Progetto Arduino di Sistemi di Elettronica Digitale

ESP32 Bluetooth Liquid Controller

Realizzato da: Ahir Armando

Introduzione

Ho voluto realizzare questo progetto, ispirandomi ad un viaggio che feci nella lontana estate del 2012, dai miei nonni in India.

Ogni casa in questa zona dell'India è dotata, di serbatoi di acqua che, venivano riempiti due volte al giorno, attraverso un sistema idraulico, gestito dalla compagnia garante; ciò implicava che, se si sprecava l'acqua inutilmente, c'era il rischio di rimanerne senza, oppure usufruire di pompe (elettriche o manuali) per prelevare l'acqua dalle falde, aspettando il rifornimento successivo.

Durante il mio soggiorno in India, mi è capitato di svuotare completamente il serbatoio per pulirlo. In quel momento ho iniziato a pensare ad un impianto che potesse essere utile per monitorare il livello dell'acqua ed avvertire quando l'acqua nel serbatoio fosse in esaurimento.

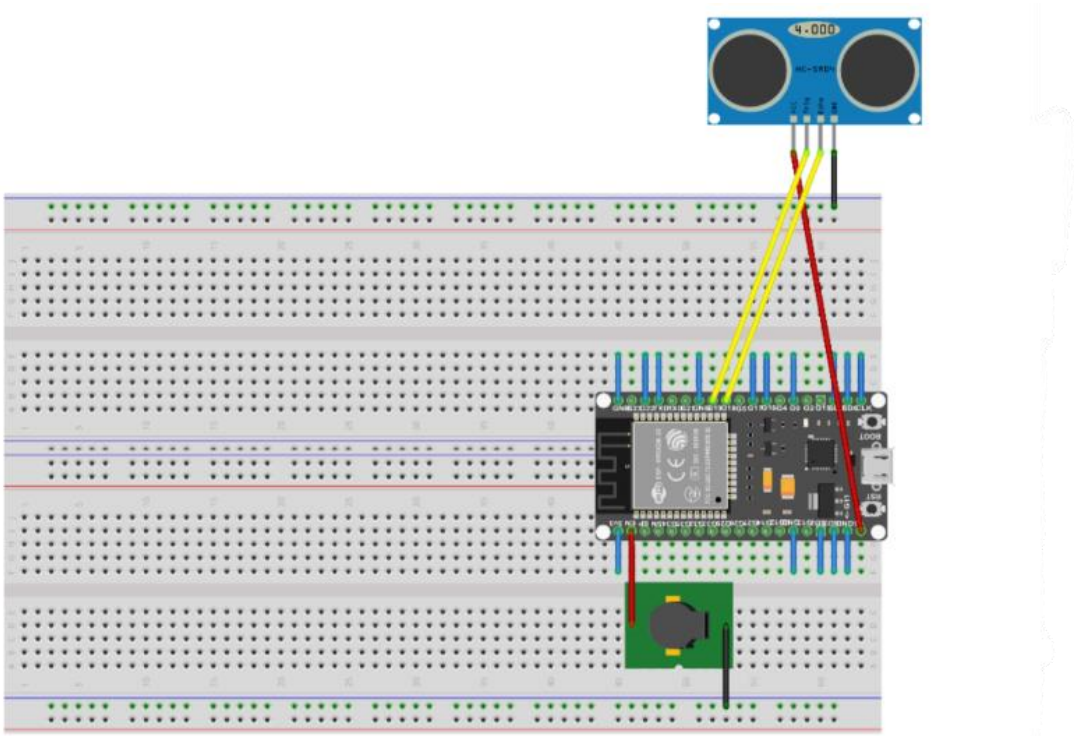
Ho voluto realizzare questo modulo di monitoraggio del livello dell'acqua all'interno di un serbatoio, controllato via Bluetooth, mettendomi alla prova sulla programmazione di microcontrollori. Per sviluppare questo progetto ho utilizzato il microcontrollore ESP32 interfacciandolo con l'ambiente di sviluppo di arduino.

