8_t y, uint8_t radius, uint16_t start_angle, uint16_t sweep, SSD1306_COLOR color);
void ssd1306_DrawArcWithRadiusLine(uint8_t x, uint8_t y, uint8_t radius, uint16_t start_angle, uint16_t sweep, SSD1306_COLOR color);
void ssd1306_DrawCircle(uint8_t par_x, uint8_t par_y, uint8_t par_r, SSD1306_COLOR color);
void ssd1306_FillCircle(uint8_t par_x,uint8_t par_y,uint8_t par_r,SSD1306_COLOR par_color);
void ssd1306_Polyline(const SSD1306_VERTEX *par_vertex, uint16_t par_size, SSD1306_COLOR color);
void ssd1306_DrawRectangle(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color);
void ssd1306_FillRectangle(uint8_t x1, uint8_t y1, uint8_t x2, uint8_t y2, SSD1306_COLOR color);
```

Display OLED ssd1306

Le funzioni della libreria da noi usate sono state:

```
char ssd1306_WriteString(char* str, SSD1306_Font_t Font, SSD1306_COLOR color);
void ssd1306_SetCursor(uint8_t x, uint8_t y);
```

La funzione *ssd1306_WriteString* scorre una stringa e per ogni carattere chiama *ssd1306_WriteChar*, che legge la bitmap dal font e disegna i pixel corrispondenti nel buffer video, aggiornando la posizione corrente. I pixel non vengono scritti direttamente sul display ma in una memoria interna, che viene poi inviata al controller SSD1306 tramite I²C durante l'aggiornamento dello schermo.

La funzione *ssd1306_SetCursor* imposta la posizione corrente da cui inizierà la scrittura successiva nel buffer del display.

```
void ssd1306_Init(void);
void ssd1306_UpdateScreen(void);
```

La funzione *ssd1306_UpdateScreen* trasferisce il contenuto del buffer video interno al display SSD1306, pagina per pagina. Per ogni pagina di 8 righe verticali, imposta l'indirizzo della pagina nella RAM del controller, poi definisce l'offset orizzontale (colonna) e invia i dati della porzione corrispondente del buffer tramite I²C. In questo modo, l'intera immagine costruita in RAM viene effettivamente mostrata sullo schermo.

Display OLED ssd1306

```
void Show_On_Display()
{

    ssd1306_SetCursor(10, 0);
    ssd1306_WriteString("APC25", Font_11x18, White);


    ssd1306_SetCursor(10, 22);
    ssd1306_WriteString("il livello e'", Font_7x10, White);


    ssd1306_FillRectangle(10, 38, 10 + 11 * 10, 38 + 18, Black);
    ssd1306_SetCursor(10, 38);
    ssd1306_WriteString(stato, Font_11x18, White);

    ssd1306_UpdateScreen();
    HAL_Delay(1000);

