Corso di Progettazione di Sistemi Operativi

Federico Germinario
Giacomo Papaluca

Introduzione

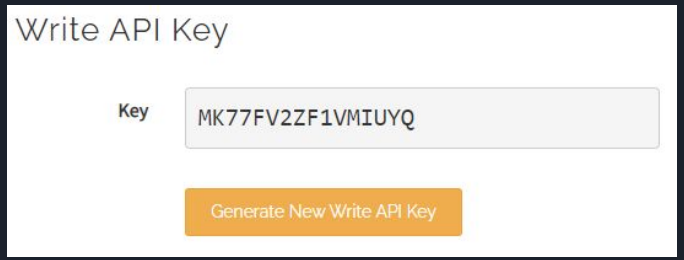
- Gran parte dei sistemi IoT al giorno d'oggi mira a garantire alle persone uno stile di vita più agiato e più salutare.

- L'obiettivo che ci siamo posti per questo progetto è stato quello di creare un dispositivo che raccolga dei dati riguardanti la qualità dell'aria e del manto stradale ed invii questi dati ad un cloud, che mostra in ogni momento il loro andamento.

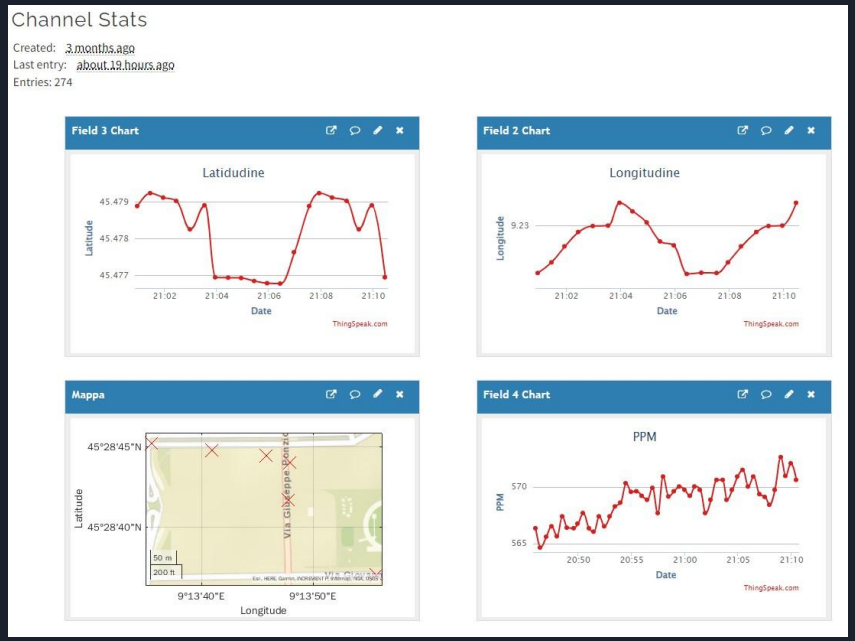
- L'idea è quindi quella di creare un sistema che, raccogliendo le informazioni sopracitate in tempo reale, possa fornire agli utenti la possibilità di scegliere il tragitto più piacevole possibile per spostarsi fra le strade di una città.

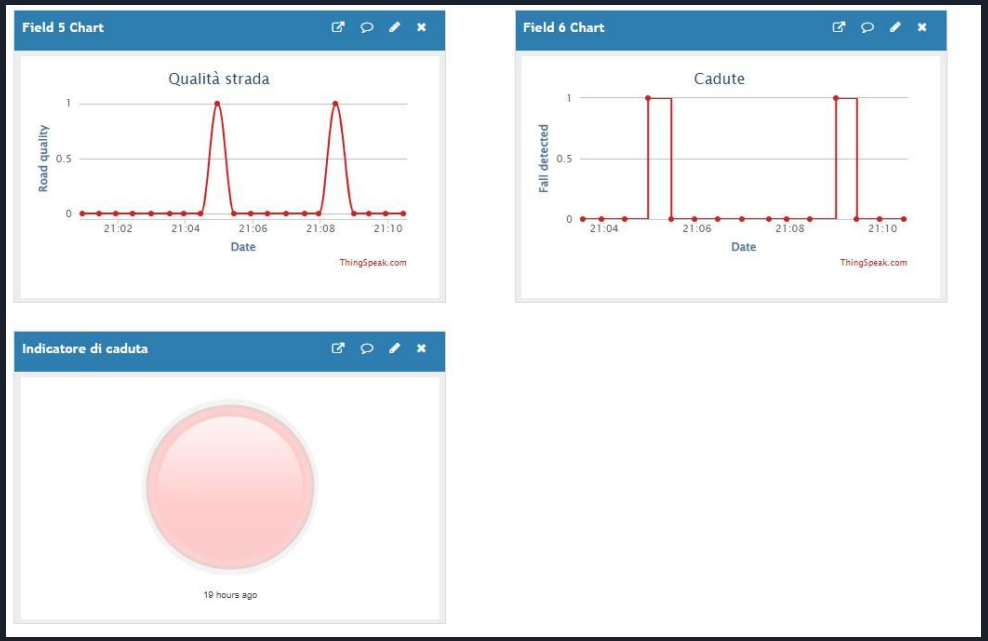
Cloud (ThingSpeak)

