UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Sviluppo di un servizio OTA per la programmazione di un sensore ambientale custom

Development of an OTA service for programming a custom environmental sensor

Candidato:

Letizi Maicol

Relatore:

Prof. Gambi Ennio

Correlatore:

Ing. Di Buò Gianluca

Anno Accademico 2019-2020

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Sviluppo di un servizio OTA per la programmazione di un sensore ambientale custom

Development of an OTA service for programming a custom environmental sensor

Candidato:

Letizi Maicol

Relatore:

Prof. Gambi Ennio

Correlatore:

Ing. Di Buò Gianluca

Anno Accademico 2019-2020

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici

INTRODUZIONE

I parametri ambientali come la temperatura e l'umidità circostante hanno una varietà di cambiamenti ogni giorno, che non è facilmente prevedibile dall'uomo perché non si ripete mai esattamente. Le variazioni di un ambiente possono essere dirompenti e costose per gli esseri umani se non se le aspettano e non le controllano adeguatamente per pianificare tali variazioni nel futuro. Si pensi al continuo incremento incontrollato dell'inquinamento atmosferico per esempio, causato dall'evoluzione delle attività industriali. Questo fenomeno ambientale può essere un rischio per la salute dell'uomo e degli ecosistemi. È quindi necessario un sistema automatizzato in grado di misurare tali parametri, senza che l'uomo debba intervenire con l'ausilio di sensori che tengano conto dei costi e della precisione, evitando anche l'ingombro di questi apparecchi. Queste nuove soluzioni sono state rese possibili dal progresso tecnologico. Il mondo infatti si sta muovendo verso la generazione di *Internet of Things* (IoT), che prevede il collegamento di miliardi di dispositivi alla rete Internet. L'IoT ha consentito la creazione di hardware sempre più piccolo, performante ed efficiente in termini di consumo energetico, come le *Wireless Sensor Network*, riuscendo a ridurre i costi, monitorare aree molto più estese, con condizioni ambientali estreme e in punti difficili da raggiungere dall'uomo.

Sono state rivolte attenzioni anche negli ambienti indoor, per studiare gli stili di vita delle persone, infatti la nostra respirazione è una delle prime sorgenti di anidride carbonica. Conoscere la qualità dell'aria indoor è quindi importante perché ha ripercussioni sulla salute causando effetti indesiderati. Parametri importanti da conoscere in questo tipo di ambiente, sono il monossido di carbonio (CO), gli ossidi di azoto (NOx), la formaldeide e i composti organici volatili (VOC). In [1] è approfondito uno studio sul benessere termico igrometrico di un edificio.

Lo scopo di questa tesi è realizzare il firmware di un dispositivo in grado di misurare i parametri ambientali in un ambiente indoor, trasmettere i dati attraverso la rete Internet e ricevere aggiornamenti da remoto via *Over The Air* (OTA). Il dispositivo in questione si chiama *DONET*, è stato progettato dall'azienda *IDEA Soc. Coop.* ed è dotato di sensori per la misura di temperatura, umidità, luminosità, CO_2 e TVOC dell'aria. DONET rappresenta un nodo di una rete di sensori, che fornisce i dati raccolti ad un gateway, per mezzo del quale vengono

salvati in un database e mostrati all'utente attraverso un'applicazione web, raggiungibile da un qualsiasi Internet browser. In questa tesi viene valutata la sostituzione del chip *ESP8266* del dispositivo con la sua versione avanzata *ESP32*, più potente, in grado di sostenere anche l'aggiornamento OTA, un comodo servizio che permette di diffondere nuove versioni del firmware e aggiornare tutti i dispositivi attraverso la rete Internet, evitando di aggiornarli manualmente uno ad uno. Esistono librerie OTA disponibili per caricare i programmi sul chip ESP8266, ma offrono meccanismi di sicurezza limitati, rendendo il sistema vulnerabile ad attacchi informatici. Inoltre, è stato valutato e implementato un cambio del linguaggio di programmazione, preferendo *MicroPython* ad *Arduino (C/C++)*, perché è un linguaggio di alto livello, essendo più veloce dal punto di vista della scrittura e del debug del codice. La tesi è suddivisa nei capitoli mostrati di seguito.

1. **Wireless Sensor Network**. Viene spiegato nel dettaglio come è fatta una WSN, i suoi principali ambiti di utilizzo e il protocollo di comunicazione MQTT adottato per la trasmissione dei dati.

2. **Aggiornamento firmware Over The Air**. Si introduce l'OTA a livello generale ed è spiegato l'algoritmo implementato nel dispositivo DONET, utilizzando un repository *GitHub*.

3. **Architettura e funzionamento del sistema**. Viene fatta una panoramica dell'infrastruttura informatica a cui è connessa DONET, del suo hardware, del broker MQTT e del funzionamento generale del dispositivo.

4. **MicroPython**. Viene mostrato lo studio svolto sul linguaggio di programmazione MicroPython, per quanto riguarda i vantaggi rispetto al C/C++ e la sua installazione in un sistema embedded e in un IDE per la programmazione.

5. **Firmware DONET**. In questo capitolo è spiegata nel dettaglio l'implementazione in MicroPython del firmware di DONET, per la connessione alla rete WiFi locale, la connessione al broker MQTT, il controllo dei sensori di misura, l'invio dei dati e l'aggiornamento OTA.

6. **Risultati Sperimentali**. Vengono mostrati i risultati ottenuti durante l'esecuzione di DONET con i chip ESP8266 ed ESP32.

Capitolo 1

WIRELESS SENSOR NETWORK

In questo primo capitolo sono descritte le caratteristiche generali di una Wireless Sensor Network per la raccolta di dati provenienti da sensori. Si fa un introduzione al suo funzionamento, ai suoi scopi e ai principali settori di applicazione. Viene mostrata la sua architettura e il protocollo di comunicazione implementato per l'invio delle informazioni al punto centrale di gestione.

1.1 Introduzione alle WSN

Una Wireless Sensor Network (WSN) [2, 3] è una rete wireless composta da minuscoli nodi autonomi e interconnessi (anche detti *mote, sensor node, wireless node*), aventi poca RAM e una CPU a basse prestazioni e distribuiti nello spazio, per monitorare condizioni fisiche o ambientali, quali temperatura, suono, vibrazione, pressione, movimento, ecc. I nodi infatti, sono capaci di ospitare sensori dotati di dispositivi di rilevamento ed elaborazione, che possono comunicare tramite segnali radio e hanno una velocità di elaborazione, capacità di archiviazione e larghezza di banda di comunicazione limitate. I sensori raccolgono i dati e attraverso i nodi vengono inviati periodicamente ad un gateway (o sink) centrale che gestisce la rete, raccoglie i dati e li inoltra ad un altro sistema remoto per ulteriori elaborazioni. Nell'utilizzo delle WSN si ha un'esigenza di bassa complessità del dispositivo e di un basso consumo di energia per ottenere una lunga durata della rete, ma nella progettazione è necessario trovare un giusto equilibrio tra capacità di comunicazione ed elaborazione dei dati. La maggior parte della ricerca sulle WSN si è concentrata sugli sviluppi nell'architettura e nella progettazione di algoritmi e protocolli efficienti dal punto di vista energetico, computazionale e nella localizzazione.

La struttura di una WSN è sempre la stessa ma le architetture di elaborazione e di comunicazione dei dati dipendono dal tipo di applicazione. La rete sviluppata nel progetto di questa tesi ha come obiettivo la misurazione dei parametri ambientali ed è composta da:

- un insieme di nodi wireless dotati di sensori
- una rete di interconnessione
- un punto di raccolta dati
- un insieme di risorse computazionali nel punto di arrivo dei dati, per effettuare analisi sui dati raccolti

Una struttura tipica di una WSN è mostrata in Figura 1.1.

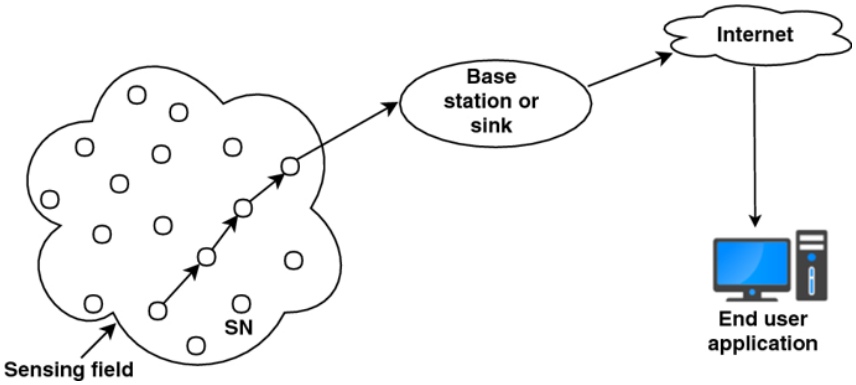


Figura 1.1: Struttura tipica di una WSN

Le WSN fanno parte di un'area di studio molto attiva dal punto di vista dell'innovazione tecnologica. Grazie alle loro potenzialità riescono ad essere impiegate in molti campi d'utilizzo e hanno il potenziale per cambiare la nostra vita in molti modi diversi. Per l'analisi delle varie applicazioni, una prima suddivisione può essere fatta in base alla finalità della rete [4, 5]. Di seguito si riportano quattro macro-settori in cui è possibile dividere il mondo delle WSN.