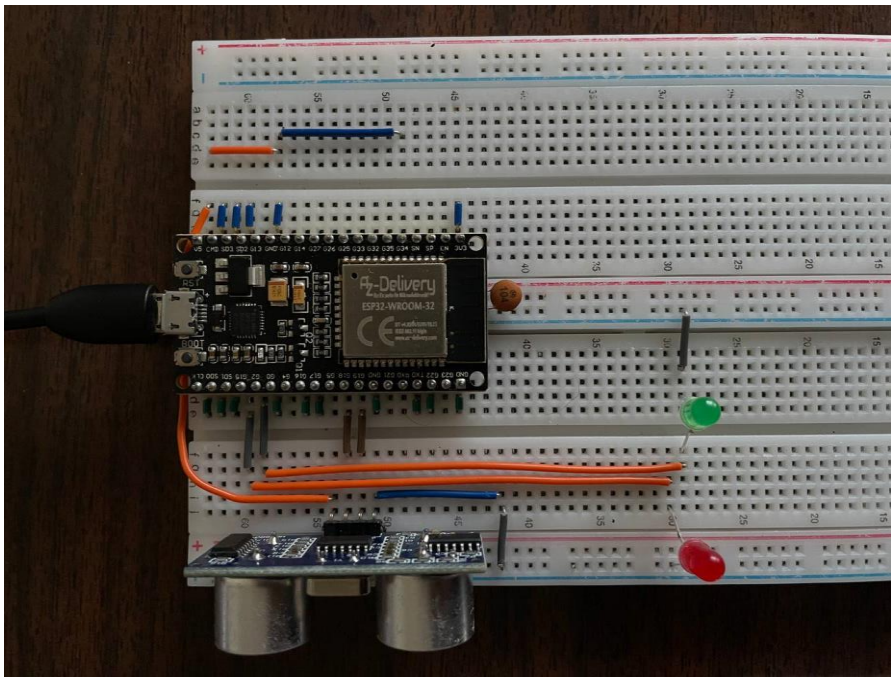
ESP32 è un microcontrollore a 32 bit realizzato dall'azienda cinese "Espressif Systems". È dotato di un processore dual – core a 240 MHz, memoria flash integrata, connessione Wifi, bluetooth, bluetooth Low energy e una vasta gamma di periferiche per l'input/output. È stato sviluppato come successore del popolare microcontrollore ESP8266, con un aumento delle prestazioni e delle funzionalità.

Questo microcontrollore è ampiamente utilizzato nell'area dell' internet of things (IoT), per realizzare progetti domestici, oppure per sensori di monitoraggio e molto altro ancora. La flessibilità di ESP32, combinata con la suo basso consumo per l'elaborazione, lo rende adatto per la realizzazione di molte applicazioni, dal controllo di dispositivi a basso consumo energetico, all'elaborazione di segnali complessi e l'interfacciamento con servizi cloud.

Realizzazione circuitale e lista dei componenti

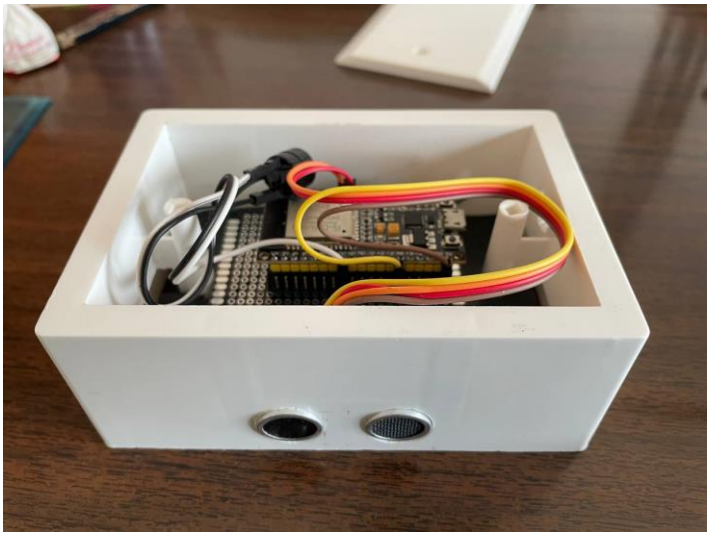
- ESP32 microcontroller SoC (System on Chip)
- Sensore di profondità HC – SR04
- Buzzer(cicalino) YXDZ