- ThingSpeak è una piattaforma open source che consente di aggregare, visualizzare e analizzare flussi di dati nel cloud provenienti da dispositivi IoT.
- I dati vengono archiviati in canali che possono essere pubblici o privati
- Ogni canale può contenere al massimo 8 serie di dati
- `GET https://api.thingspeak.com/update?api_key=MK77FV2ZF1VMIUYQ&field1=0`

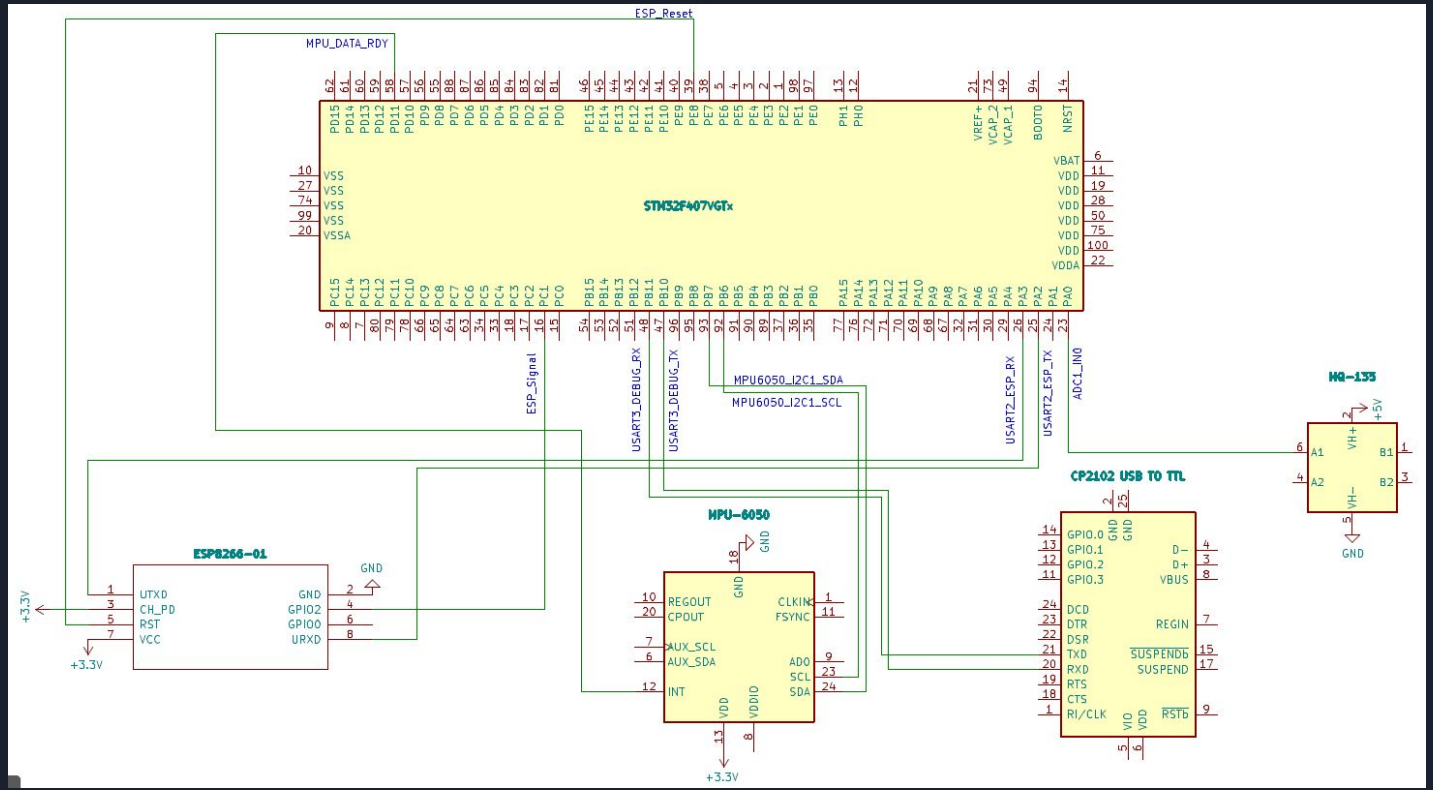


ThingSpeak

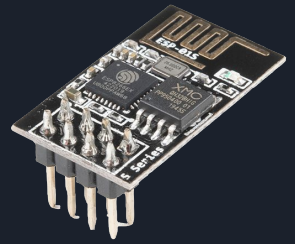

Channel Stats
Created: 3 months ago
Last entry: about 19 hours ago
Entries: 274
Field 3 Chart
Latidudine
Latitude
45.479
45.478
45.477
21:02
21:04
21:06
21:08
21:10
Date
ThingSpeak.com
Field 2 Chart
Longitudine
Longitude
9.23
21:02
21:04
21:06
21:08
21:10
Date
ThingSpeak.com
Mappa
45°28'45"N
45°28'40"N
Latitude
50 m
200 ft
9°13'40"E
9°13'50"E
Longitude
Field 4 Chart
PPM
PPM
570
565
20:50
20:55
21:00
21:05
21:10
Date
ThingSpeak.com


