PROGETTO IOT

NOSCHESE VALERIO

PELUSO RAFFAELE

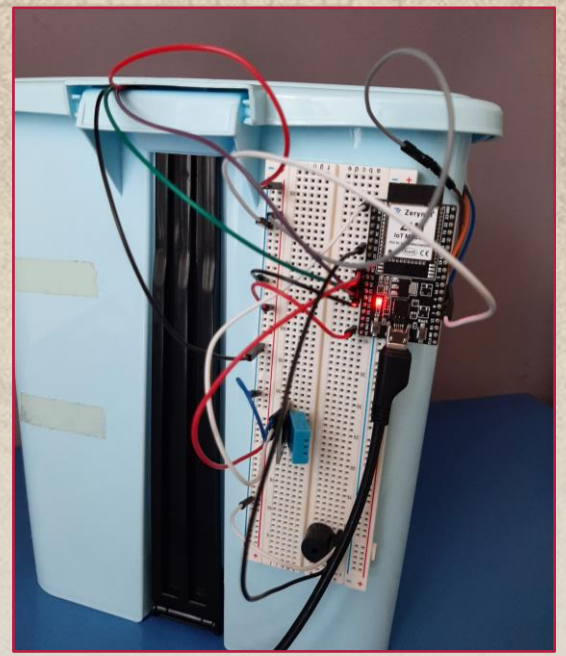
ROBERTO GAETANO

SMART BIN

- L’idea del progetto è nata dall‘intenzione di migliorare la gestione della raccolta dei rifiuti in luoghi turistici.
- In tali aree, il controllo dei rifiuti risulta essere troppo basso o addirittura inesistente.
- Il prototipo in questione è un bidone della spazzatura che fornisce dati come il livello dei rifiuti e dati meteorologici quali temperatura e umidità, recuperabili dalla rete.
- L'obiettivo principale è che quando il livello di riempimento del cestino è massimo, questo invii una notifica ai netturbini incaricati dello svuotamento. Così il luogo rimarrà in condizioni ottimali per la flora, la fauna e i visitatori.

FUNZIONALITÀ

- Gestione di allarmi con notifiche per il raggiungimento della capienza massima
- Gestione di allarmi con notifiche per valori critici di umidità o temperatura
- Visualizzazione dei dati misurati
- Allarme acustico in caso di pericolo incendio del bidone



STRUMENTI UTILIZZATI

- Sensore a ultrasuoni HCSR04 – gestione della capienza del bidone
- Sensore di temperatura e umidità DHT11 – gestione di valori critici
- Display LCD1602 I2C– visualizzazione misurazioni per i passanti
- Buzzer - generazione del suono
- ZDM – comunicazione in rete dei valori tramite MQTT e gestione remota buzzer/display
- Grafana – dashboard per visualizzazione in real time dei dati con notifiche per allarmi

PROCESSO DI PROGETTAZIONE

- Definizione delle classi per i sensori
- Codice generale di gestione
- Comunicazione con ZDM
- Dashboard Grafana