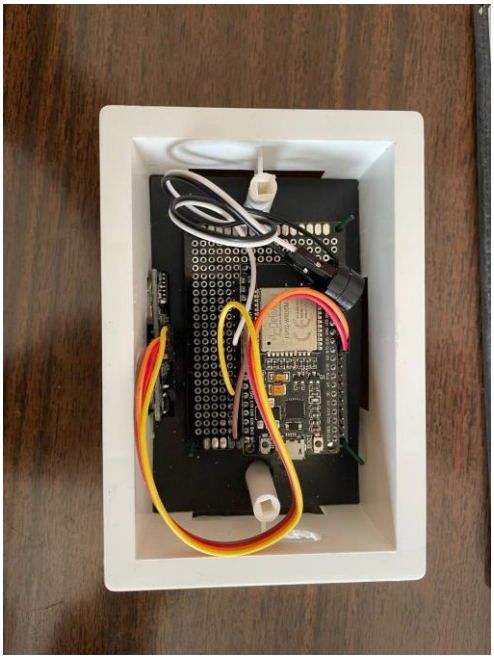




Elenco lavorazioni

- Forature e filettature
- Saldatura a stagno su PCB
- Smussature bordi





Sensore HC – SR04

Il sensore HC-SR04 è un dispositivo di misurazione della distanza che utilizza gli ultrasuoni. Per capire come funziona, è utile sapere che gli ultrasuoni sono onde sonore ad alta frequenza, che si propagano nell'aria a una velocità di circa 340 metri al secondo. Il sensore HC-SR04 funziona inviando un impulso di ultrasuoni, che viene emesso dal trasmettitore del sensore. Questo impulso viaggia attraverso l'aria fino a incontrare un oggetto, che lo riflette. L'impulso riflesso viene poi rilevato dal ricevitore del sensore.

Il tempo impiegato per l'impulso ad andare dall'emettitore all'oggetto e ritornare al ricevitore viene quindi misurato dal sensore HC-SR04. Questo tempo di volo viene poi utilizzato per calcolare la distanza tra l'oggetto e il sensore, utilizzando la formula:

$$distanza = \frac{tempo\ di\ volo * velocità\ del\ suono}{2}$$

Dove la velocità del suono nell'aria viene generalmente considerata pari a circa 340 m/s.

Per rendere la misura più accurata, il sensore HC-SR04 di solito invia diversi impulsi di ultrasuoni e ne calcola la media dei tempi di volo. Il sensore dispone inoltre di un circuito integrato che si occupa di gestire l'emissione e la ricezione degli impulsi di ultrasuoni e di convertire il tempo di volo in una misura di distanza.

Buzzer(cicalino)

In generale, un buzzer è un dispositivo elettronico che produce un suono quando viene attivato. Esistono diverse tipologie di buzzer, tra cui quelli piezoelettrici e quelli elettromagnetici.

Il buzzer piezoelettrico è costituito da una lamina di materiale piezoelettrico che vibra quando viene applicata una corrente elettrica ad essa. Questa vibrazione produce un suono ad una frequenza determinata dal design del buzzer. I buzzer piezoelettrici sono molto utilizzati in applicazioni di segnalazione acustica, come ad esempio nei sistemi di allarme, nei giocattoli e nei dispositivi di avviso sonoro.

Il buzzer elettromagnetico, invece, utilizza un'armatura metallica che viene attratta da un magnete quando una corrente elettrica viene applicata alla bobina del magnete. Questo movimento produce un suono ad una determinata frequenza.

Codice sorgente

In seguito, il codice scritto per programmare la scheda AZ – Delivery ESP32, basata sul modulo ESP32 – WROOM. Il codice è stato compattato utilizzando dei pattern per delineare i componenti indipendenti tra loro; in particolare si ha:

- **void loop()** e **void setup()** che sono funzioni obbligatorie per la realizzazione di un programma in Arduino; *setup()* è una funzione che serve per configurare le variabili dichiarate; ad esempio, se ho un led sul pin 13, dichiaro int led = 13 e nel *setup()* configuro il pin 13 come un output per vedere il segnale in uscita sul led; *void loop()* invece è una funzione che, come suggerisce il nome, continuerà in loop la porzione di codice scritta in essa. In questo progetto il *loop()* prevede un collegamento tra un dispositivo android e il modulo bluetooth dell'ESP32 (tramite la funzione **void bluetooth()**). Per la comunicazione su android, ho installato l'applicazione "*Serial Bluetooth Terminal*". Quando il dispositivo android verrà associato, il loop attenderà un input specifico da parte del dispositivo stesso, che è il comando denominato "controllo_ON"; il comando fa partire un ciclo "do – while" presente nella funzione sensor().
- **void sensor()**, funzione che gestisce il sensore HC – SR04, calcolando ciclicamente la distanza tra il sensore e l'oggetto riscontrato dall'ultrasuoni. Il ciclo si interrompe se il sensore legge una distanza che non è compresa tra SOGLIA_MAX e SOGLIA_MIN.
- **void buzzAlarm()**,è la funzione che gestisce l'allarme, azionando il buzzer.

```
#include<stdio.h>
#include<stdlib.h>
#include <esp_system.h>
#include "BluetoothSerial.h"

#define SOUND_SPEED 0.034 //velocita' del suono approssimata, utilizzata nel
sensore HC - SR04 per il calcolo della distanza
#define MAX_VOL 20
#define SOGLIA_MIN 4
#define SOGLIA_MAX 16

//definizione delle variabili
BluetoothSerial SerialBT;
const int triggerPin = 18, echoPin = 19, buzzerPin = 33;
double distance, livello;
unsigned long duration;
String input = "";

void setup() {
  SerialBT.begin("ESP32 AHIR");
  Serial.begin(9600);

  pinMode(triggerPin, OUTPUT);
  pinMode(echoPin, OUTPUT);
  pinMode(buzzerPin,OUTPUT);
}

void loop() {

  bluetooth();
  if(input == "controllo_ON") {
      sensor();
  }
}

//inizializzazione bluetooth

void bluetooth() {
  SerialBT.connect();
  if(SerialBT.available()){  //Se c'è qualche informazione dalla linea
seriale...
    char incomingChar = SerialBT.read();

    if (incomingChar != '\n'){
      input += String(incomingChar); //cast da carattere a stringa, la
variabile input diventa una sommatoria di caratteri.
    }else{
      input = "";
    }
    Serial.write(incomingChar);  //verifica: scrittura su seriale del comando
dato in input
  }
}

//gestione allarme
```

```
void buzzAlarm() {
   for(int i = 0; i < 4; i++) {
    digitalWrite(buzzerPin,HIGH);
    delay(50);

    digitalWrite(buzzerPin,LOW);
    delay(25);
  }
}

//gestore sensore
void sensor() {
  do {
    digitalWrite(triggerPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(triggerPin,LOW);

    duration = pulseIn(echoPin,HIGH); //conta il tempo nel quale il pin al
primo parametro si trova nello stato nel secondo parametro
    distance = duration * SOUND_SPEED / 2;
    livello = MAX_VOL - distance; // livello = differenza tra distanza
misurata e il totale del volume del serbatoio

    SerialBT.print("RUNNING:  ");
    SerialBT.print(livello);
    SerialBT.println(" cm.");

    Serial.print("RUNNING:  ");
    Serial.print(livello);
    Serial.println(" cm.");

    delay(500);
  }while(livello < SOGLIA_MAX && livello > SOGLIA_MIN);

  if(livello >= SOGLIA_MAX) { //se DISTANZA raggiunge il valore massimo di
soglia impostato, allora il serbatoio è PIENO.
    SerialBT.println("WARNING: SERBATOIO PIENO");
    Serial.println("WARNING: SERBATOIO PIENO");
    buzzAlarm();
    } else if(livello <= SOGLIA_MIN){ //se DISTANZA oltrepassa una SOGLIA
minima impostata, allora il serbatoio è IN ESAURIMENTO

      SerialBT.println("WARNING: LIQUIDO IN ESAURIMENTO");
      Serial.println("WARNING: LIQUIDO IN ESAURIMENTO");
      buzzAlarm();
    }
}
```

Approfondimento sui Timer di arduino e ISR (Interrupt Service Routine)

Nella realizzazione di questo progetto, una parte critica è stata uscire da una funzione loop. Avevo valutato di utilizzare un costrutto di arduino per l'utilizzo dei timer per implementare la funzione di interrupt hardware del microcontrollore.

I timer di Arduino sono componenti hardware che possono essere utilizzati per contare il tempo o per generare segnali di clock. I microcontrollori di Arduino hanno almeno un timer integrato, mentre alcuni modelli ne hanno anche più di uno. I timer di Arduino possono essere configurati per generare un interrupt a intervalli regolari, ovvero ogni volta che il timer raggiunge un valore specifico. In questo modo, è possibile creare funzioni che vengono eseguite ad intervalli di tempo specifici. Ad esempio, è possibile utilizzare un timer per accendere e spegnere un led ad una frequenza specifica, oppure per campionare un segnale analogico a determinati intervalli di tempo.

Per utilizzare i timer di Arduino è possibile utilizzare la funzione "**millis()**" che restituisce il numero di millisecondi trascorsi dall'avvio del microcontrollore. Questo valore può essere utilizzato per impostare un timer interno e gestire eventi temporizzati. La funzione "**delay()**" è un altro modo per gestire il tempo in Arduino, ma questa blocca l'esecuzione del codice per un certo periodo di tempo, mentre la funzione millis() consente di eseguire altre attività nel frattempo.