Field 5 Chart
Qualità strada
Road quality
1
0.5
0
21:02
21:04
21:06
21:08
21:10
Date
ThingSpeak.com
Field 6 Chart
Cadute
Fall detected
1
0.5
0
21:04
21:06
21:08
21:10
Date
ThingSpeak.com
Indicatore di caduta
19 hours ago

Architettura del sistema



ESP8266-01



- l’ESP8266 è un chip con WiFi integrato a basso costo
- Ha un supporto completo al protocollo TCP/IP
- Il modulo viene distribuito con un firmware ufficiale che permette di pilotarlo e gestirlo attraverso dei comandi AT
- Abbiamo scelto di riprogrammare il dispositivo in modo da adattarlo ai nostri bisogni

Caratteristiche tecniche:

- Processore: RISC a 32 bit (80/160 Mhz)
- 64 KByte di RAM per le istruzioni
- 96 KByte di RAM dati
- UART su pin dedicati
- 2 PIN GPIO

Ciclo di vita dell'ESP8266

1. L'STM32 raccoglie i dati di interesse
    1.1. nel frattempo la scheda di rete cerca di connettersi ad un hotspot e una volta connetto invia un segnale all'STM32
2. Il segnale giunge all'STM32 e invia tutti i dati all'ESP
3. L'ESP invia i dati a ThingSpeak e va in deep sleep
4. Alla scadenza di un timer da 30 secondi l'STM32 invia un segnale di Wake Up alla scheda di rete e il ciclo viene ripetuto