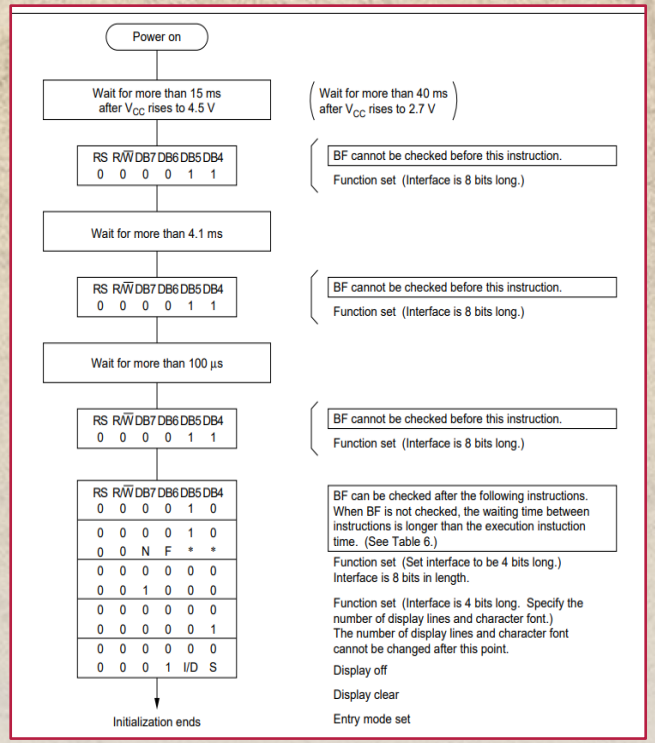
DISPLAY LCD1602 - I2C

- Descrizione
  - L'LCD 16x2 è un display LCD con adattatore I2C.
  - L'adattatore ha 4 pin: VCC per l'alimentazione (5V); GND per massa; SDA: dati seriali I2C; SCL: clock I2C.
  - Il display permette di mostrare i caratteri a video, attraverso i quali è possibile comporre messaggi di servizio.
- Problema del display LCD1602
  - Datasheet incompleto per adattatore I2C
  - Formato dei messaggi
  - Risoluzione

Display LCD1602 con adapter I2C

DISPLAY LCD1602 - I2C: CODICE

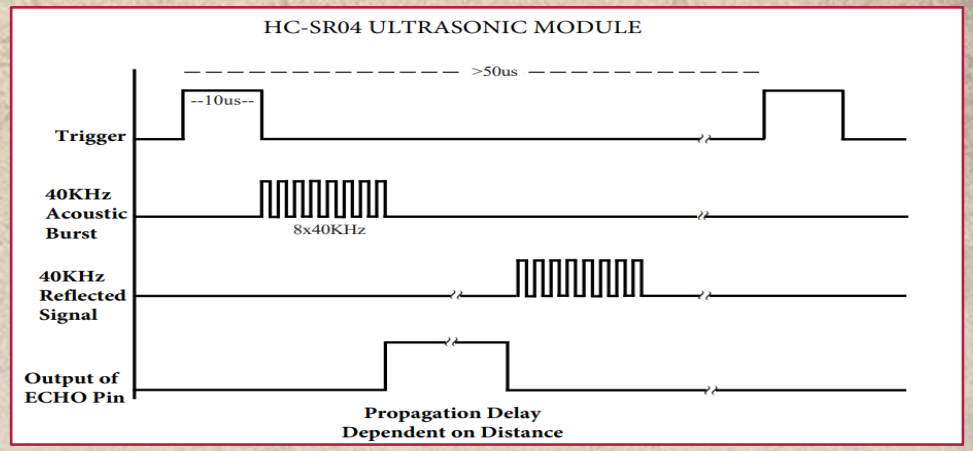
```python
def __init__(self, port):
    self.porta = port #porta i2c
    self.backup = 0

    #inizializzazione display
    #possiamo inviare 4bit dati alla volta, quindi dobbiamo
    #settare l'interfaccia lcd a 4bit (pag.46 datasheet HD44780U)
    self.invia(INIZIALIZATION)
    self.toggle_enable()
    sleep(5)
    self.invia(INIZIALIZATION)
    self.toggle_enable()
    sleep(2)
    self.invia(INIZIALIZATION)
    self.toggle_enable()
    sleep(2)

    self.invia(CONFIGURATION)
    self.toggle_enable()

    self.invia_istruzione(FUNCTION_SET | 0x08, COMMAND_MODE) #settiamo 2 linee con font 5x8 dots
    self.invia_istruzione(DISPLAY_CONTROL | 0x04, COMMAND_MODE)#accendiamo display
    self.invia_istruzione(ENTRY_MODE_SET | 0x02, COMMAND_MODE)#autoincremento cursore
    self.invia_istruzione(CLEAR, COMMAND_MODE)

    sleep(5)
```