- **Monitoraggio**. Questo tipo di rete viene utilizzata per tracciare in maniera continuativa una certa grandezza. La sorgente da monitorare può essere un singolo sensore o una sottorete da cui proviene un aggregato di dati. Una rete di questo genere richiede un campionamento solitamente isocrono e fisso con un consumo energetico considerevole.
- **Riconoscimento di eventi**. In questo scenario, la rete deve accorgersi di situazioni di allarme, ossia quando una determinata grandezza esce dai livelli stabiliti.
- **Classificazione di oggetti**. L'obiettivo di queste rete è il riconoscimento di alcune grandezze tra un insieme di prototipi noti.
- **Tracciamento di oggetti**. In questo caso la rete funge da sistema di sorveglianza, riconoscendo e stimando la posizione di alcuni oggetti in una determinata area geografica.

Oltre a questa prima suddivisione si possono descrivere i vari scenari in cui le WSN sono presenti. Inizialmente erano adoperate solo in contesti particolari come la rilevazione di radiazioni, tracciamento di obiettivi e sorveglianza militare, rilevazione di dati biomedici, monitoraggio di un'area e rilevazione di attività sismiche. Più recentemente, l'attenzione è stata rivolta verso reti di sensori biologici e chimici per applicazioni di sicurezza nazionale, nonché verso lo sviluppo di soluzioni commerciali alla portata di molti. Un elenco di possibili applicazioni è il seguente:

- **Settore militare**:
  - monitoraggio di forze nemiche
  - monitoraggio di forze alleate ed equipaggiamenti
  - sorveglianza di campi di battaglia
  - stima dei danni
  - rilevazione di attacchi nucleari e biochimici
- **Settore ambientale**:
  - rilevamento di incendi boschivi
  - rilevamento di inondazioni
  - monitoraggio di frane
  - monitoraggio di microclimi
  - agricoltura di precisione

  Sono fondamentali in questo ambito, per controllare aree geografiche molto estese, soprattutto a ridosso dei centri abitati, per valutare l'impatto ambientale di alcune scelte umane. Si potrebbero sfruttare anche per lo studio di zone pericolose come vulcani o zone a rischio sismico. Anche in ambito agricolo sono utili, permettendo di controllare un certo tipo di coltura, intervenire in tempo reale con trattamenti per debellare insetti, oppure per l'automazione dell'irrigazione, consentendo un uso più efficiente dell'acqua e ridurre gli sprechi.
- **Settore sanitario**:
  - supporto a interfacce per disabili
  - monitoraggio di medici e di pazienti all'interno dell'ospedale
  - diagnostica e somministrazione di farmaci
  - telecontrollo di dati fisiologici umani

Si possono realizzare reti di sensori molto piccoli e distribuiti all'interno dei tessuti per misurare temperatura, pressione sanguigna e tanti altri parametri. Un esempio sono i dispositivi per attività ginniche, come orologi da polso o bracciali appositi che rilevino il battito cardiaco e le calorie bruciate durante l'attività.

- **Settore domestico**:
  - Home Automation
  - lettura contatori (smart metering)

  Tutti gli elettrodomestici più moderni hanno un'elettronica molto evoluta e il futuro consiste nell'interazione tra i vari elettrodomestici per collaborare autonomamente e gestire in maniera efficiente le faccende domestiche. Altre applicazioni sono la gestione del riscaldamento, della ventilazione e dell'illuminazione.

- **Settore commerciale**:
  - controllo ambientale in costruzioni industriali, edifici, ponti, cavalcavia, tunnel, ecc.
  - controllo di inventari
  - ambiente automotive
  - antifurto intelligenti
  - localizzazione di persone o oggetti in musei o scuole

- **Settore del trasporto**:
  - tracking e riconoscimento dei veicoli
  - raccolta informazioni sul traffico

Ovviamente i dati raccolti devono essere trasmessi in un tempo relativamente rapido, in base al tipo di applicazione della rete, in modo che si possano intraprendere tempestivamente le azioni dovute. Le reti di sensori possono essere incorporate in unità mobili come i robot, pertanto, sono piccoli, di basso costo e robusti; inoltre hanno diverse configurazioni: dai nodi connessi a una LAN e collegati permanentemente all'energia elettrica, a nodi che comunicano con tecnologia wireless alimentati da piccole batterie. Una rete di sensori deve consentire particolari funzionalità sui nodi, come l'elaborazione del segnale digitale, la compressione dello stesso, la gestione degli errori, la codifica delle informazioni. È necessario, perciò, che i componenti hardware siano costruiti ad hoc.

1.2 Caratteristiche delle WSN

Le WSN sono caratterizzate da alcuni aspetti chiave come, per esempio, i vincoli di potenza elaborativa, la durata limitata della batteria, un basso duty cycle e connessioni molti-a-uno, che creano delle sfide progettuali nell'ambito di trasporto dei dati, gestione della rete, disponibilità, confidenzialità, integrità e innetwork processing. Per di più, lo stack del protocollo realizzato deve essere quanto più leggero possibile, per poter soddisfare i vincoli imposti dai limiti hardware dei nodi. Ogni tipologia di WSN, come accennato in precedenza, si differenzia in base al tipo di applicazione. Caratteristica fondamentale di una WSN è la *Quality of Service (QoS)* [6] che rappresenta la descrizione o la misurazione delle prestazioni complessive di un servizio offerto dalla rete. I parametri *QoS* adottati tradizionalmente sono delay, bit rate, jitter, probabilità di errore nella trasmissione dati e banda a disposizione. Alcuni di questi non servono se si stanno utilizzando software che tollerano i ritardi e se i nodi scambiano pochi pacchetti alla volta. Inoltre, in alcuni casi, basta ricevere qualche dato ogni tanto, in altri invece, tutti i dati devono essere ricevuti oppure devono essere ricevuti entro un certo tempo. Ciò che conta è quindi la quantità e la qualità dei dati ricevuti, da un nodo di raccolta di una certa area. Ad esempio, delle metriche valide possono essere l'affidabilità nel rilevamento di certi eventi o l'approssimazione di determinate misure.

Non è possibile creare una WSN in grado di gestire tutte le applicazioni elencate in Paragrafo 1.1. Ognuna di esse avrà delle sfide progettuali particolari [5] e molte di queste saranno condivise. Tali sfide comuni si possono riassumere nelle seguenti caratteristiche:

- **Tolleranza ai guasti**. I nodi dei sensori sono vulnerabili e spesso usati in ambienti pericolosi. Per esempio, potrebbero esaurire la batteria, avere problemi hardware, subire danni fisici o perdere in qualunque altro modo il collegamento wireless con gli altri (presenza di un ostacolo di disturbo non previsto tra i nodi). I protocolli implementati in una rete di sensori dovrebbero essere in grado di rilevare questi guasti il più presto possibile ed essere sufficientemente robusti da gestire un numero relativamente grande di guasti mantenendo la funzionalità complessiva della rete. Ciò è particolarmente rilevante per la progettazione del protocollo di routing, che deve garantire la disponibilità di percorsi alternativi per il reinstradamento dei pacchetti.

- **Lifetime**. Nella maggior parte degli scenari, i nodi sono alimentati da batterie che non è possibile o non è conveniente sostituire. In ogni caso, l'obiettivo di una WSN è quello di rimanere attiva più a lungo possibile o, almeno, per la durata della sua missione. Quindi, il risparmio energetico assume un ruolo fondamentale in una WSN, per cercare di mantenere

la rete attiva per un tempo indeterminato. Tuttavia, l'implementazione di meccanismi di risparmio energetico richiede dei compromessi con la QoS. La soluzione è ovviamente quella di trovare un giusto bilanciamento tra le due caratteristiche. La definizione di lifetime non è univoca, nel senso che dipende dall'applicazione che si vuole misurare: a volte infatti si indica con lifetime il tempo entro il quale il primo nodo della rete non funziona più; altre volte invece corrisponde al momento in cui la metà dei nodi vengono persi; oppure indica la prima volta in cui una regione sotto controllo non è più monitorata da alcun nodo.

- **Scalabilità**. Le reti di sensori variano in scala da diversi nodi a potenzialmente diverse centinaia di migliaia. I protocolli implementati nelle reti di sensori devono essere scalabili a questi livelli ed essere in grado di mantenere prestazioni adeguate.

- **Densità dei nodi non uniforme**. In una rete di sensori possono esserci zone con molti e altre con pochissimi nodi sparsi. La densità dei nodi può variare nello spazio e nel tempo (ad esempio perché i nodi finiscono la batteria) e la rete deve essere in grado di adattarsi a queste variazioni.

- **Programmabilità**. I nodi devono essere in grado di poter cambiare i propri compiti in qualunque momento, ovvero devono poter essere riprogrammabili. A tal proposito, sono molti gli studi relativi a reti che riescono a riprogrammarsi autonomamente senza dover programmare manualmente ogni singolo nodo.

- **Auto-mantenimento**. Dato che sia la WSN sia l'ambiente in cui si trova sono in continuo mutamento, la rete deve essere in grado di adattarsi, monitorando il proprio stato di salute, aggiornando i propri parametri, decidendo tra nuovi compromessi (ad esempio diminuendo la QoS quando l'energia sta per terminare).

- **Vincoli hardware**. Ogni nodo del sensore deve avere un'unità di rilevamento, un'unità di elaborazione, un'unità di trasmissione e un alimentatore. Facoltativamente, i nodi possono avere diversi sensori integrati o dispositivi aggiuntivi come un sistema di localizzazione per consentire il routing in base alla posizione. Tuttavia, ogni funzionalità aggiuntiva comporta un costo aggiuntivo, aumentando il consumo di energia e le dimensioni fisiche del nodo. Pertanto, le funzionalità aggiuntive devono essere sempre bilanciate rispetto ai requisiti di costo e bassa potenza.