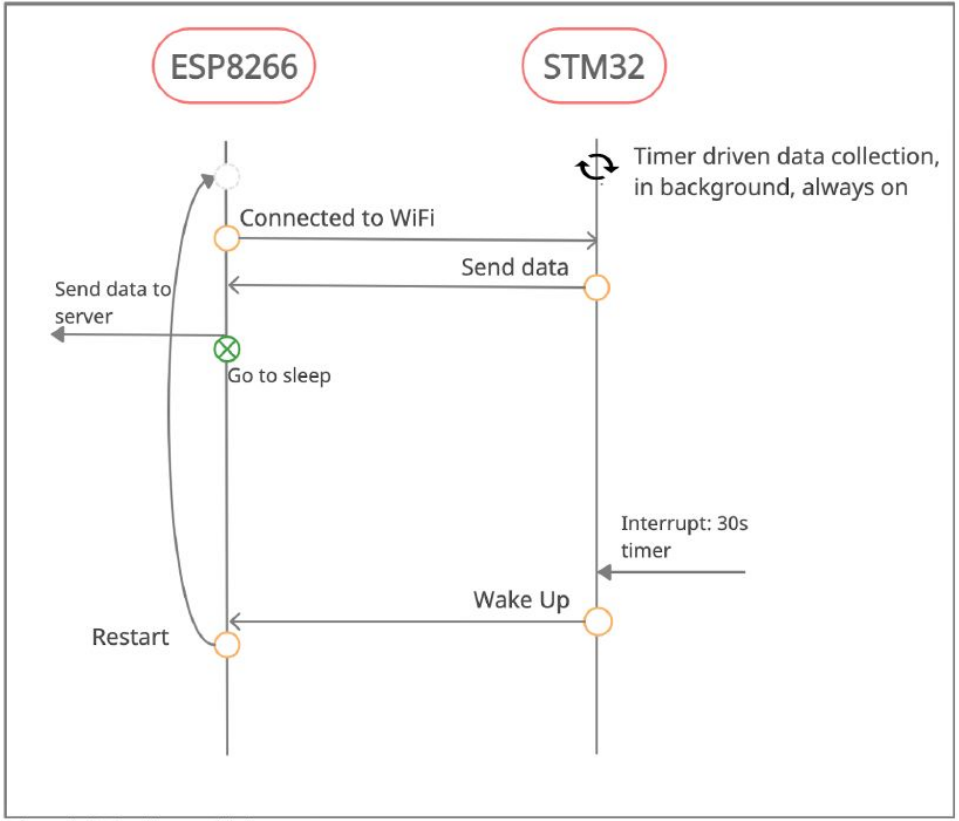


figura 1: timing diagram del sistema

Codice ESP8266

```
void setup() {
  Serial.begin(115200);
  WiFi.disconnect();
  EEPROM.begin(512);                //Inizializzazione EEPROM
  delay(10);

  pinMode(LED_BUILTIN, OUTPUT);     //Inizializzazione LED_BUILTIN come pinout
  pinMode(2, OUTPUT);               //Inizializzazione GPIO2 come pinout

  digitalWrite(LED_BUILTIN, HIGH);  //Led spento
  digitalWrite(2, LOW);

  Serial.setTimeout(1000);          //Aspetto max 1 sec una stringa sulla seriale

  String esid;
  for (int i = 0; i < 32; ++i){
    esid += char(EEPROM.read(i));
  }

  String epass = "";
  for (int i = 32; i < 96; ++i){
    epass += char(EEPROM.read(i));
  }

  WiFi.mode(WIFI_STA);               //Solo client
  ThingSpeak.begin(client);          //Inizializzazione ThingSpeak
  connect_to_wifi(esid.c_str(), epass.c_str());
}
```

```
void connect_to_wifi(String ssid, String pass) {
  WiFi.begin(ssid, pass); //Connessione alla rete WiFi
  delay(1000);
  if (testWifi()){
    digitalWrite(LED_BUILTIN, LOW); //Accendi led

    digitalWrite(2, HIGH); //Invia un segnale alto su GPIO2 per 50 ms
    delay(50);
    digitalWrite(2, LOW);
  }else{
    // La connessione alla rete WiFi è fallitta
    // Accensione Hotspot WiFi
    setupAP();

    while ((WiFi.status() != WL_CONNECTED)){
      delay(20);
      server.handleClient();
    }

    digitalWrite(LED_BUILTIN, LOW); //Accendi led
    digitalWrite(2, HIGH);          //Invia un segnale alto su GPIO2 per 50 ms
    delay(50);
    digitalWrite(2, LOW);
  }
}
```

Codice ESP8266

```
void loop() {
  read_serial_packet();
  if(data_available){
    send_to_thingsspeak();
    data_available = false;
  }
}

void read_serial_packet() {
  if(Serial.available()) {
    data_available = true;
    currentLine = Serial.readStringUntil('\n');

    int commaSplitIndex = currentLine.indexOf(',');
    if (commaSplitIndex > 0) {
      String longitudeStr = currentLine.substring(0, commaSplitIndex);
      currentLine = currentLine.substring(commaSplitIndex + 1);
      commaSplitIndex = currentLine.indexOf(',');
      String latitudeStr = currentLine.substring(0, commaSplitIndex);
      currentLine = currentLine.substring(commaSplitIndex + 1);
      commaSplitIndex = currentLine.indexOf(',');
      String ppmStr = currentLine.substring(0, commaSplitIndex);
      currentLine = currentLine.substring(commaSplitIndex + 1);
      commaSplitIndex = currentLine.indexOf(',');
      String roadQualityStr = currentLine.substring(0, commaSplitIndex);
      String fallDetectedStr = currentLine.substring(commaSplitIndex + 1);

      lastLongitude = longitudeStr.toFloat();
      lastLatitude = latitudeStr.toFloat();
      lastPpm = ppmStr.toFloat();
      lastRoadQuality = roadQualityStr.toInt();
      lastFallDetected = fallDetectedStr.toInt();
    }
  }
}
```

```
void send_to_thingsspeak() {
  ThingSpeak.setField(2, lastLongitude);
  ThingSpeak.setField(3, lastLatitude);
  ThingSpeak.setField(4, lastPpm);
  ThingSpeak.setField(5, lastRoadQuality);
  ThingSpeak.setField(6, lastFallDetected);

  //Invio i dati a ThingSpeak tramite chiamata REST HTTPS
  ThingSpeak.writeFields(myChannelNumber, myWriteAPIKey);

  //Attivo la modalità deep sleep finché non riceverò un hard reset
  ESP.deepSleep(0);
}
```

Item	Modem-sleep	Light-sleep	Deep-sleep
Wi-Fi	OFF	OFF	OFF
System clock	ON	OFF	OFF
RTC	ON	ON	ON
CPU	ON	Pending	OFF
Substrate current	15 mA	0.4 mA	~20 uA

Fonte: https://randomnerdtutorials.com/esp8266-deep-sleep-with-arduino-ide/

STM32F407VGT6-DISC1

Centro nevralgico del sistema:

- Inizializza tutti i componenti periferici e li controlla con un paradigma timer driven:
  - gestione ciclo di vita ESP-01 con reset ogni 30s ed ESP Signal
  - trasferimenti in RAM da MPU6050 con I2C DMA ogni 8s (road quality)
  - ADC1 ch0 su TRGO di TIM3 per leggere dati ogni 5s da MQ135 (air quality)
- Colleziona, elabora ed infine invia al cloud i dati rilevati
- Utilizza SleepOnExit per risparmio energetico

Caratteristiche tecniche:

- Processore: 32-bit Arm® Cortex®-M4 with FPU core (168 MHz max, usato a 50 MHz)
- 192 KByte di RAM
- 1 Mbyte Flash memory

STM32F407 (Main)

```
/* Initialize all configured peripherals */
MX_GPIO_Init();

MX_USART2_UART_Init();   //UART used to communicate with the ESP8266
MX_TIM2_Init();          //30 sec timer to wake up ESP8266
MX_USART3_UART_Init();   //Debugging UART
MX_I2C1_Init();          //I2C to communicate with MPU-6050
MX_ADC1_Init();          //ADC1 used to convert signals from the MQ-135
MX_DMA_Init();           //Dma to transfer inertial data into the STM32 SRAM
MX_TIM3_Init();          //5 sec timer to start ADC1 conversion
MX_TIM4_Init();          //8.1 sec timer for reading inertial data from MPU-6050
/* USER CODE BEGIN 2 */
RetargetInit(&huart3);   //Retarget printf to uart3

fake_gps_init();

MPU6050_Init();
HAL_DMA_Init(&hdma_i2c1_rx);

HAL_TIM_Base_Start(&htim3);
HAL_ADC_Start_IT(&hadc1);

HAL_NVIC_DisableIRQ(EXTI1_IRQn);
__HAL_TIM_CLEAR_FLAG(&htim2, TIM_SR_UIF); //Clear update interrupt flag
HAL_TIM_Base_Start_IT(&htim2);            //Start timer 30 s
```

```
  HAL_SuspendTick(); //The SysTick interrupt is disabled and then the Tick increment is suspended

  /*
   * Set SLEEPONEXIT bit of SCR register. When this bit is set, the processor re-enters SLEEP mode
   * when an interruption handling is over.
   */
  HAL_PWR_EnableSleepOnExit();

  HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1){
    /* USER CODE END WHILE */

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

TIM 2