}
```

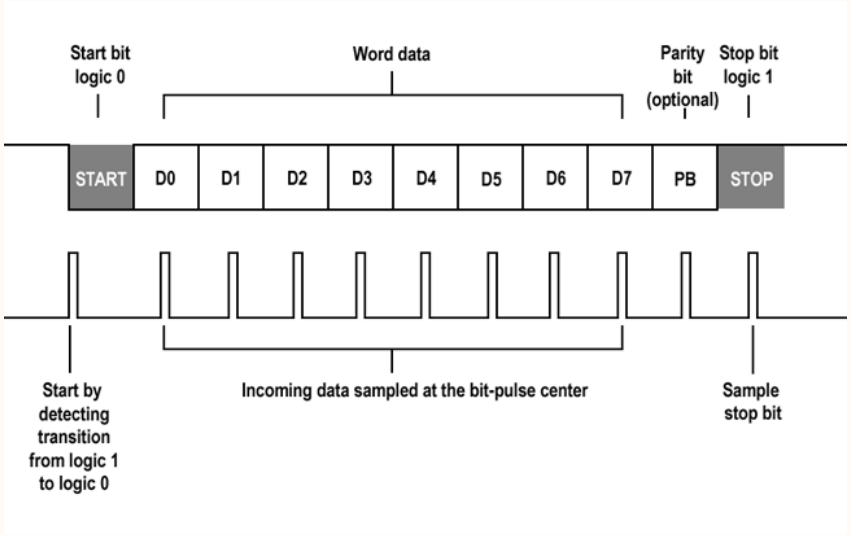
La funzione *Show_On_Display* fa uso delle primitive della libreria illustrate in precedenza per scrivere sul display su più righe utilizzando diversi font. La funzione principale è mostrare “stato”, una variabile globale che tiene traccia del livello dell’acqua.

Comunicazioni - UART

Per l’invio dell’altezza calcolata dal secondo sensore di prossimità del primo sistema al secondo, si è deciso di utilizzare il protocollo di comunicazione seriale asincrono UART. Come principali parametri operativi abbiamo scelto:

- **Baud Rate** pari a 115200 Bits/s
- **Word Length** pari a 8 Bits
- **Non** usare il **Bit di Parità**
- Usare un solo **Bit di Stop**

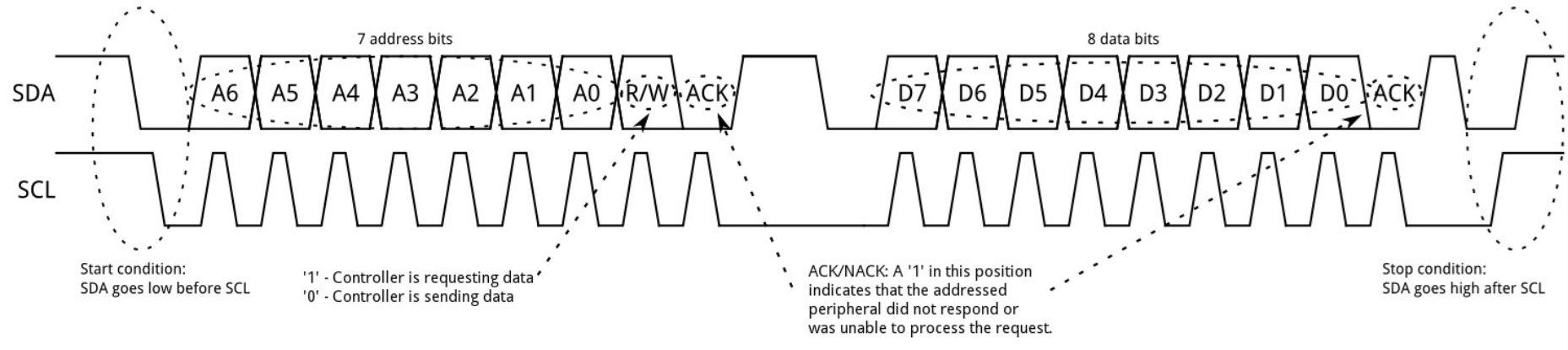
A lato è presente un tipico diagramma temporale di una trasmissione UART.



Comunicazioni - I²C

Per la comunicazione tra il secondo sistema e lo schermo OLED, si utilizza il protocollo di comunicazione seriale sincrono I²C. Il protocollo è composto da tre linee: SDA, SCL e GRND; le prime due si trovano in un valore logico di riposo alto, quindi sono collegate mediante resistenze all'alimentazione.
La comunicazione avviene in questo modo:

- Un master abbassa la linea di dato (intenzione di voler comunicare) e poco dopo abbasserà la linea di clock, ad ogni finestra bassa di clock il master manda un bit e alle finestre alte uno slave campiona il bit.
- La prima cosa che manda il Master sono i 7 bit dell'identificativo univoco dello slave con cui si vuole comunicare, seguito dal bit per leggere o scrivere su quello slave. Quest'ultimo, se esiste, risponderà con un ACK.
- Questo processo è seguito dalla trasmissione dei Byte o da parte del Master allo Slave (W) o dallo Slave al Master (R), ad ogni Byte mandato c'è la risposta di un ACK da chi ha ricevuto. In conclusione verrà mandato il bit di stop indicativo della fine trasmissione



Riepilogo Pin Usati - S1 e S2

Nome Pin	Configurazione	Collegamento	Sistema
PA15	TIM2_CH1; PWM GENERATION	Servomotore	S1
PA8	GPIO Output	Sensore di prossimità 1 – Trigger	S1
PC9	TIM3_CH4; Input Capture Direct Mode	Sensore di prossimità 1 – Echo	S1
PD13	GPIO Output	Sensore di prossimità 2 – Trigger	S1
PD12	TIM4_CH1; Input Capture Direct Mode	Sensore di prossimità 2 – Echo	S1
PC12	Uart5 TX	S2 – Uart5 RX PD2	S1
PD2	Uart5 RX	S1 – Uart5 TX PC12	S2
PB7	I2C_1 SDA	OLED ssd1306 – SDA	S2
PB6	I2C_1 SCL	OLED ssd1306 – SCL	S2

Sistema di Tx - S1 - Timer