- **Mezzi di trasmissione**. La comunicazione tra i nodi viene normalmente implementata usando la comunicazione radio sulle popolari bande ISM. Tuttavia, alcune reti di sensori

utilizzano la comunicazione ottica o infrarossa, con quest'ultima che ha il vantaggio di essere robusta e praticamente priva di interferenze.

- **Consumo di energia**. Come abbiamo già visto, molte delle sfide delle reti di sensori ruotano attorno alle risorse di energia limitate. La dimensione dei nodi limita la dimensione della batteria. La progettazione di software e hardware deve considerare attentamente le problematiche dell'uso efficiente dell'energia. Ad esempio, la compressione dei dati potrebbe ridurre la quantità di energia utilizzata per la trasmissione radio, ma utilizza energia aggiuntiva per il calcolo e/o il filtraggio. La politica energetica dipende anche dall'applicazione; in alcune applicazioni, potrebbe essere accettabile disattivare un sottoinsieme di nodi per risparmiare energia, mentre altre applicazioni richiedono che tutti i nodi funzionino contemporaneamente.

Per risolvere tutte queste sfide progettuali, nel corso degli anni, sono stati sviluppati diversi meccanismi per la comunicazione, architetture di sistema e protocolli di varie tipologie. Con tali sviluppi si riescono ad ottenere le seguenti importanti funzionalità:

- **Connessioni wireless multi-hop**. La comunicazione diretta tra due nodi non è sempre possibile, poiché potrebbero esserci ostacoli oppure perché i nodi sono molto distanti tra loro e l'utilizzo di una potenza trasmissiva elevata comporterebbe un rapido esaurimento della batteria. Quindi la soluzione è quella di adoperare dei nodi che fungano da relay verso altri nodi (Figura 1.2).

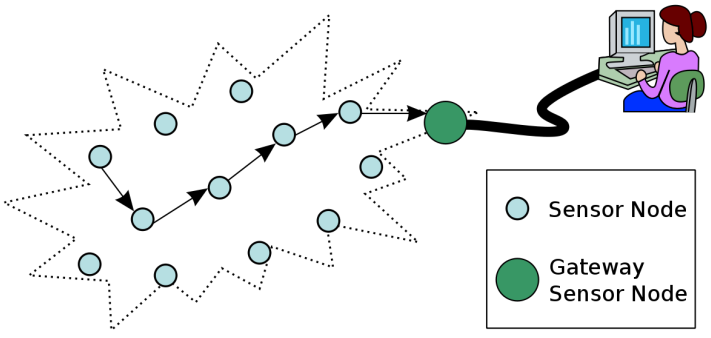


Figura 1.2: Rappresentazione del multi-hop

- **Operazioni energeticamente efficienti**. È importante che tutte le operazioni compiute tengano in considerazione il risparmio energetico e bisogna, possibilmente, evitare la formazione di regioni o gruppi di nodi che esauriscano la propria energia molto più rapidamente degli altri.

- **Autoconfigurazione**. La rete deve essere in grado di configurare automaticamente tutti i suoi parametri vitali. Per esempio, deve gestire autonomamente l'ingresso di un nuovo nodo oppure l'aggiornamento delle tabelle di routing dopo la perdita di alcuni nodi.

- **Collaborazione e in-network processing**. In alcune applicazioni un singolo nodo non è in grado di capire se si è verificato un evento. Per questo motivo è necessario che i nodi collaborino tra di loro e compiano in-network processing, ovvero eseguano alcuni calcoli sui dati, tipicamente data-aggregation (ad esempio il calcolo della temperatura media di una zona) in modo da ridurre la quantità di dati trasmessa attraverso la rete, oppure sfruttando la correlazione tra misurazioni di più sensori.

- **Data-centric**. In una rete di comunicazione tradizionale, lo scambio di dati avviene tra entità aventi ognuna un indirizzo di rete specifico, quindi si tratta di un'architettura address-centric. In una WSN, non importa tanto chi fornisce il dato, ma da quale regione proviene. Infatti, un nodo può essere ridondato da più nodi, e quindi si perde l'individualità dei vari componenti. Ciò che interessa è richiedere una certa informazione ad una certa area monitorata, e non richiedere una certa informazione ad un certo nodo. È un concetto simile alla query di un database 'Visualizza tutte le aree in cui la temperatura è maggiore di X' oppure 'Richiedi i dati di umidità della regione X'.

- **Località**. Per risparmiare risorse hardware, il nodo deve interessarsi e memorizzare informazioni di routing solo verso i nodi vicini a lui. Così facendo, nel momento in cui la rete dovesse crescere esponenzialmente, le risorse hardware occupate rimarrebbero inalterate. Chiaramente, conciliare località e protocolli di routing efficienti è una delle sfide da affrontare.

- **Bilanciamento dei trade-off**. Sia durante la fase di progettazione della WSN sia durante il suo runtime, bisogna ponderare diversi trade-off, anche contraddittori tra loro. Alcuni di questi sono già stati accennati: lifetime e qualità del servizio, lifetime della rete e lifetime del singolo nodo, densità della rete ed efficienza del routing, solo per citarne alcuni.

1.3 Struttura di un nodo wireless

Vista la necessità dell'implementazione di un'infrastruttura di rete in cui i nodi svolgono diversi compiti su richiesta del gateway di controllo, questi devono essere sempre più evoluti, cioè non possono essere dei semplici trasduttori di grandezze fisiche, ma devono essere sistemi più complessi che integrano, oltre alla capacità di misura, anche capacità di memorizzazione, di

calcolo ed ovviamente interfacce di comunicazione. Queste osservazioni portano alla definizione degli *smart sensor*, dispositivi integrati dotati di microcontrollori in grado di effettuare attività di comunicazione ed elaborazione dell'informazione.

Come mostrato in Figura 1.3, un nodo wireless è costituito principalmente da quattro componenti di base:

- Unità di **rilevamento**. Sono generalmente composti da sensori e convertitori da analogico a digitale (ADC). I segnali analogici prodotti dai sensori vengono convertiti in segnali digitali dall'ADC e quindi immessi nell'unità di elaborazione.
- Unità di **elaborazione**. Corrisponde a una piccola unità di archiviazione (es. microcontrollore) e può gestire le procedure che consentono al nodo sensore di collaborare con gli altri nodi per eseguire le attività di rilevamento assegnate.
- Unità di **ricetrasmettitore**. Serve per collegare il nodo alla WSN.
- Unità di **potenza**. Per dare energia al nodo si ricorre a batterie, che possono essere di qualsiasi tipo, a seconda dell'hardware e dell'applicazione da realizzare. Le reti che devono avere un lifetime molto lungo possono essere supportate da un'unità di assorbimento di energia come celle solari per ricaricare le batterie.

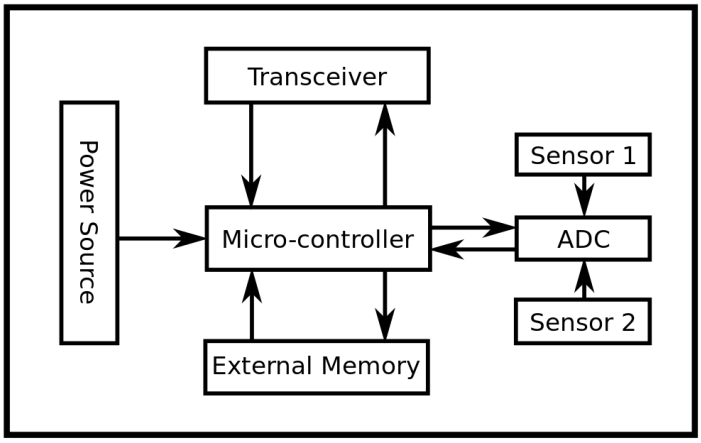


Figura 1.3: Componenti di un nodo sensore

Uno schema funzionale a blocchi di un nodo wireless è mostrato in Figura 1.4 [3]. L'approccio di progettazione modulare offre una piattaforma flessibile e versatile per soddisfare le esigenze di un'ampia varietà di applicazioni. Ad esempio, a seconda dei sensori da implementare, il blocco di condizionamento del segnale può essere riprogrammato o sostituito. Ciò consente di utilizzare

un'ampia varietà di sensori diversi con il nodo di rilevamento wireless. Allo stesso modo, il collegamento radio può essere sostituito come richiesto per il requisito di portata wireless di una determinata applicazione e la necessità di comunicazioni bidirezionali.