```
htim2.Instance = TIM2;
htim2.Init.Prescaler = 24999;     //25MHz/25000 = 1000 Hz
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
htim2.Init.Period = 29999;        //1000Hz / 30000 = 0.033Hz = 30s
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
```

UpdateEvent = Timer_clock /((Prescaler + 1) * (Period + 1))
= 25 MHz / ((24999 + 1) * (29999 + 1))
= 25 MHz / 750 M
= 0,033 Hz = 1/0,033 s ≈ **30 s**

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */
    if (htim->Instance == TIM2) {
        HAL_ResumeTick();
        DEBUG_PRINT(("[TIM2] 30 sec timer expired! SEND RESET TO ESP8266!\r\n\n"));

        // Check MPU
        if(!MPU_OK){
            MPU6050_Init();
        }

        reset_esp8266();

        HAL_SuspendTick();
    }
```

Problema rimbalzi ESP8266

- Dopo l’hard reset il dispositivo invia dei segnali spuri all’STM32
- Questo causava l’attivazione di molte callback lato STM32
- Circuito anti-rimbalzo (filtro passa basso di tipo R-C)
- Problema risolto via software

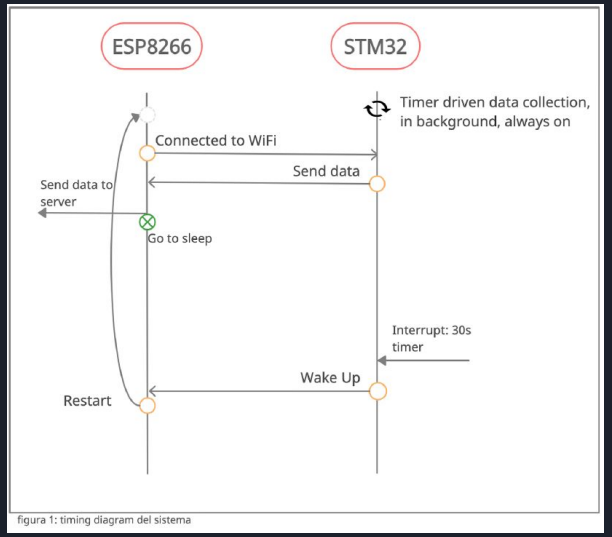
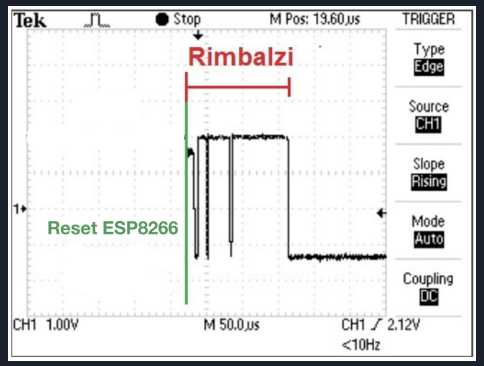


figura 1: timing diagram del sistema



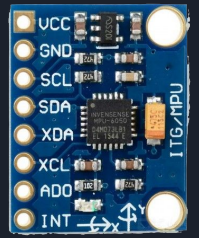
Risoluzione problema rimbalzi ESP8266

- Nel main disattivo le callback dell'EXTI1
- Quando TIM 2 scade viene chiamata la funzione reset_esp8266()
- Quando ricevo il segnale da parte dell'ESP che è riuscito a connettersi ad una rete WiFi (ESP_Signal), aggrego i dati, li invio alla scheda di rete e disattivo le callback dell'EXTI1

```
if (htim->Instance == TIM2) {
    HAL_ResumeTick();
    DEBUG_PRINT(("[TIM2] 30 sec timer "
    "expired! SEND RESET TO ESP8266!\r\n\n"));

    // Check MPU
    if(!MPU_OK){
        MPU6050_Init();
    }

    reset_esp8266();
    HAL_SuspendTick();
}
```

```
void reset_esp8266(){
    HAL_NVIC_DisableIRQ(EXTI1_IRQn);

    HAL_GPIO_WritePin(ESP_Reset_GPIO_Port, ESP_Reset_Pin, GPIO_PIN_RESET);
    HAL_Delay(20);
    HAL_GPIO_WritePin(ESP_Reset_GPIO_Port, ESP_Reset_Pin, GPIO_PIN_SET);
    HAL_Delay(200);

    HAL_NVIC_ClearPendingIRQ(EXTI1_IRQn); //Clears the pending bit of an EXTI1 interrupt.
    __HAL_GPIO_EXTI_CLEAR_IT(ESP_Signal_Pin);

    HAL_NVIC_EnableIRQ(EXTI1_IRQn);
}
```