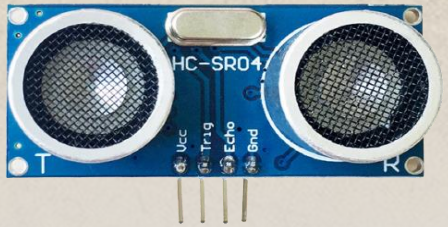



```python
def invia(self, data):
    self.backup = data
    self.porta.write(bytearray([data]))


def toggle_enable(self): #pag58 HD44780U
    sleep(5)
    self.invia(self.backup | ENABLE)
    sleep(5)
    self.invia(self.backup & ~ENABLE)
    sleep(5)

def invia_istruzione(self, data, mode):
    #formato messaggi(1 byte), ogni bit setta valore logico pin lcd
    #DB7|DB6|DB5|DB4|A|E|(R/W)|RS (datasheet LCD1602 pag.10)
    #possiamo inviare 4 bit istruzione alla volta
    #quindi ci servono 2 write per scrivere un'istruzione
    #inviamo prima la parte alta, poi la parte bassa(pag.22 datasheet HD44780)
    # mode = 1 (RS = 1) inviamo dati
    # mode = 0 (RS = 0) inviamo un comando

    #parte alta istruzione
    high = data & 0xF0
    message = high | mode | BACKLIGTH_ON

    self.invia(message)
    self.toggle_enable()

    #parte bassa istruzione
    low = ((data<<4) & 0xF0)
    message = low | mode | BACKLIGTH_ON

    self.invia(message)
    self.toggle_enable()

    sleep(2) #istruzione più lunga impiega 1.52 ms

def clear(self):
    self.invia_istruzione(CLEAR, COMMAND_MODE)
```

SENSORE AD ULTRASUONI

- HC-SR04 è un sensore a ultrasuoni.
- Il sensore ha 4 pin: VCC per l'alimentazione (5V); GND per massa; Trig: pin che avvia impulsi ultrasonici; Echo: pin che definisce il tempo trascorso.
- Il sensore permette di misurare la distanza da un ostacolo. Quando il microcontrollore imposta il pin Trigger su 1, il sensore invia 8 impulsi ultrasonici. Una volta ricevuti, il sensore imposta a 1 il pin Echo, viene quindi misurato il tempo che intercorre tra l'attivazione del pin Trigger e l'impostazione del pin Echo. Infine, utilizzando la velocità del suono, si ricava la distanza dall'ostacolo.





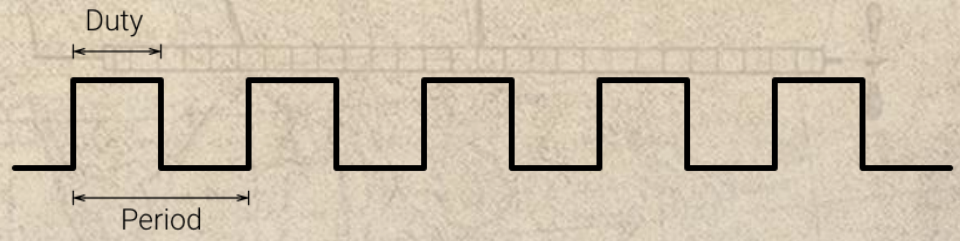
SENSORE AD ULTRASUONI: CODICE

```
def __init__(self, echoPin, triggerPin):
    self.echoPin = echoPin
    self.triggerPin = triggerPin
    gpio.mode(self.echoPin, INPUT)
    gpio.mode(self.triggerPin, OUTPUT)

def distance(self):
    #fase di inizializzazione per stabilizzare il sensore
    gpio.set(self.triggerPin, LOW)
    sleep(500, MILLIS)
    #invio un impulso di 10 microsecondi dal triggePin
    gpio.set(self.triggerPin, HIGH)
    sleep(10, MICROS)
    gpio.set(self.triggerPin, LOW)
```

```
    start = time.time()
    end = time.time()

    #calcolo tempo durante il quale echoPin è settato ad HIGH
    while gpio.get(self.echoPin) == 0:
        start = time.time()
    while gpio.get(self.echoPin) == 1:
        end = time.time()

    #tempo che impiega il segnale ad arrivare
    duration = end - start
    #343 m/s   34300 cm/s
    distance = duration * 34300 / 2 #andata e ritorno
    distance = round(distance, 2)
    return distance
```

BUZZER + PWM

- I buzzer sono alimentati a corrente continua e dotati di circuito integrato. Possono essere classificati come attivi e passivi.
- Hanno 2 pin: il pin da connettere per alimentarlo e gestirlo; GND per massa.