Rispetto a quanto detto prima per i sensori citati, si ricapitolano di seguito i timer usati per il sistema con un clock di 48MHz

Nome Timer	Canale	Modalità	Prescaler	ARR	Collegamento
TIM2	CH1	PWM Generation CH1	228	9999	Servomotore
TIM3	CH4	Input Capture direct mode	47	65535	Sensore di prossimità 1 - Echo
TIM4	CH1	Input Capture direct mode	47	65535	Sensore di prossimità 2 - Echo

Sistema di Tx - S1 - Variabili di Sistema

Per quanto riguarda l'integrazione corretta di tutti i sensori nel primo sistema, di seguito sono riportate tutte le costanti e variabili di sistema necessarie:

- *buffer_misure* serve in quanto per la detection dell'oggetto non dipende da una singola misura del sensore di prossimità ma si è decisi di prendere sempre il valore mediano di ogni 7 misurazioni. Clò ha garantito di avere maggiore precisione e sicurezza. *indice_buffer* serve per scorrere circolarmente il vettore e *buffer_pieno* è una variabile logica per triggerare nel ciclo caldo il calcolo del mediano. *s1* e *s2* sono le istanze della struct definita in precedenza.
- *s2_distance_cm_old* rappresenta l'ultimo valore mandato dal UART all'altro sistema, questa serve in quanto non vogliamo che ci sia un overhead nel trasmettere sempre l'ultima lettura del secondo sensore ma mandare solo le letture che hanno avuto una variazione determinante rispetto a questo valore, che ovviamente cambierà nel tempo
- *threeshold_a* e *threeshold_e* sono rispettivamente la soglia di attivazione per il riconoscimento del fondo del recipiente e la soglia di terminazione per la fine di erogazione dell'acqua. La prima soglia in particolare è riprogrammabile in base al recipiente da utilizzare. Nel nostro caso d'uso abbiamo usato una borraccia molto profonda da riempire, di conseguenza la soglia d'attivazione è stata impostata ad un valore alto. Altresì, con un bicchiere sarebbe stata minore.
- *theta0* e *thetaf* sono l'angolo a riposo e di massima tensione del servomotore
- *h_max* è la capienza massima del dispenser usata, e serve per il calcolo della variazione da valutare come significativa o meno.
- *StatoServo_t*: una struttura dati che definisce i possibili stati in cui può trovarsi il sistema di controllo del servomotore. È stato realizzato un automa a stati finiti per garantire la giusta attivazione del servomotore in base alla detection dell'oggetto, *statoCorrente* è inizializzato allo stato *STATO_IN_ATTESA_A_0*

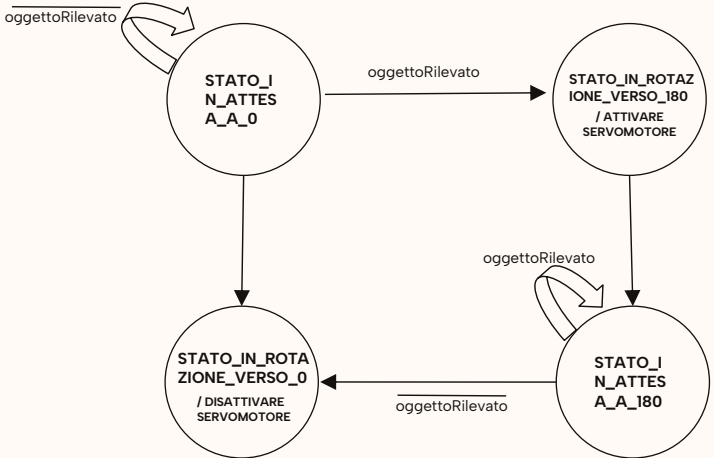
```c
#define NUM_MISURAZIONI 7