MPU-6050



- Microprocessore dotato di vari MEMS e ADC
- Usato per raccogliere dati sulla qualità del manto stradale
- 3-axis accelerometer e 3-axis gyroscope (fs: 1 KHz)
- Interrupt pin per comunicare l’avvio delle misurazioni
- Buffer fifo da 1024 byte per accumulare misurazioni (12 byte per ogni misurazione accel+gyro)
- Prescaler di /100 per ridurre il data rate a 10Hz e ridurre quindi i consumi
- Parametri settati attraverso scrittura dei registri: HAL_I2C_Mem_Write() da STM32

Inizializzazione MPU-6050

```c
void MPU6050_Init(){
    uint8_t check, Data;
    uint8_t attempts = 0;

    // Check if the device is ready
    while(check != 104){
        HAL_I2C_Mem_Read(&hi2c1, MPU6050_ADDR, WHO_AM_I_REG, 1, &check, 1, 1000);
        if(attempts++ > 50){
            break;
        }
    }

    if(check == 104){ // If the device is ready

        // We write all zeros in register 0X6B to wake up the sensor
        Data = 0;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, PWR_MGMT_1_REG, 1, &Data, 1, 1000);

        // Let's put Gyro fs at 1KHz
        Data = 0x02;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, CONFIG_REG, 1, &Data, 1, 1000);

        // DATA RATE = Gyroscope Output Rate (1 Khz) / (1 + SMPLRT_DIV (99)) ==> 10 Hz
        Data = 0x63;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, SMPLRT_DIV_REG, 1, &Data, 1, 1000);

        // Accelerometer configuration:
        // XA_ST = 0, YA_ST = 0, ZA_ST = 0, FS_SEL = 0 ==> Full Scale Range = +- 2g
        Data = 0x00;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, ACCEL_CONFIG_REG, 1, &Data, 1, 1000);

        // Gyroscope configuration:
        // XG_ST = 0, YG_ST = 0, ZG_ST = 0, FS_SEL = 0 ==> Full Scale Range = +- 250 */s
        Data = 0x00;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, GYRO_CONFIG_REG, 1, &Data, 1, 1000);

        // Enable write buffers for accel and gyro
        Data = 0x78;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, FIFO_EN_REG, 1, &Data, 1, 1000);

        // Enable the buffer
        Data = 0x44;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, USER_CTRL, 1, &Data, 1, 1000);

        // Enable interrupt with data read
        Data = 0x01;
        HAL_I2C_Mem_Write(&hi2c1, MPU6050_ADDR, INT_ENABLE, 1, &Data, 1, 1000);
    }
}
```

- Scriviamo registri specifici per settare i parametri dell'MPU
- Polling è una buona soluzione perchè sono comunicazioni brevi e puntuali (8 bit payload)

Calibrazione MPU-6050

- Dobbiamo assicurarci che i dati del sensore siano corretti

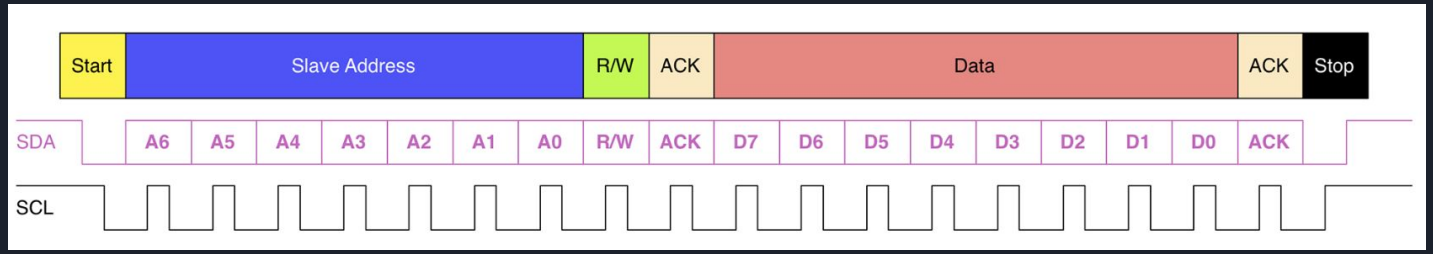
Calibrazione:

- Teniamo fermo l'MPU, appoggiato su una superficie piana
- Leggiamo una serie di valori misurati (qualche centinaia di misurazioni)
- Facciamo una media di quelle misurazioni e salviamo le medie per ogni asse nelle variabili offset_*
- Ad ogni misurazione effettuata dal microprocessore aggiungiamo l'offset in modo che a sensori fermi le misurazioni sugli assi siano il più vicino possibile a 0