Il comando **attachInterrupt()** è una funzione di Arduino che consente di attivare un'interfaccia di interrupt hardware, che permette di interrompere immediatamente l'esecuzione del programma principale e di eseguire una funzione definita dall'utente quando si verifica un evento specifico.

L'interfaccia di interrupt hardware può essere utilizzata, ad esempio, per rispondere a un segnale di un sensore o a un input esterno, come un pulsante o un segnale proveniente da un altro microcontrollore. Quando si verifica l'evento specifico (ad esempio, quando il pulsante viene premuto), l'interfaccia di interrupt hardware genera un segnale di interrupt che interrompe immediatamente il flusso di esecuzione del programma principale e avvia l'esecuzione della funzione definita dall'utente.

La funzione attachInterrupt() ha tre parametri in ingresso: il primo è il numero del pin di interrupt che si vuole utilizzare, il secondo è la funzione che si vuole eseguire quando si verifica l'interrupt, e il terzo indica il tipo di interrupt che si vuole utilizzare (ad esempio, se si vuole attivare l'interrupt su un fronte di salita o di discesa del segnale).

Interrupt Service Routine (ISR) è una funzione di gestione degli interrupt che viene eseguita immediatamente dopo l'arrivo di un segnale di interrupt generato da un dispositivo hardware, come un sensore, un pulsante, una scheda di rete o un timer. L'ISR viene utilizzata per interrompere momentaneamente il flusso di esecuzione del programma principale e gestire l'evento che ha generato l'interrupt.

L'ISR deve essere scritta in modo efficiente e veloce, poiché il microcontrollore deve tornare al più presto possibile alla normale esecuzione del programma principale. L'ISR deve anche essere scritta con una particolare attenzione alla sicurezza, poiché l'esecuzione del codice all'interno dell'ISR può causare problemi se non gestita correttamente.

Il parametro "mode" della funzione attachInterrupt() in Arduino specifica il tipo di transizione del segnale di input sul pin di interrupt che deve attivare l'ISR (Interrupt Service Routine), ovvero la funzione che viene eseguita quando l'interrupt si verifica.

Ci sono tre tipi di mode possibili: CHANGE, FALLING e RISING.

- **CHANGE**: attiva l'ISR quando il segnale di input sul pin di interrupt cambia di stato, ovvero da HIGH a LOW o viceversa.
- **FALLING**: attiva l'ISR quando il segnale di input sul pin di interrupt passa da HIGH a LOW.
- **RISING**: attiva l'ISR quando il segnale di input sul pin di interrupt passa da LOW a HIGH.

Ad esempio, se si utilizza il parametro RISING, l'ISR verrà eseguita quando il segnale di input sul pin di interrupt passa da LOW a HIGH. Questo può essere utile, ad esempio, per gestire un pulsante: quando il pulsante viene premuto, il segnale sul pin di interrupt passa da LOW a HIGH e l'ISR viene eseguita. Invece, utilizzando FALLING, l'ISR verrebbe eseguita quando il pulsante viene rilasciato, ovvero quando il segnale passa da HIGH a LOW.

Codice sperimentale

```
volatile unsigned long duration;
volatile bool measurementComplete = false;
void onEchoReceived() {
  digitalWrite(buzzerPin,HIGH);
  delay(100);
  digitalWrite(buzzerPin,LOW);
  delay(50);
}
void setup() {
  Serial.begin(9600);
  pinMode(triggerPin, OUTPUT);
  pinMode(echoPin, INPUT);
  attachInterrupt(digitalPinToInterrupt(echoPin), onEchoReceived, RISING);
//parte critica...
}


[…]


//Interrupt service routine
ISR(digitalPinToInterrupt(echoPin)) {
  if (digitalRead(echoPin) == HIGH) {
    duration = micros() - duration;
    measurementComplete = true;
  } else {
    duration = micros();
  }
}
```

BIBLIOGRAFIA

- Bluetooth Serial | Ionic Documentation: https://ionicframework.com/docs/v5/native/bluetooth-serial
- HC – SR04 Ultrasonic sensor working, pinouts & features: https://components101.com/sensors/ultrasonic-sensor-working-pinout-datasheet
- ESP32 Documentation about interrupts allocation: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/intr_alloc.html
- Modalità deep sleep con ESP32: https://it.emcelettronica.com/modalita-deep-sleep-con-esp32