uint32_t buffer_misure[NUM_MISURAZIONI] = {0};
uint8_t indice_buffer = 0;
bool buffer_pieno = false;

static bool oggettoRilevato = false;



hcsr04_t s1 = {0, 0, 0, 0, 0};
hcsr04_t s2 = {0, 0, 0, 0, 0};


static uint32_t s2_distance_cm_old = 0;

const uint32_t threeshold_a = 18;
const uint32_t threeshold_e = 3;
const uint8_t theta0 = 0;
const uint8_t thetaf = 180;

typedef enum {
    STATO_IN_ATTESA_A_0,
    STATO_IN_ROTAZIONE_VERSO_180,
    STATO_IN_ATTESA_A_180,
    STATO_IN_ROTAZIONE_VERSO_0
} StatoServo_t;

static StatoServo_t statoCorrente = STATO_IN_ATTESA_A_0;


const uint32_t h_max = 15;
```

Sistema di Tx - S1 - FSM

```
switch (statoCorrente) {
  case STATO_IN_ATTESA_A_0:

    if (oggettoRilevato) {
      statoCorrente = STATO_IN_ROTAZIONE_VERSO_180;
    }
    break;

  case STATO_IN_ROTAZIONE_VERSO_180:

    for (uint8_t angle = theta0; angle <= thetaf; angle += 5) {
      Set_Servo_Angle(&htim2, TIM_CHANNEL_1, angle);
      HAL_Delay(50);
    }
    Set_Servo_Angle(&htim2, TIM_CHANNEL_1, thetaf);

    statoCorrente = STATO_IN_ATTESA_A_180;
    break;

  case STATO_IN_ATTESA_A_180:

    if (!oggettoRilevato) {
      statoCorrente = STATO_IN_ROTAZIONE_VERSO_0;
    }
    break;

  case STATO_IN_ROTAZIONE_VERSO_0:

    for (uint8_t angle = thetaf; angle > theta0; angle -= 5) {
      Set_Servo_Angle(&htim2, TIM_CHANNEL_1, angle);
      HAL_Delay(50);
    }
    Set_Servo_Angle(&htim2, TIM_CHANNEL_1, theta0);

    statoCorrente = STATO_IN_ATTESA_A_0;
    break;
}
```

L'implementazione della FSM è presente all'interno del **ciclo caldo**, dove si evidenzia cosa deve fare il sistema in base allo stato in cui si trova.

- In **STATO_IN_ATTESA_A_0**, il sistema si trova in una condizione di attesa passiva, in cui si controlla se l'oggetto è stato rilevato dal sensore di prossimità. Se la condizione è vera (*oggettoRilevato == true*), si passa allo stato successivo.
- In **STATO_IN_ROTAZIONE_VERSO_180**, il servomotore viene fatto ruotare gradualmente dalla posizione iniziale *theta0* fino alla posizione finale *thetaf*: il momento avviene durante il ciclo for, che aumenta l'angolo ogni 5 gradi, con un ritardo di 50 ms tra ciascun passo. Una volta conclusa la rotazione, il sistema entra nello stato *STATO_IN_ATTESA_A_180*.
- In **STATO_IN_ATTESA_A_180**, il servomotore è fermo nella posizione finale mentre il sistema resta in attesa di un nuovo cambiamento, ovvero la scomparsa dell'oggetto o la fine dell'erogazione dell'acqua: quando questa condizione è verificata, si passa allo stato successivo.
- In **STATO_IN_ROTAZIONE_VERSO_0**, il servomotore deve tornare nella posizione iniziale. Viene effettuato quindi un ulteriore ciclo for simile al precedente ma nel senso opposto: si parte da *thetaf* fino a giungere a *theta0* decrementando ogni volta l'angolo di 5 gradi con un ritardo di 50 ms per ogni passo. Alla fine, il sistema ritorna allo stato iniziale.



Sistema di Tx - S1 - Mediano

In merito al calcolo del valore mediano sono state realizzate due funzioni:

- ***comparatore***: effettua la differenza tra due termini, castati a uint32_t. In base al risultato è possibile capire quale dei due valori precede l'altro
  - a < b differenza negativa
  - a > b differenza positiva
  - a = b differenza 0

- ***calcola_mediana***: restituisce il valore mediano dopo aver fatto una copia del buffer fornito in *copia[size]* tramite la funzione *memcpy*, dopo di che si usa la funzione *qsort* che implementa l'algoritmo d'ordinamento Quick Sort dove la comparazione tra elementi avviene mediante la funzione *comparatore*. Una volta ordinato l'array restituisce il mediano.