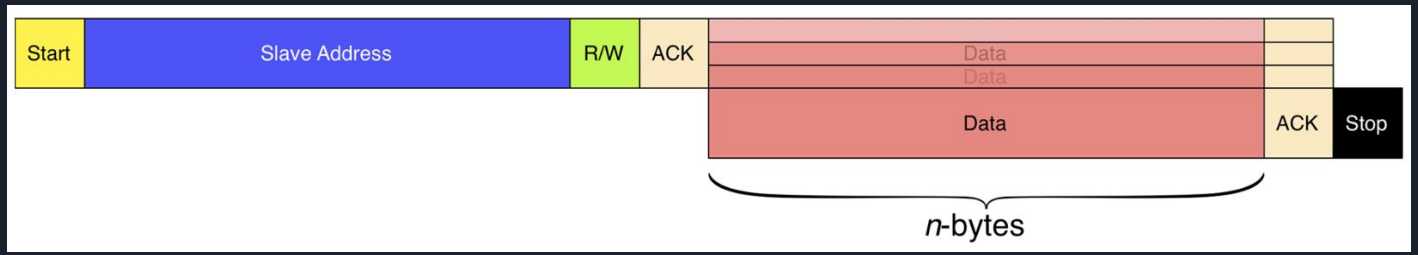
```
//Calibration offsets
float offset_gyroX = 6.725720;
float offset_gyroY = 8.554494;
float offset_gyroZ = 5.479887;

float offset_accelX = 0.889819;
float offset_accelY = 0.694547;
float offset_accelZ = 1 - 0.260066;
```

I2C Protocol

- Comunicazione master-slave (STM32-MPU6050 nel nostro caso)
- 2 linee: SCL (Serial Clock Line) e SDA (Serial Data Line)
- Frame da 8 bit, se indirizzi maggiori, devo usare più frame per address



- Burst mode transmission: master invia slave address, mantiene il segnale di clock e riceve/invia su SDA i dati dallo slave

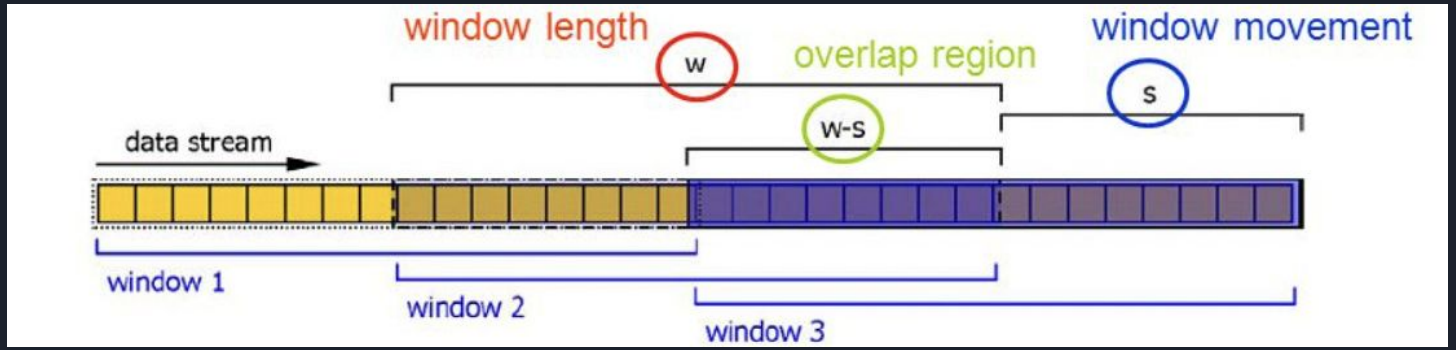


MPU-6050 DATA RETRIEVING

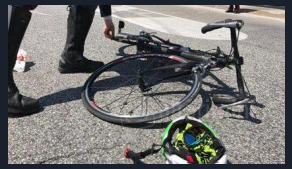
- MPU FIFO buffer (1024 bytes) abilitato per salvare dati inerziali
- Al termine della prima misurazione manda interrupt a STM32 che fa partire TIM4 (8s)
- TIM4 period elapsed: sospendo momentaneamente la scrittura del buffer per poterlo leggere e avvio la richiesta di trasmissione master-slave I2C
    - invio registro FIFO_RW allo slave per dire che voglio leggere il fifo
    - leggo in burst mode tutti i byte del fifo
- Trasmissione effettuata da DMA da peripheral a memory => grossa mole di dati, risparmiamo CPU time

MPU-6050 DATA PROCESSING

- Metodo sliding windows per assicurarsi di non perdere informazioni
- Misurazioni suddivise in finestre di 20 misurazioni, 50% overlap
- Tipicamente 960 byte, 80 misurazioni per accel e 80 per gyro ==> 7 finestre per ogni burst di dati
- Overlap ci permette di evitare che un evento che accade tra 2 finestre non venga saltato

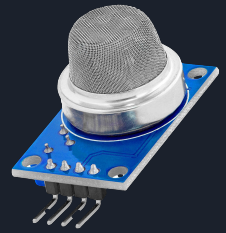


MPU-6050 DATA PROCESSING (2)