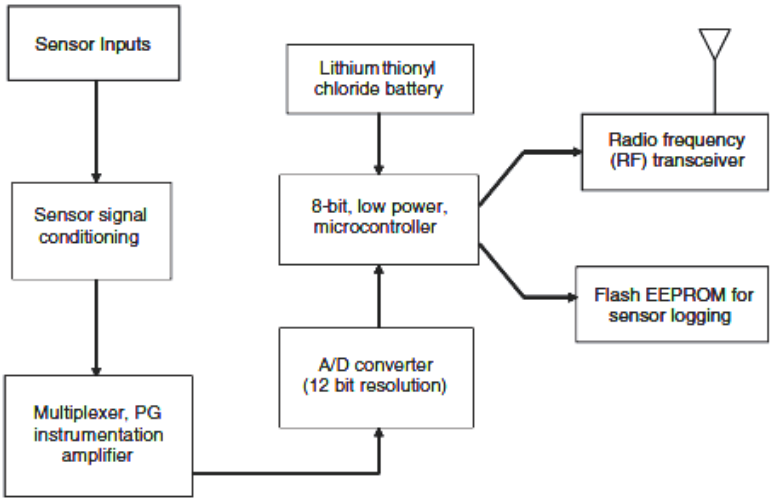


Figura 1.4: Schema a blocchi funzionale di un nodo sensore

Usando la memoria flash, i nodi remoti acquisiscono dati su comando da una stazione base o da un evento rilevato da uno o più ingressi al nodo. Inoltre, il firmware incorporato può essere aggiornato tramite la rete wireless sul campo. La CPU del nodo ha una serie di funzioni tra cui:

- gestione della raccolta dei dati dai sensori
- svolgere funzioni di gestione dell'alimentazione
- interfacciamento dei dati del sensore al livello radio fisico
- gestione del protocollo di rete radio

I componenti più adatti a questo scopo sono i microcontrollori, ovvero processori per sistemi embedded, poiché richiedono poca energia, hanno spesso delle memorie integrate e non hanno controller di memoria, sono facilmente programmabili, si integrano bene con altri componenti come sensori e hanno delle modalità di risparmio energetico.

Un aspetto chiave di qualsiasi nodo di rilevamento wireless è quello di ridurre al minimo la potenza consumata dal sistema. Di solito, il sottosistema radio richiede la massima quantità di energia. Pertanto, i dati vengono inviati sulla rete radio solo quando è necessario. Un algoritmo deve essere caricato nel nodo per determinare quando inviare i dati in base all'evento rilevato. Inoltre, è importante ridurre al minimo la potenza consumata dal sensore stesso, quindi l'hardware

dovrebbe essere progettato in modo da consentire al microprocessore di controllare con giudizio l'alimentazione della radio, del sensore e del condizionatore del segnale del sensore.

1.4 Architettura di una WSN

Per descrivere l'architettura di rete di una Wireless Sensor Network, si può ricorrere al modello OSI (*Open Systems Interconnection*), in Figura 1.5 [4]. Venga tenuto da subito in considerazione che la separazione dei layer in una WSN non è così netta; al contrario, i layer si sovrappongono tra loro, in particolare il livello 2 (*livello Data Link*) e il livello 3 (*livello Network*), come conseguenza dei trade-off elencati nei paragrafi precedenti. Questo stack di protocollo unisce potenza e consapevolezza del routing, integra i dati con i protocolli di rete, comunica la potenza in modo efficiente attraverso il supporto wireless e promuove gli sforzi cooperativi dei nodi dei sensori.

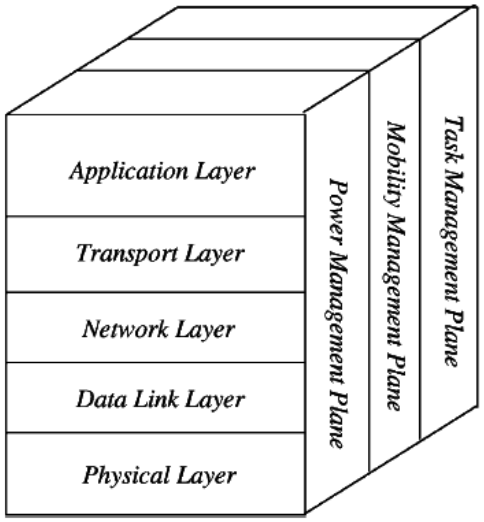


Figura 1.5: Stack di protocollo della WSN (modello OSI)

Lo stack di protocollo è composto da:

1. **Physical Layer**. Si occupa di controllare la rete, gli hardware che la compongono e i dispositivi che permettono la connessione. Trasmette un flusso di dati non strutturati attraverso un collegamento fisico, occupandosi della forma e dei livelli di tensione del segnale.

2. **Data Link Layer**. Permette il trasferimento affidabile di dati attraverso il livello fisico con la necessaria sincronizzazione ed effettua controllo degli errori e delle perdite di segnale. Questo è necessario per far apparire, al livello superiore, il mezzo fisico come una linea di trasmissione senza errori. Il Data Link è composto da due sottolivelli: l'LLC (*Logical Link*

Control), che fa da tramite tra i livelli superiori e quello fisico; il MAC (*Medium Access Control*), che assembla i dati da inviare in frame o al contrario, disassembla i frame ricevuti in dati per estrarre le informazioni, gestendo anche la correzione degli errori.

3. **Network Layer**. Rende i livelli superiori indipendenti dai meccanismi e dalle tecnologie di trasmissione usate per la connessione. Si occupa quindi del routing dei dati forniti dal livello di trasporto, protocolli di routing wireless multi-hop specifici tra i nodi del sensore e il gateway. Il routing è fondamentale per individuare i cammini ottimali per poter trasferire i dati raccolti dai nodi al gateway.

4. **Transport Layer**. Permette di stabilire, mantenere e terminare un trasferimento di dati trasparente e affidabile tra due host; inoltre evita che troppi pacchetti dati arrivino allo stesso router contemporaneamente con effetto di perdita dei pacchetti stessi.

5. **Application Layer**. Fornisce un insieme di protocolli che operano a stretto contatto con le applicazioni, permettendo di interfacciare utente e macchina. Si possono creare e utilizzare diversi tipi di software in base alle attività di rilevamento.

Inoltre, i piani di alimentazione, mobilità e gestione delle attività monitorano l'alimentazione, il movimento e la distribuzione delle attività tra i nodi del sensore. Questi piani aiutano i nodi del sensore a coordinare l'attività di rilevamento e ridurre il consumo energetico complessivo.

1.5 Protocollo di comunicazione MQTT

Nel mondo dell'informatica utilizziamo HTTP come protocollo di trasferimento dei dati, ma questo è nato per tutto ciò che riguarda la tecnologia web (payload di grandi dimensioni) e non per trasferire dati e informazioni di piccole dimensioni, quindi non è adatto per quei dispositivi che non hanno una larga banda di comunicazione, come le piattaforme embedded, che devono mantenere dei bassi consumi e non hanno elevate potenze di calcolo.

Il protocollo che nel corso del tempo si è rivelato particolarmente adatto a tutto ciò che riguarda l'IoT è MQTT (*Message Queuing Telemetry Transport*) [1, 7], di *IBM*, studiato proprio per le situazioni in cui la banda è limitata e le reti sono poco affidabili. Infatti, l'*Organization for the Advancement of Structured Information Standards* (OASIS) ha dichiarato che il protocollo MQTT è lo standard di riferimento per la comunicazione per l'IoT.

Figura 1.6: Diversi tipi di Client MQTT in una WSN

1.5.1 Funzionamento

MQTT corrisponde ad un protocollo di trasmissione dati TCP/IP basato su un modello di pubblicazione e sottoscrizione che opera attraverso un *message broker*. Invece di inviare messaggi a un determinato set di destinatari, i mittenti pubblicano i messaggi su un certo argomento (detto *topic*) sul message broker. Ogni destinatario si iscrive agli argomenti che lo interessano e, ogni volta che un nuovo messaggio viene pubblicato su quel determinato argomento, i broker MQTT gestiscono la ricezione dei messaggi e il successivo invio ai sottoscrittori; allo stesso tempo, il broker provvede alla gestione delle liste di argomenti a cui i destinatari sono interessati. Come si può vedere in Figura 1.6, qualsiasi device o applicazione può essere un *client MQTT* che si appoggia a un'apposita libreria MQTT a sua volta connessa in rete a un broker MQTT. In questo modo è molto semplice configurare una messaggistica uno-a-molti.

I nodi della WSN vengono suddivisi in *Publisher*, quelli che inviano i messaggi e *Subscriber*, quelli che ricevono i messaggi. I benefici dell'utilizzo del protocollo MQTT sono:

- **Space Decoupling**. Publisher e Subscriber conoscono soltanto l'indirizzo IP del broker MQTT e non sono a conoscenza dell'esistenza di altri nodi.
- **Time Decoupling**. Publisher e Subscriber non vengono eseguiti nello stesso istante, quindi il broker può memorizzare i messaggi da inoltrare ai Subscriber offline in un secondo momento.
- **Synchronization Decoupling**. Un client che sta eseguendo, per esempio, un'operazione di pubblicazione non viene interrotto per ricevere un messaggio pubblicato da un altro nodo se ha effettuato la sottoscrizione a quel tipo di messaggio. Il messaggio verrà inserito in

coda finché il client in questione non ha terminato l'operazione esistente. Questo permette di ridurre la ripetizione delle stesso operazioni evitandone le interruzioni.