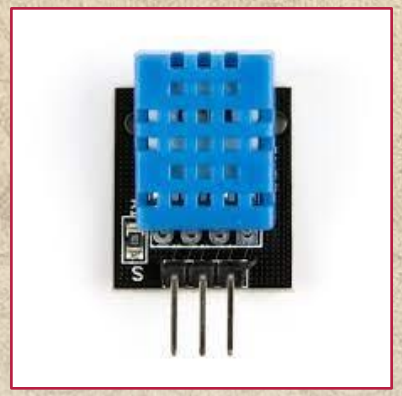


- Il PWM (Pulse Width Modulation) è la tecnica per erogare potenza parziale ad un carico tramite mezzi digitali.
- Il Duty Cycle è la misura percentuale di quanto a lungo il segnale rimane on rispetto al periodo.

BUZZER + PWM: CODICE

```
class buzzer:

    def __init__(self, buzzerPin):
        self.buzzerPin = buzzerPin
        gpio.mode(self.buzzerPin, OUTPUT)

    def allarme(self): #gestione di un allarme temporizzato
        tempo = time.time()
        timeout = tempo + 15
        while tempo <= timeout:
            for x in range(2272,1000,-25): #440hrz a 1000hrz
                pwm.write(self.buzzerPin,x,100, MICROS)
                sleep(50)
            for x in range(1000,2272,25):# 1000hrz a 440hrz
                pwm.write(self.buzzerPin,x,100,MICROS)
                sleep(50)
            tempo = time.time()

    def stop_allarme(self):
        pwm.write(self.buzzerPin, 0,0)
```

SENSORE DHT11



```
class DHT11:

    def __init__(self,pin):
        self.dhtPin = pin
        gpio.mode(self.dhtPin, INPUT)

    def hum_temp(self):
        return dht11.read(self.dhtPin)
```

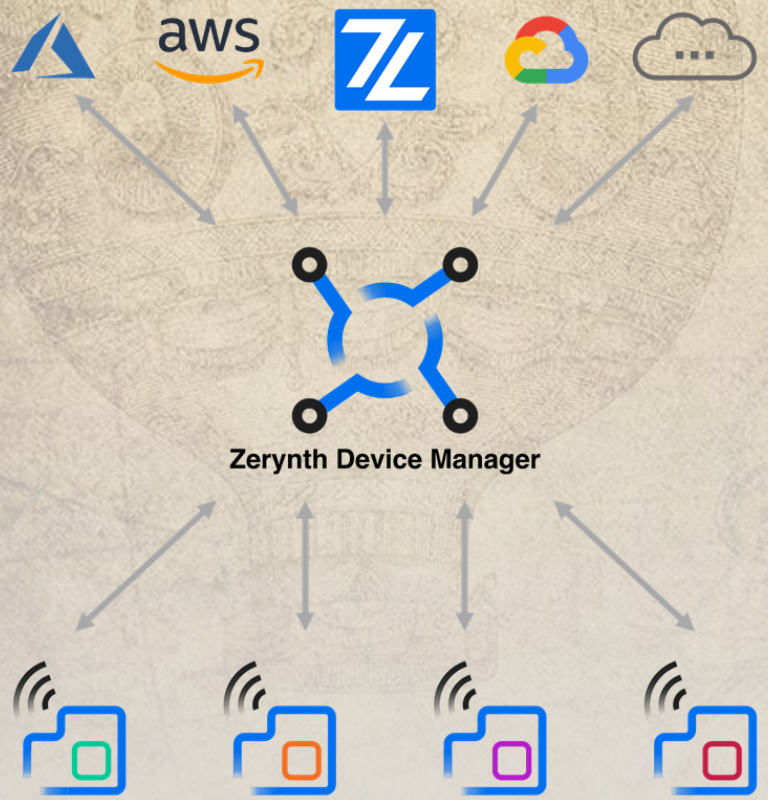
- Il DHT11 è un sensore di umidità e temperatura.
- Il sensore ha 3 pin: VCC per l'alimentazione (3.3v); GND per massa; il pin di lettura.
- Il sensore permette di ricavare il valore di temperatura e umidità rilevati in tempo reale nell'ambiente circostante.