```c
int comparatore(const void *a, const void *b) {
    return (*(uint32_t*)a - *(uint32_t*)b);
}

uint32_t calcola_mediana(uint32_t *buffer, uint8_t size) {
    uint32_t copia[size];
    memcpy(copia, buffer, size * sizeof(uint32_t));
    qsort(copia, size, sizeof(uint32_t), comparatore);
    return copia[size / 2];
}
```

Sistema di Tx - S1 - Mediano

All'interno del ciclo caldo si osserva che dopo la lettura del valore del primo sensore, avviata tramite la funzione *HCSR04_Trigger*, vengono effettuatii seguenti passaggi:

- viene salvato il valore in *buffer_misure*
- si aggiorna *indice_buffer* con la logica di un array circolare
- se l'indice vale 0 dopo l'aggiornamento significa che ho completato 7 misurazioni quindi *buffer_pieno* diventa true
- se la condizione *buffer_pieno* è true entro nel blocco then e calcolo il valore mediano, in base a ciò:
    - se non era stato rilevato in precedenza nessun oggetto la variabile *oggettoRilevato* può passare a true se il mediano è strettamente maggiore della soglia di terminazione e minore della soglia di attivazione (significa che ho rilevato il fondo di un recipiente)
    - se era stato rilevato in precedenza può passare a false *oggettoRilevato* se la mediana è maggiore uguale della soglia di attivazione (oggetto allontanato) o la mediana è minore uguale della soglia di terminazione (livello dell'acqua massimo all'interno del recipiente)
- all'interno del ramo thenviene impostato *buffer_pieno* a false.
- Infine se c'è stata una variazione di *oggettoRilevato* c'è l'avanzamento nella FSM

```
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

  HCSR04_Trigger(GPIOA, GPIO_PIN_8, &htim3);
  HAL_Delay(100);

  buffer_misure[indice_buffer] = s1.distance_cm;
  indice_buffer = (indice_buffer + 1) % NUM_MISURAZIONI;
  if (indice_buffer == 0) buffer_pieno = true;

  if (buffer_pieno) {
      uint32_t mediana = calcola_mediana(buffer_misure, NUM_MISURAZIONI);

      if (!oggettoRilevato) {
          if (mediana > threeshold_e && mediana < threeshold_a) {
              oggettoRilevato = true;
          }

      } else {
          if (mediana >= threeshold_a || mediana <= threeshold_e) {
              oggettoRilevato = false;
          }
      }
      buffer_pieno = false;
  }
```

Sistema di Tx - S1 - Delta variation

Per evitare overhead nell'uso dell'UART, inviando impropriamente qualunque valore calcolato dal secondo sensore di prossimità anche se non fa variare le soglie di Alto – Medio – Basso visualizzabili nel ricevitore; si è pensato di inviare solo le misurazioni che hanno una differenza rispetto al valore precedentemente inviato tale da essere maggiore del 5% dell'altezza massima del contenitore.
Come notiamo la differenza è in modulo, se la nuova misura è più grande della vecchia è una semplice differenza tra le due, se invece è il contrario la differenza cambia l'ordine degli operandi

```
bool delta_variation(){
  uint32_t delta;

    if (s2.distance_cm > s2_distance_cm_old){
        delta = s2.distance_cm - s2_distance_cm_old;}
    else{
        delta = s2_distance_cm_old - s2.distance_cm;}

  bool condition = (float) delta > (float) h_max*0.05;

  return condition;

}
```

Nel ciclo caldo è possibile osservare come dopo la seconda misurazione, si fa una trasmissione bloccante solo se *delta_variation* è true, oltre ad aggiornare il valore vecchio con la nuova misura, si noti come nella trasmissione essendo che l'UART manda solo 8 bit alla volta, nella firma della funzione posso usare solo valori del tipo uint8_t. Quindi si fa un casting ad un puntatore uint8_t al riferimento della nostra misurazione (di tipo uint32_t) e si specifica che la dimensione del buffer che si manda è 4. In questo modo si scompone automaticamente il nostro elemento a 32 bit in un vettore di 4 elementi a 8 bit.