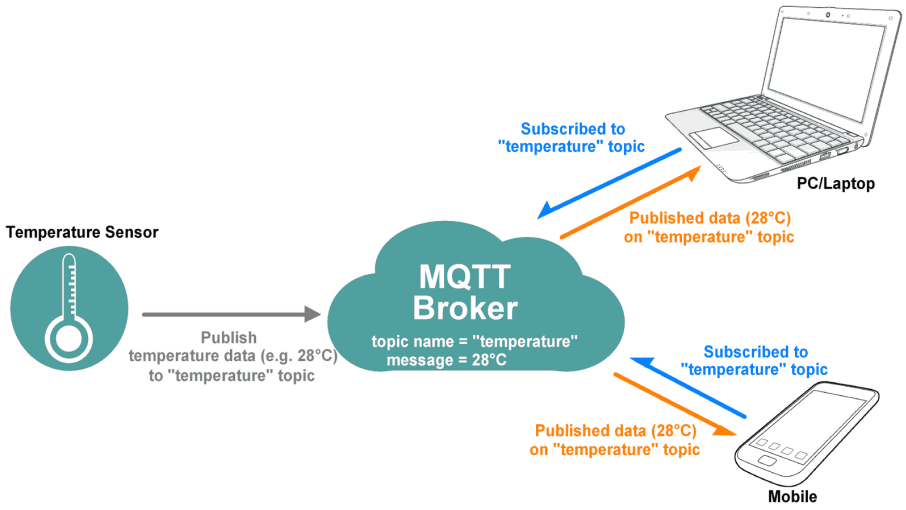


Figura 1.7: Nodi wireless comunicano attraverso un broker MQTT

Un esempio di comunicazione MQTT di più nodi wireless attraverso un broker è mostrato in Figura 1.7. Il protocollo MQTT lavora scambiando una serie di MQTT control packet:

- *Connect.* Il Client chiede la connessione al server.
- *Publish.* Il Client invia un messaggio al server e viceversa.
- *Subscribe.* Il Client si sottoscrive a un determinato topic, chiedendolo al server.
- *Disconnect.* Il Client si disconnette dal server.

1.5.2 Livelli di servizio

Pur essendo usato per esigenze di riduzione al minimo in termini di ampiezza di banda, MQTT è in grado di mantenere una certa affidabilità e un certo grado di certezza di invio e ricezione dei dati. Il protocollo MQTT possiede tre livelli di servizio [8]:

1. *At most once* (al massimo una volta). La distribuzione dei messaggi è eseguita in base a quanto meglio garantito dalla rete. Visto che in questo caso la perdita di messaggi o la loro duplicazione possono comunque verificarsi, l'uso ottimale è identificato con i dati raccolti da sensori ambientali dove non importa se una singola lettura viene smarrita in quanto quella successiva godrà presto di un'altra pubblicazione.
2. *At least once* (almeno una volta). Viene invece assicurato l'arrivo dei messaggi, ma possono comunque verificarsi dei duplicati.

3. *Exactly once* (esattamente una volta). Si assicura che i messaggi arrivino, appunto, esattamente una volta.

MQTT supporta infine anche un certo grado di sicurezza considerato che con un suo pacchetto è possibile passare direttamente anche nomi utente e password. Se si vogliono cifrare i dati trasmessi, tra i metodi suggeriti ci sono invece l'affiancamento di SSL (*Secure Sockets Layer*), il che in ogni caso aggiunge pesantezza al tutto, oppure l'uso di applicazioni di criptatura a monte e a valle della trasmissione.

Nel progetto di questa tesi, i nodi che partecipano alla WSN fungono da Publisher secondo il protocollo MQTT. Invece, il compito del punto di raccolta dei dati è svolto da un computer Raspberry, che svolge il ruolo di broker MQTT.

Capitolo 2

AGGIORNAMENTO FIRMWARE OVER THE AIR

L'aggiornamento Over The Air, o semplicemente OTA, è un metodo di aggiornamento firmware tramite Internet in modalità wireless e quindi senza fili. Gli aggiornamenti sono importanti perché possono essere utilizzati per correggere e risolvere le vulnerabilità della sicurezza, aggiungere nuove funzionalità, incorporare nuovi protocolli di comunicazione e soddisfare i requisiti di conformità. Ogni giorno, siamo abituati ad eseguire aggiornamenti OTA, come i normali aggiornamenti del sistema operativo o delle applicazioni dei nostri dispositivi portatili (smartphone, tablet, ecc.). Questo metodo ha degli ottimi vantaggi, sia dal punto di vista del produttore che dell'utente: il primo può inviare e gestire in modo centralizzato un aggiornamento a tutti i suoi utenti, senza ricorrere a supporti fisici e in tempi rapidi; l'utente riceve la notifica e può rifiutare o meno l'installazione, oppure è costretto forzatamente dal produttore. Il meccanismo OTA richiede che il software e l'hardware esistenti del dispositivo di destinazione supportino tale funzione, vale a dire la ricezione e l'installazione del nuovo firmware tramite la rete wireless dal produttore. Per esempio, nei nostri smartphone, può capitare che gli aggiornamenti siano disponibili solo per i dispositivi costruiti con la più recente tecnologia.

In questo capitolo viene descritto nel dettaglio OTA per sistemi a microcontrollori, per comprendere le sue caratteristiche e il funzionamento. Viene inoltre presentato l'algoritmo OTA implementato nel dispositivo DONET. Per inoltrare gli aggiornamenti si fa uso di un repository GitHub, per cui è mostrata una breve guida per l'autenticazione API.

2.1 OTA in Sistemi Embedded

Per programmare un qualsiasi microcontrollore, si è abituati a collegarsi alla board tramite un connettore USB-seriale, necessario per poter apportare modifiche al software o per il collegamento

al monitor seriale. In applicazioni IoT, i sistemi embedded vengono distribuiti in quantità elevate e con una durata della batteria limitata, inoltre, sono distribuiti in luoghi difficili o poco pratici per l'accesso di un operatore umano. Alcuni esempi possono essere i dispositivi che monitorano lo stato di salute di una persona o di una macchina. Queste sfide, associate al rapido ciclo di vita del software, costringono molti sistemi a richiedere il supporto per gli aggiornamenti OTA, rappresentando una vera e propria necessità a cui non si può rinunciare. L'aggiornamento OTA [9] offre la possibilità di aggiornare l'applicazione in esecuzione sul dispositivo senza accedere fisicamente sullo stesso. Alcuni articoli trattano diverse implementazioni di OTA come in [10, 11, 12]. Utilizzando un link radio (tipicamente WiFi, ma possono essere utilizzati altri protocolli di comunicazione, come la rete mobile 3G/4G), il dispositivo viene connesso ad un server sul quale è presente la versione da utilizzarsi per l'aggiornamento. Questa viene trasferita sul microcontrollore o sul microprocessore del sistema incorporato, sostituendo il software presente e verrà messa in esecuzione al riavvio successivo. In [13, 14] sono presentate alcune implementazioni di algoritmi OTA per ESP8266.

Un esempio di un sistema incorporato che potrebbe richiedere aggiornamenti OTA è mostrato in Figura 2.1 [15]. Un microcontrollore è collegato a una radio e un sensore che può per esempio misurare la temperatura e invia i dati periodicamente utilizzando la radio. Questo sistema è quello che nel Capitolo 1 è chiamato nodo wireless o *client* e corrisponde alla destinazione dell'aggiornamento OTA. L'altra parte del sistema è denominata *cloud* o *server* ed è il fornitore del nuovo software. Server e client comunicano tramite ricetrasmettitori radio.

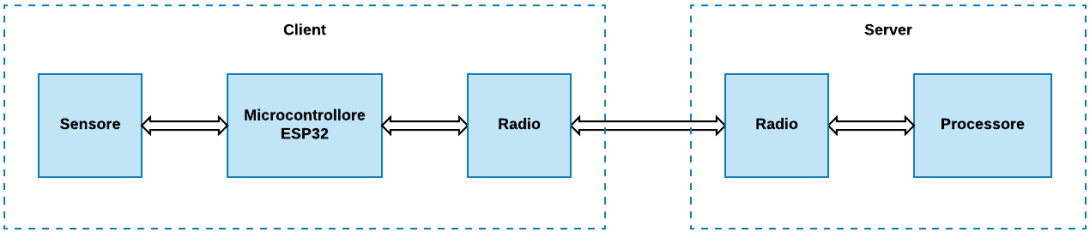


Figura 2.1: Esempio di architettura client-server in un sistema embedded

Il software viene trasferito come una sequenza di byte, dopo che è stato convertito in un formato binario dal formato di origine. Il processo di conversione compila i file del codice sorgente (*c*, *cpp*, *py*), li collega insieme in un file eseguibile (*exe*, *elf*), convertito in un formato di file binario portatile (*bin*, *hex*). A un livello elevato, questi formati di file contengono una sequenza di byte che appartengono a un indirizzo specifico di memoria nel microcontrollore. In genere,

concettualizziamo le informazioni inviate su un collegamento wireless come dati, come ad esempio un comando per modificare lo stato del sistema o i dati del sensore raccolti dal sistema. Nel caso dell'aggiornamento OTA, i dati corrispondono al nuovo firmware in formato binario. In molti casi, il file binario è troppo grande per inviare un singolo trasferimento dal server al client, il che significa che il file binario dovrà essere inserito in pacchetti separati, in un processo chiamato *packetizing*. Per visualizzare meglio questo processo, la Figura 2.2 mostra come diverse versioni del software producono diversi file binari e quindi diversi pacchetti da inviare durante l'aggiornamento OTA. In questo semplice esempio, ogni pacchetto contiene 8 byte di dati.

Figura 2.2: Processo di conversione binaria e di pacchettizzazione di un'applicazione software.

Sulla base di questa descrizione di alto livello del processo di aggiornamento OTA, sorgono tre sfide principali che la soluzione di aggiornamento OTA deve affrontare:

- **Memoria**. La soluzione software deve organizzare la nuova applicazione in memoria volatile o non volatile del dispositivo client in modo che possa essere eseguita al termine del processo di aggiornamento. La soluzione deve garantire che una versione precedente del firmware venga mantenuta come applicazione di fallback nel caso in cui quella nuova presentasse problemi. Inoltre, dobbiamo mantenere lo stato del dispositivo client tra reset e cicli di accensione, come la versione del firmware che stiamo attualmente eseguendo e dove si trova in memoria.
- **Comunicazione**. Il nuovo firmware deve essere inviato dal server al client in pacchetti discreti, ciascuno destinato a un indirizzo specifico nella memoria del client. Lo schema per il pacchetto, la struttura dei pacchetti e il protocollo utilizzato per trasferire i dati devono essere presi in considerazione nella progettazione del software.
- **Sicurezza**. Ci sono tre aspetti fondamentali per la sicurezza del download di una nuova versione del firmware:
  - *Autenticazione*. Bisogna assicurarsi che il server da cui proviene l'aggiornamento sia una parte attendibile.
  - *Riservatezza*. Si deve garantire che il nuovo firmware sia offuscato da eventuali osservatori, poiché potrebbe contenere informazioni riservate.