- Per ogni finestra di misurazioni:
  - calcoliamo vettore risultante di componenti x,y e z di accelerometro e giroscopio
  - salviamo i vettori in array ausiliari e calcoliamo le medie delle intensità delle accelerazioni lineari
- Algoritmo per qualità della strada:
  - una soglia è calcolata come inversamente proporzionale all'intensità dell'accelerazione angolare
  - conto quante volte in ogni finestra un vettore di accelerazione lineare supera in intensità la media della finestra di un valore pari o maggiore alla soglia
  - un manto stradale pessimo produrrà molte vibrazioni => verranno contate molte misurazioni
  - un manto stradale in buone condizioni produce poche vibrazioni => il conteggio sarà prossimo allo 0
- Rilevazione cadute:
  - riconosco una serie di misurazioni ad alta intensità di vibrazione
  - attendo un periodo di tempo (60 misurazioni, da testare in un contesto vero)
  - se rilevo stabilità nel device (utente fermo) suppongo sia caduto

MQ-135



- Semplice componente passivo dotato di una resistenza variabile
- La resistenza varia a seconda della concentrazione di molecole di sostanze nell'aria
- La tensione in uscita cresce al crescere di questa concentrazione (aria meno pura)
- Calibrato per la Co2:
  - posizionato in un ambiente con ppm di Co2 noto (fonte)
  - misurata tensione in uscita con voltmetro

- Sapendo che a 400 ppm l'output è di 320 mV
- Misurando la vcc come 4.668 V (non esattamente 5V)
- Usando un ADC con risoluzione 12 bit
- La formula a fianco calcola la traduzione da tensione a ppm

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef*hadc){
    uint16_t rawValue; float ppm; float v;

    rawValue = HAL_ADC_GetValue(hadc);

    v = ((float)rawValue) / 4095 * 4668;
    ppm = ((v - 320.0) / 0.65) + 400;

    DEBUG_PRINT(("[HAL_ADC_ConvCpltCallback] PPM: %f\r\n\n", ppm));
    mq_data[mq_index] = ppm;
    mq_index = (mq_index + 1) % MQ_DATA_LENGTH;
}
```

MQ-135 DATA GATHERING AND PROCESSING

- La lettura del sensore viene effettuata ad ogni overflow del timer TIM3 (5s)
- Per evitare interrupt inutili:
  - l'interrupt di TIM3 è disattivato nell'NVIC
  - l'ADC è settato in modalità timer driven attraverso la TRGO line di TIM3
  - conversione effettuata in automatico ad ogni overflow
  - a conversione terminata conv completed callback calcola il valore di inquinamento
- Le letture vengono salvate in un array
- Quando i dati vengono inviati al cloud invio una media delle letture nell'array

Simulatore GPS

- Tutte le misurazioni effettuate devono avvalersi di una posizione GPS così da far corrispondere i valori acquisiti a un determinato punto sulla mappa.
- Abbiamo raccolto un insieme di coordinate GPS simulando un percorso
- Coordinate scaricate tramite Google Earth
- Tutte le coordinate (latitudine e longitudine) sono state memorizzate in un array circolare
- Ogni qual volta la scheda ESP8266 è pronta ad inviare i dati al server, viene selezionata una coordinata dell'array e viene inviata ad essa insieme a tutti gli altri dati calcolati.

Analisi dei tempi di esecuzione

- Applicazione ciclica, timer driven
- Non sarebbe rilevante calcolare un unico tempo di esecuzione
- Abbiamo calcolato i tempi di esecuzione delle callback
- Il collo di bottiglia è la callback HAL_I2C_MasterRxCpltCallback
- Registro CYCCNT dell'unità DWT

```
TIME (ESP_Signal) =  28.27 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  148.77 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  144.38 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  144.64 ms
TIME (tim4 callback) =  10.14 ms
TIME (HAL_I2C_MasterRxCpltCallback) =  144.26 ms
```

Analisi del consumo

- Punto focale del nostro progetto
- Abbiamo evitato l'utilizzo di un real time OS
- Utilizzo solo di APIs HAL
- Deep Sleep ESP8266 (20 uA vs 50 mA in running mode)
- STM32: *HAL_PWR_EnableSleepOnExit()*
- Utilizzando la SleepOnExit abbiamo dimezzato il consumo medio della scheda
- Siamo passati da circa 20 mA a 10 mA di corrente dissipata

Analisi del consumo

- ESP8266 lavora in run mode (50 mA) per circa 2 secondi e in Deep Sleep (0.02 mA) per circa 28 secondi
- Il consumo medio del dispositivo è di 3.3 mA
- MPU-6050 ha un consumo massimo di 3.9 mA
- Il consumo medio del sistema è circa 17.2 mA (10 mA + 3.9 mA + 3.3 mA)
- Con un power bank da 10000 mAh il sistema può essere alimentato per circa 580 ore
- Il collo di bottiglia è sicuramente l'MQ-135 che consuma in media 150 mA
- In commercio sono presenti sensori più costosi che assorbono molto meno

$$P_{AVG} = [D \times P_{ON}] + [(1 - D) \times P_{OFF}]$$

$$D = \frac{T_{ON}}{T_{ON} + T_{OFF}} = duty\ cycle$$

$P_{ON} = on$ power, $P_{OFF} = off$ power

$T_{ON} = on$ time, $T_{OFF} = off$ time

DEMO