MAIN

```
while True:
    try: #dht11 può fallire nella lettura
        hum, temp = dht.hum_temp(dhtPin)
    except:
        print("errore lettura dht11")
    riempimento = bidoneRiemp()
    display.aggiorna_valori(temp, hum)
    if(riempimento>=0 and riempimento<=100): #percentuale corretta per sensore a ultrasuoni
        agent.publish(payload={"temp": str(temp), "hum": str(hum), "riempimento": str(riempimento)}, tag = "bidone")
        print("dati pubblicati: ", temp, hum, riempimento)
    else: #valori anomali del sensore ad ultrasuoni
        agent.publish(payload={"temp": str(temp), "hum": str(hum)}, tag = "bidone")
        print("dati pubblicati: ", temp, hum)
    sleep(2500)
```

```
def alert(agent, args):
    print("Job ricevuto!", args)
    if not "causa" in args:
        return {"msg": "Argomento non valido per il job alert"}
    c = args["causa"]
    if c == "temperatura": #temperatura critica
        display.allarme_temperatura()
        buzzer.allarme()
    elif c == "riempimento": #bidone pieno
        display.allarme_pieno()
    elif c == "reset": #reset display
        display.default()
    else:
        return {"msg": "Argomento non valido per il job alerta"}
    buzzer.stop_allarme() #spegnimento buzzer
    return {"msg": "allarme attivato: %s" % c}

#riempimento del bidone in percentuale
def bidoneRiemp():
    dist = hcsr.distance(echoPin,triggerPin)
    return 100 - (dist * 100) /altezzaBidone
```

ZDM



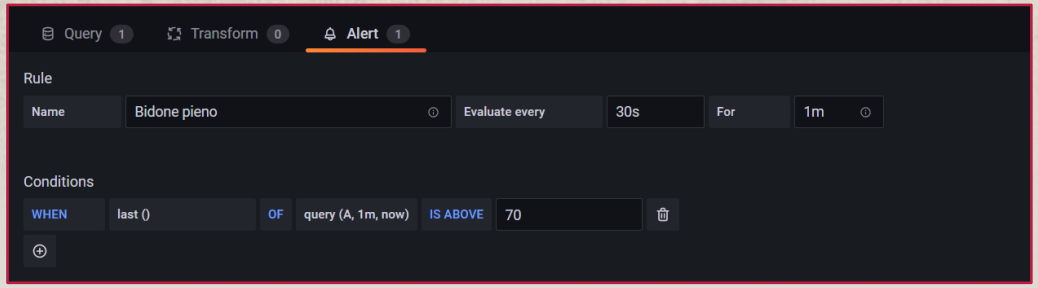
- Zerynth Device Manager (ZDM) è un servizio di gestione di dati per il monitoraggio e gestione remota dei dispositivi IoT in modo sicuro.
- ZDM raccoglie e aggrega i dati prodotti.
- Comunicazione con la classe agent tramite MQTT.
- Gestione remota tramite jobs.

```
#configurazione agente zdm
agent = zdm.Agent(jobs = {"alert":alert})
agent.start()
print("pubblicazione...")
```

GRAFANA

- Grafana è un'applicazione web per la visualizzazione e l'analisi interattiva dei dati.
- Permette di visualizzare grafici e allarmi per il web.
- I dati possono essere visualizzati su appositi pannelli (dashboard). La creazione di questi pannelli può essere eseguita tramite builder di query interattivi.


```
SELECT
  ts_device as time,
  (payload->>'riempimento')::float as riempimento
FROM data
ORDER BY ts_device
```



Name	Type	Description
ts_device	time	Timestamp corresponding to when the data has been sent by device
ts_zdm	time	Timestamp corresponding to when the data has arrived on ZDM
ts_db	time	Timestamp corresponding to when the data has been save on database
workspace_id	string	Workspace id of the device
fleet_id	string	Fleet id of the device
device_id	string	Id of the device
tag	string	Publication tag with whom the data has been sent by device
payload	jsonb	Payload published by device