– *Integrità*. Bisogna assicurarsi che il nuovo firmware non sia danneggiato quando viene inviato.

2.2 API GitHub

L'algoritmo OTA implementato in DONET utilizza un repository privato su GitHub, nel quale vengono caricati gli aggiornamenti del firmware che il dispositivo potrà scaricare e installare. Per questo motivo è utile fare una breve introduzione su *API GitHub* [16] e sull'autenticazione per l'accesso.

API in informatica è l'acronimo di *Application Programming Interface* e si tratta di applicazioni che, mediante modalità standard, espongono le funzionalità di altre applicazioni, cioè è un insieme di definizioni, protocolli e metodi di comunicazione chiaramente definiti tra i vari componenti software. Il significato di API è quello di semplificare la possibilità di dialogo tra un'applicazione e un'altra. API GitHub è un'interfaccia fornita da GitHub per gli sviluppatori che vogliono creare applicazioni destinate a GitHub e per lavorare con il repository di codice. Utilizzando repository privati, per poter accedere è necessaria un'autenticazione. Esiste un'autenticazione di base con nome utente e password, ma esiste anche un altro modo più semplice e migliore, che utilizza un *token*, offrendo i seguenti vantaggi:

- **Accesso revocabile**. I token possono essere revocati in qualsiasi momento dalla schermata delle impostazioni di un utente in GitHub
- **Accesso limitato**. Quando si richiede l'accesso le applicazioni dichiarano il livello di autorizzazioni di cui hanno bisogno e i token vengono creati con l'ambito appropriato come concesso dall'utente

I token devono essere trattati come delle password, perciò non bisogna condividerli con altri utenti o conservarli in luoghi non sicuri. Per generare un token bisogna andare nel proprio account GitHub, cliccare sulla foto del profilo nell'angolo in alto a destra e selezionare *Settings* (Figura 2.3).

Nella barra laterale sinistra, selezionare *Developer settings* > *Personal access tokens* > *Generate new token* e fornire un nome descrittivo al token. A questo punto bisogna selezionare gli ambiti o le autorizzazioni che si desiderano concedere con questo token. Per lo scopo di questo progetto, basta selezionare le impostazioni mostrate in Figura 2.4. Una volta generato il token, bisogna copiarlo negli *Appunti* per utilizzarlo, perché una volta usciti dalla pagina non sarà più possibile visualizzarlo.

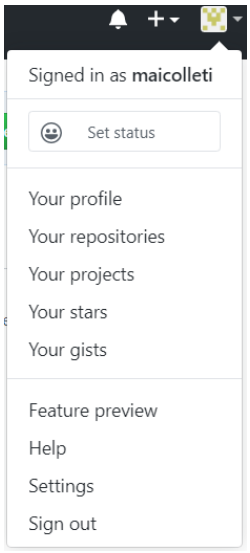


Figura 2.3: Impostazioni profilo personale GitHub

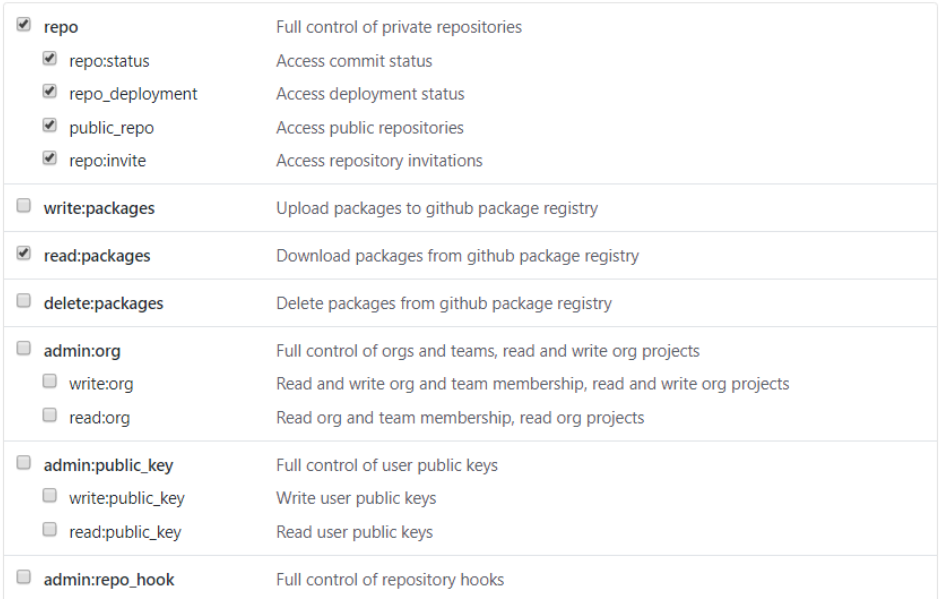


Figura 2.4: Autorizzazioni token API GitHub

2.3 OTA in DONET

Per eseguire l'aggiornamento OTA nel dispositivo DONET si utilizza, come già introdotto, un repository privato GitHub, riadattando l'idea mostrata in [17]. Prima di descrivere l'algoritmo implementato è importante comprendere come sono strutturati i firmware in DONET e nel repository. In DONET è presente una cartella *main* che contiene tutte le librerie in MicroPython (compresa quella per l'OTA) necessarie agli scopi desiderati, in più è presente un file *main.py* che viene eseguito dal chip dopo l'accensione e contiene solo un loop infinito che richiama le

funzioni dalle librerie. Nel repository invece, è presente la stessa cartella *main* contenente le librerie. Quando vengono caricati dei file nel repository è possibile associare un numero della release del codice (ad esempio *1.0*) e può essere aggiornato quando si fanno delle modifiche al codice. Il firmware dell'aggiornamento OTA scritto in linguaggio MicroPython è mostrato in Paragrafo 5.6, mentre di seguito è spiegata la logica di funzionamento con la quale è stato scritto. Quando si chiede a DONET di controllare se è presente un nuovo aggiornamento OTA vengono eseguite le seguenti operazioni:

1. accesso al repository in GitHub tramite il token assegnato
2. acquisizione del numero dell'ultima versione del codice disponibile in GitHub
3. confronto di questo numero di versione con quello dell'ultima versione presente nel chip:
    - se è uguale, allora l'aggiornamento non è presente e termina il controllo
    - se è maggiore, allora significa che è presente una nuova versione da scaricare
4. download dei file contenuti nella cartella *main* del repository all'interno di nuova cartella chiamata *next* all'interno del chip
5. eliminazione della cartella *main* che contiene il vecchio codice nel chip
6. cambio nome della cartella *next* in *main* e aggiornamento del numero della release del codice
7. riavvio del dispositivo ed esecuzione del nuovo codice scaricato.

Capitolo 3

ARCHITETTURA E FUNZIONAMENTO DEL SISTEMA

In questo capitolo viene descritta l'architettura della rete wireless: si osserva in generale l'infrastruttura informatica, per poi descrivere nel dettaglio la componentistica del nodo DONET e il broker MQTT. Infine, è spiegato il funzionamento del dispositivo e di come è stato ideato il suo algoritmo.

3.1 Infrastruttura informatica

Il sistema è stato creato per monitorare parametri come temperatura, umidità, luminosità e qualità dell'aria all'interno di una stanza di un edificio. Bisogna quindi raccogliere questi parametri attraverso sensori e DONET è il dispositivo realizzato per farlo. DONET è anche in grado di inviare i dati raccolti al broker MQTT che funge da gateway coordinatore, il quale è connesso ad un database per la memorizzazione e gestione dei dati.

Si è realizzato un sito web che riceve i dati trasmessi dai dispositivi DONET e capace di accedere al database in modo da caricare i dati presenti in esso. Per la realizzazione del sito è stato necessario creare un server web. Non avendo disponibilità hardware per la creazione del server si è deciso di appoggiarsi ad un servizio di hosting che ospita fisicamente su un server web tutti i file che costituiscono il sito, per essere raggiungibile attraverso Internet. Il database raccoglie i dati provenienti da DONET, i dati di registrazione dell'utente al sito e le informazioni riguardanti l'edificio in cui è situato il nodo. Il risultato finale è che l'utente o il sistema può consultare, ad esempio tramite un browser, le informazioni di un'area remota e agire di conseguenza. L'architettura del sistema è mostrata in Figura 3.1 e si suddivide in quattro blocchi:

1. **Dispositivi IoT**. Corrispondono ai nodi DONET.

2. **Server**. Contiene il broker MQTT e un'applicazione Python.

3. **Hosting**. Contiene il server web e il database per la memorizzazione dei dati.

4. **Lato Client**. Costituito dal sito web e dall'applicazione mobile che permettono all'utente di interfacciarsi con il resto della rete.