```
HCSR04_Trigger(GPIOD, GPIO_PIN_13, &htim4);
HAL_Delay(100);

if(delta_variation()){
      HAL_UART_Transmit(&huart5, (uint8_t *) &s2.distance_cm, 4, HAL_MAX_DELAY);
      s2_distance_cm_old = s2.distance_cm;
    }

HAL_Delay(100);
```

Sistema di Tx - S1 - Ciclo Caldo

In conclusione, anche se già presentato parzialmente nelle slide di prima, si mostra qui le inizializzazione delle componenti e il ciclo caldo completo

```
228    MX_GPIO_Init();
229    MX_USB_PCD_Init();
230    MX_TIM2_Init();
231    MX_TIM3_Init();
232    MX_TIM4_Init();
233    MX_UART5_Init();
234    /* USER CODE BEGIN 2 */
235
236    HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_4);
237    HAL_TIM_IC_Start_IT(&htim4, TIM_CHANNEL_1);
238    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
239
240    Set_Servo_Angle(&htim2, TIM_CHANNEL_1, 0);
241    HAL_Delay(500);
```

```
246  while (1)
247  {
248    /* USER CODE END WHILE */
249
250    /* USER CODE BEGIN 3 */
251
252    HCSR04_Trigger(GPIOA, GPIO_PIN_8, &htim3);
253    HAL_Delay(100);
254
255    buffer_misure[indice_buffer] = s1.distance_cm;
256    indice_buffer = (indice_buffer + 1) % NUM_MISURAZIONI;
257    if (indice_buffer == 0) buffer_pieno = true;
258
259    if (buffer_pieno) {
260        uint32_t mediana = calcola_mediana(buffer_misure, NUM_MISURAZIONI);
261
262        if (!oggettoRilevato) {
263            if (mediana > threeshold_e && mediana < threeshold_a) {
264                oggettoRilevato = true;
265            }
266
267        } else {
268            if (mediana >= threeshold_a || mediana <= threeshold_e) {
269                oggettoRilevato = false;
270            }
271        }
272        buffer_pieno = false;
273    }
274
275    switch (statoCorrente) {
276      case STATO_IN_ATTESA_A_0:
277
278        if (oggettoRilevato) {
279
280          statoCorrente = STATO_IN_ROTAZIONE_VERSO_180;
281        }
282        break;
```

Sistema di Tx - S1 - Ciclo Caldo

```
284         case STATO_IN_ROTAZIONE_VERSO_180:
285
286           for (uint8_t angle = theta0; angle <= thetaf; angle += 5) {
287             Set_Servo_Angle(&htim2, TIM_CHANNEL_1, angle);
288             HAL_Delay(50);
289           }
290           Set_Servo_Angle(&htim2, TIM_CHANNEL_1, thetaf);
291
292
293           statoCorrente = STATO_IN_ATTESA_A_180;
294           break;
295
296         case STATO_IN_ATTESA_A_180:
297
298           if (!oggettoRilevato) {
299
300             statoCorrente = STATO_IN_ROTAZIONE_VERSO_0;
301           }
302           break;
303
304         case STATO_IN_ROTAZIONE_VERSO_0:
305
306           for (uint8_t angle = thetaf; angle > theta0; angle -= 5) {
307             Set_Servo_Angle(&htim2, TIM_CHANNEL_1, angle);
308             HAL_Delay(50);
309           }
310           Set_Servo_Angle(&htim2, TIM_CHANNEL_1, theta0);
311
312
313           statoCorrente = STATO_IN_ATTESA_A_0;
314           break;
315       }
316
317
318       HCSR04_Trigger(GPIOD, GPIO_PIN_13, &htim4);
319       HAL_Delay(100);
320
321       if(delta_variation()){
322             HAL_UART_Transmit(&huart5, (uint8_t *) &s2.distance_cm, 4, HAL_MAX_DELAY);
323             s2_distance_cm_old = s2.distance_cm;
324           }
325
326       HAL_Delay(100);
```

Sistema di Rx - S2- Variabili di sistema

- *buffer_rx*: viene usata per salvare il messaggio che riceve dal sistema 1. Tale vettore è di tipo uint8_t contenente al più 4 elementi.
- *received*: contiene la conversione in formato uint32_t di *buffer_rx* ed è usata dalla funzione *Calculate_level*
- *max_value*: può essere modificato in base alla capienza del dispenser usato, nel nostro caso 15, che indica i cm di profondità.
- *rimanente*: variabile usata dalla funzione Calculate_level.
- *stato*: stringa contenente il livello dell'acqua calcolato