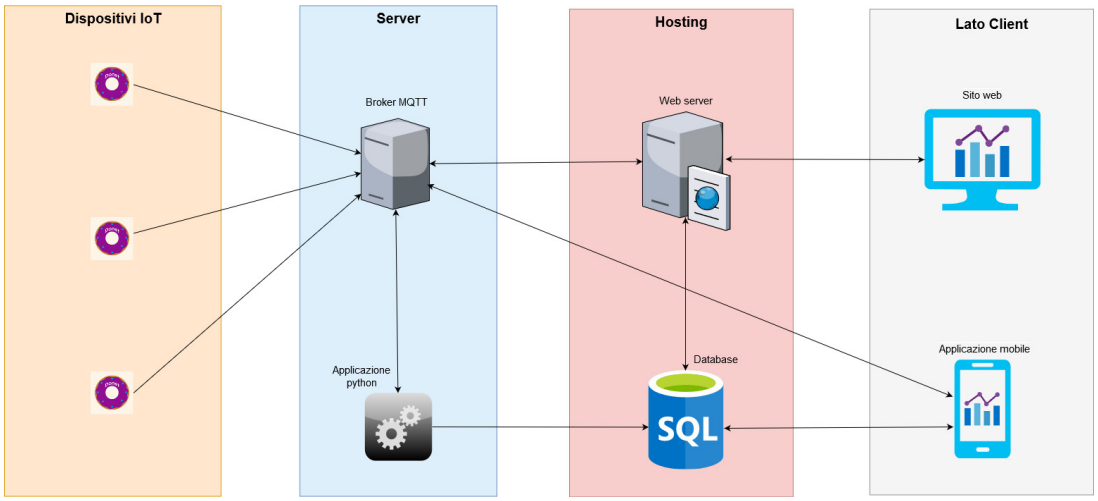


Figura 3.1: Architettura completa della WSN di DONET

L'obiettivo di questa tesi è di realizzare il nodo wireless DONET con MicroPython come linguaggio di programmazione, perciò nei paragrafi seguenti sono descritti in dettaglio soltanto i primi due blocchi dell'architettura. Per maggiori informazioni riguardanti i blocchi di *Hosting* e *Lato Client*, consultare [1].

3.2 Hardware DONET

I dispositivi IoT di questo progetto sono chiamati DONET, acronimo di *DON* (derivante da *Donut*) e *NET* (derivante da *Network*). Come già annunciato, DONET è un dispositivo dotato di un microcontrollore con antenna capace di connettersi a un Access Point locale, tre sensori per misurare temperatura, umidità relativa, luminosità, anidride carbonica e composti organici volatili di una stanza. DONET è mostrato in Figura 3.2 e oltre a rilevare i parametri ambientali, è in grado di inviarli al broker remoto, attraverso la connessione WiFi locale. È dotato anche di LED RGB a bordo, che si accendono a seconda degli eventi. Inoltre, è alimentato da una tensione di 5 V che viene immediatamente convertita a 3.3 V perché tutto il circuito elettronico funziona a tale voltaggio. Si è scelto di non alimentare il nodo wireless mediante una batteria in quanto il consumo di corrente elettrica è ridotto ma non è tale da garantire una durata superiore

ad un mese con una batteria standard. Di seguito sono descritti i componenti hardware di un dispositivo DONET.

Figura 3.2: Nodo wireless DONET

3.2.1 Chip ESP32

Il chip ESP32 è costruito da *Espressif Systems* [18] e progettato per l'IoT e progetti relativi ai sistemi integrati. ESP32 è un sistema a basso costo e basso consumo su una serie di chip di microcontrollori con funzionalità WiFi e Bluetooth integrate e una struttura altamente integrata alimentata da un microprocessore *Tensilica Xtensa LX6 dual-core*. Il chip fornisce la funzionalità di microcontrollore, permettendo il rilevamento dei dati dai sensori, la loro elaborazione e trasmissione al gateway di rete, grazie al supporto per il protocollo di rete TCP/IP. Insieme al rilascio di ESP32, *Espressif Systems* offre anche un modulo *ESP-WROOM-32* corrispondente, di dimensioni ridotte e molto semplice da usare grazie a componenti integrati come antenna, oscillatore e flash.

La Tabella 3.1 confronta le caratteristiche del chip ESP32 con quelle di ESP8266 ed evidenzia quali sono le funzionalità in più a disposizione [19]. ESP32 ha un core aggiuntivo a 32 bit e a bassa potenza con velocità di clock massima di 240 MHz. Oltre a supportare la tecnologia Bluetooth, ha un WiFi più veloce fino a 150 Mbps contro i 72.2 Mbps del suo predecessore. Il chip ha il grande vantaggio di possedere molte più porte GPIO e si può decidere quali pin siano UART, I^2C, SPI, basta solo impostarlo sul codice, grazie alla funzione di multiplexing che consente di assegnare più funzioni allo stesso pin. L'unico canale ADC di ESP8266 non può funzionare insieme al WiFi, perciò prima di usarlo bisogna spegnere la rete, ESP32 invece risolve questo problema e ha molti più canali disponibili, oltre a quelli per il DAC. Altro aspetto da non sottovalutare riguarda la crittografia. Il protocollo MQTT non ha funzionalità di sicurezza

Chip	**ESP32**	**ESP8266**
CPU	Tensilica Xtensa LX6 32 bit Dual-Core at 160/240 MHz	Tensilica LX106 32 bit at 80 MHz (up to 160 MHz)
SRAM	520 kB	36 kB available
FLASH	2 MB (max 64 MB)	4 MB (max 16 MB)
Voltage	2.2 V to 3.6 V	3.0 V to 3.6 V
Operating Current	80 mA average	80 mA average
WiFi	802.11 b/g/n	802.11 b/g/n
Bluetooth	Bluetooth 4.2 + BLE	-
GPIO	36	17
Software PWM	16 channels	8 channels
SPI/I2C/I2S/UART	4/2/2/2	2/1/2/2
ADC	18 channels (12 bit)	1 channel (10 bit)
DAC	2 channels (8 bit)	-
Size	25.5 x 18.0 x 2.8 mm	24.0 x 16.0 x 3.0 mm
Price	$6 - $12	$3 - $6

Tabella 3.1: Confronto ESP32 con ESP8266

integrate oltre all'autenticazione nome utente/password. ESP8266 non fornisce alcun mezzo per proteggere il codice o i dati che memorizza; chiunque abbia accesso fisico al dispositivo può leggere le informazioni più sensibili, come le credenziali WiFi. Quindi è comune crittografare e autenticare attraverso una rete con SSL. Tuttavia, SSL può essere piuttosto impegnativo per ESP8266 e, quando abilitato, rimane molta meno memoria per l'applicazione in sviluppo. ESP32 cerca di affrontare questi problemi implementando crittografia hardware e firma del codice. La crittografia flash è una funzione per crittografare il contenuto del flash SPI, quindi impedisce la lettura in chiaro del flash crittografato per proteggere il firmware da letture e modifiche non autorizzate. La modalità *Secure Boot* consente a ESP32 di cifrare un firmware con una chiave di cifratura particolare. Questo è fondamentale perché quando viene rilasciato un firmware lo si firma con una chiave che gli consente di essere compatibile con moduli che hanno a loro volta solo quella specifica chiave. Ha un livello di sicurezza maggiore, non permettendo di ricavare il codice sorgente all'interno di un modulo (reverse engineering). In questo modo solo un compilatore con quella chiave di cifratura gli consente di generare codice compatibile.

In Figura 3.3 è mostrato lo schema a blocchi funzionale del chip ESP32 [20]. Il microcontrollore ha un core per protocollo e uno per applicazione, mentre le memorie incorporate sono ROM da 448 kB, SRAM da 520 kB e due orologi RTC da 8 kB. La memoria esterna può supportare fino a quattro volte il flash da 16 MB. Per quanto riguarda la connessione alla rete Internet, ESP32 può attivarsi come STA e SoftAP, cioè può funzionare come stazione ed essere connesso attraverso un router di una rete locale, oppure come server e punto di accesso al fine di fornire un'interfaccia utente, utile in questo progetto per la configurazione dei parametri necessari. Il chip può funzionare in varie modalità di alimentazione: modalità attiva, modalità modem-sleep

(dove la CPU lavora ma WiFi e Bluetooth sono spenti), modalità light e modalità deep-sleep per lavorare a prestazioni inferiori. Tra i diversi canali ADC uno di questi è collegato al sensore Hall integrato per rilevare i campi magnetici, mentre un altro al sensore per monitorare la temperatura del chip con un intervallo da -40 ℃a 125 ℃. Alcuni GPIO sono in grado di rilevare variazioni capacitive e possono essere utilizzati per sensori tattili.