```
/* USER CODE BEGIN PV */

uint8_t buffer_rx[4];
uint32_t received;
char stato[20]="";
uint32_t max_value=15;
uint32_t rimanente;


/* USER CODE END PV */
```

```
void Calculate_Level()
{
    if (received < (max_value+1) ){
      rimanente = max_value - received;
    }
    else {
      rimanente= max_value;
    }
    memset(stato, 0, sizeof(stato));

    if (rimanente >= 7) {
      strcpy(stato, "ALTO");
    } else if (rimanente >= 3) {
      strcpy(stato, "MEDIO");
    } else{
      strcpy(stato, "BASSO");
    }
```

La funzione *Calculate_Level* calcola la quantità d'acqua presente nel dispenser e la salva nella variabile rimanente. La stringa *stato* viene quindi inizializzata e viene poi sovrascritta con il livello calcolato. La suddivisione è stata fatta tra i livelli : ALTO se la quantità d'acqua nel dispenser supera i 7cm; MEDIO se è compresa tra 3 e 7; BASSO altrimenti. Tali valori possono ovviamente essere modificati in base all'utilizzo di dispenser di diversa altezza

Sistema di Rx - S2 - Callback di Ricezione

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if (huart->Instance == UART5)
  {
    received =
        (uint32_t)buffer_rx[0]       |
        ((uint32_t)buffer_rx[1] << 8)  |
        ((uint32_t)buffer_rx[2] << 16) |
        ((uint32_t)buffer_rx[3] << 24);


      HAL_UART_Receive_IT(&huart5, buffer_rx, sizeof(buffer_rx));


  }
}
```

Nella callback di ricezione della comunicazione UART viene gestito il dato che fa riferimento alla distanza misurata dal sensore di prossimità che osserva il livello di acqua. Tale valore viene convertito in formato uint32_t in quanto il valore originale da trasmettere era espresso su 32 bit, ma spedito come 4 messaggi di 8 bit.

Infine il sistema riattiva la Ricezione NON bloccante

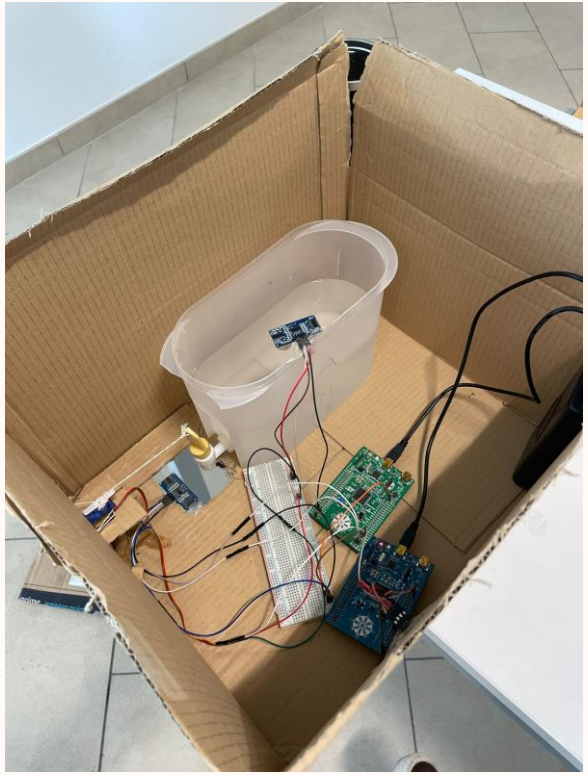
Sistema di Rx - S2 - Ciclo Caldo

```
ssd1306_Init();

HAL_UART_Receive_IT(&huart5, buffer_rx, sizeof(buffer_rx));

HAL_Delay(200);

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    Calculate_Level();
  Show_On_Display();

  HAL_Delay(200);
  /* USER CODE END WHILE */

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

Come prima cosa nel Main viene inizializzato il display tramite la primitiva associata.
Viene poi avviata la ricezione non bloccante, il cui messaggio viene salvato nella variabile buffer_rx.
Nel ciclo caldo quello che succede è semplicemente la chiamata alle funzioni descritte in precedenza.

Inoltre come nel sistema 1 il clock di sistema è di 48MHz

Prototipo Fisico



Filmino dimostrativo

Riferimenti Bibliografici - Sitografici

https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf

https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf

https://github.com/afiskon/stm32-ssd1306

https://www.flaticon.com/

GRAZIE PER
L’ATTENZIONE