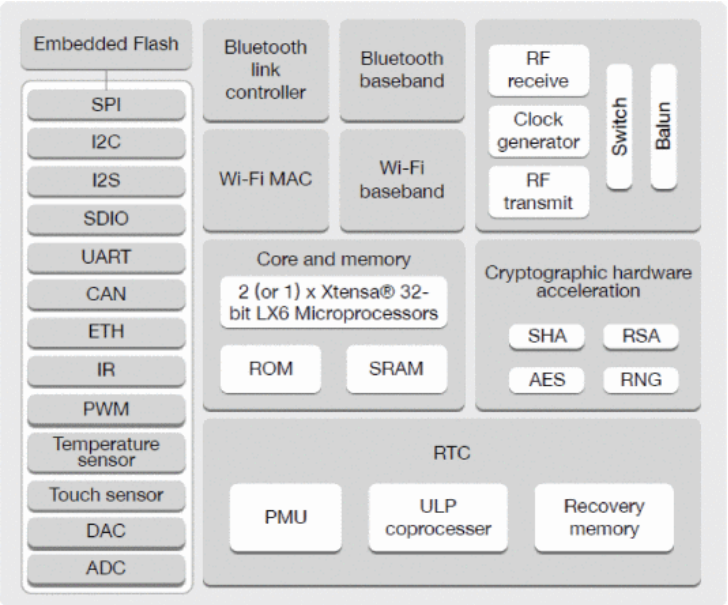


Figura 3.3: Schema a blocchi funzionale ESP32

3.2.2 Sensore di temperatura e umidità SHT21

SHT21, prodotto da *Sensirion* [21] e mostrato in Figura 3.4, possiede un sensore per la misura della temperatura a banda larga, un sensore capacitivo per la misura dell'umidità relativa, un amplificatore, un convertitore analogico-digitale (ADC), una memoria OTP (*One Time Programmable*), un'unità di elaborazione digitale e un'interfaccia I^2C per poter comunicare con il sensore. Ha le seguenti specifiche tecniche:

- *Range di operatività.* $-40℃ < T < +125℃, \quad 0\% < RH < 100\%$
- *Precisione.* $T : \pm 0.3\%, \quad RH : \pm 2.0\%$
- *Ripetibilità.* $T : \pm 0.04℃, \quad RH : \pm 0.1\%$
- *Risoluzione.* $T : 0.04℃, \quad RH : 0.04\%$

Per umidità relativa si intende il rapporto percentuale tra la quantità di vapore contenuta in una massa d'aria e la quantità massima (a saturazione) che il volume d'aria può contenere nelle stesse condizioni di temperatura e pressione.

Invio di un comando

Ogni sequenza di trasmissione di dati inizia con una condizione di start (S) e una condizione di stop (P). Per poter eseguire un'operazione di lettura o scrittura bisogna inviare un comando. Dopo l'invio della condizione di start, l'intestazione I^2C successiva è costituita dall'indirizzo del dispositivo I^2C a 7 bit *1000'000* e da un bit di direzione SDA (lettura R: *1*, scrittura W: *0*). Dopo l'emissione di un comando di misura, il microcontrollore deve attendere il completamento della misura. I comandi di base sono riassunti nella Tabella 3.2. Le modalità *Hold master* o *No Hold master* indicano al sensore se può elaborare altre attività di comunicazione I^2C mentre il sensore sta misurando oppure no.

Figura 3.4: Sensore di temperatura e umidità SHT21

Comando	Commento	Byte
Misura Trigger T	hold master	1110'0011
Misura Trigger RH	hold master	1110'0101
Misura Trigger T	no hold master	1111'0011
Misura Trigger RH	no hold master	1111'0101
Modalità Scrittura	-	1110'0110
Modalità Lettura	-	1110'0111
Soft Reset	-	1111'1110

Tabella 3.2: Comandi di base SHT21

Lettura del dato

In modalità *No Hold master*, il microcontrollore (MCU) deve effettuare un sondaggio per la terminazione dell'elaborazione interna del sensore. Questo viene fatto inviando una condizione di start seguita dall'intestazione I^2C (*1000'0001*) come mostrato in Figura 3.5. Se l'elaborazione interna è terminata, il sensore riconosce il sondaggio e i dati possono essere letti dall'MCU. Se l'elaborazione della misura non è terminata, il sensore invia un bit *NACK* e la condizione di start

deve essere emessa ancora una volta. Il sensore legge tre byte di dati: i primi due sono riferiti alla misura vera e propria della grandezza; il terzo invece è il *checksum*, usato per migliorare l'affidabilità della comunicazione. Poiché la risoluzione massima di una misurazione è di 14 bit, per la trasmissione delle informazioni di stato vengono utilizzati gli ultimi due bit meno significativi (LSB, bit 43 e 44). Il bit 1 dei due LSB indica il tipo di misura (*0*: temperatura, *1* umidità). Il bit 0 non è assegnato. Nell'esempio di Figura 3.5, la lettura eseguita da sensore è

$$S_{RH} = 0110'0011'0101'0000$$

.

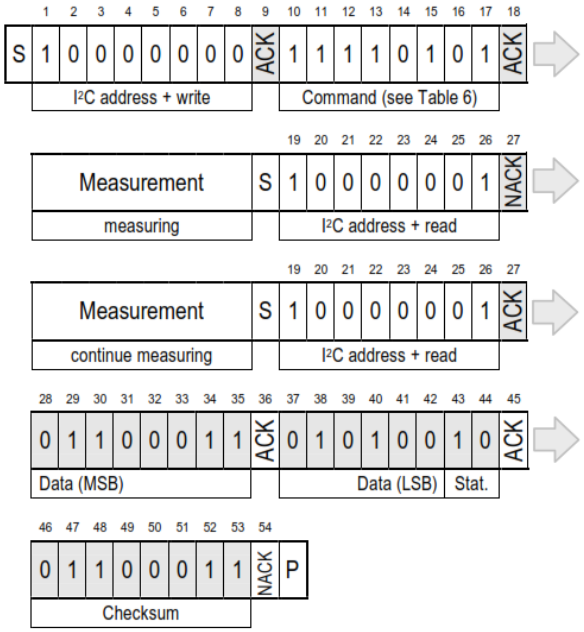


Figura 3.5: Lettura SHT21 in *No Hold master*

La durata massima delle misurazioni dipende dal tipo di misurazione e dalla risoluzione scelta. In Tabella 3.3 sono visualizzati i valori con una risoluzione di 14 bit per la misura della temperatura e di 12 bit per la misura dell'umidità relativa. I valori massimi sono scelti per la pianificazione della comunicazione dell'MCU.

Risoluzione	**RH typ [ms]**	**RH max [ms]**	**T typ [ms]**	**T max [ms]**
14 bit	-	-	66	85
12 bit	22	29	-	-

Tabella 3.3: Tempi di misura SHT21

Conversione umidità relativa

Una volta misurato il dato S_{RH}, l'umidità relativa RH è ottenuto dalla formula seguente:

$$RH = -6 + 125 \cdot \frac{S_{\mathrm{RH}}}{2^{16}} \tag{3.1}$$

Nell'esempio di Figura 3.5 il valore di umidità relativa corrisponde a 42.5% Il valore fisico RH sopra indicato corrisponde all'umidità relativa sopra l'acqua liquida secondo l'*Organizzazione Meteorologica Mondiale* (OMM).

Conversione temperatura

Il valore della temperatura è calcolato inserendo il segnale letto dal sensore S_T nella seguente formula (risultato in ℃):

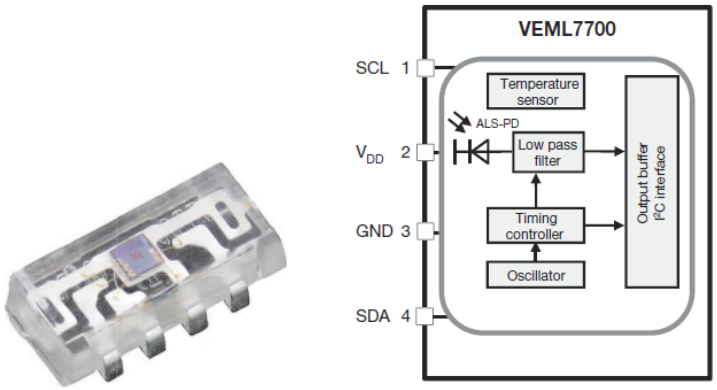
$$T = -46.85 + 175.72 \cdot \frac{S_{\mathrm{T}}}{2^{16}} \tag{3.2}$$

3.2.3 Sensore di luminosità VEML7700

VEML7700 è un sensore digitale ad alta precisione prodotto da *Vishay* [22] e misura la luminosità ambientale con risoluzione a 16 bit. Il sensore è mostrato in Figura 3.6a, mentre la Figura 3.6b mostra il suo schema a blocchi, composto da un fotodiodo ad alta sensibilità, un amplificatore a basso rumore, un convertitore ADC a 16 bit e un'interfaccia di comunicazione I^2C di facile utilizzo. Il risultato della sua misura è pari al valore digitale disponibile di luce ambientale nell'unità di misura *lux*. Il sensore ha un range di operatività da 0 lux a 120 klux e una risoluzione pari a 0.0036 lux/ct. Applicazioni tipiche di VEML7700 sono quelle per l'oscuramento della retroilluminazione di schermi TV o smartphone, per l'accensione e lo spegnimento industriale, o anche come interruttore ottico per dispositivi e display di consumo, informatici e industriali.

Registri di comando

Il sensore contiene sei codici di comando effettivi a 16 bit (Tabella 3.4) per il controllo del funzionamento, la configurazione dei parametri e il buffering dei risultati. Di seguito vengono descritti i registri utilizzati in questo progetto di tesi e sono accessibili tramite comunicazione I^2C:



(a) *Immagine sensore* (b) *Schema a blocchi*

Figura 3.6: Sensore di luminosità VEML7700

- Registro *ALS_CONF_0*. Per le impostazioni di misura della luminosità. Ci sono infatti due grandezze da impostare che vengono utilizzate dal VEML7700:
  - Guadagno (1, 2, 1/8, 1/4)
  - Tempo di integrazione (25, 50, 100, 200, 400, 800 ms)
- Registro *ALS_WH*. Per le impostazioni delle finestre di soglia ad alto limite
- Registro *ALS_WL*. Per le impostazioni delle finestre di soglia a basso limite
- Registro *Power Saving*. Definisce le modalità di risparmio energetico
- Registro *ALS*. Per la lettura del dato rilevato ad alta risoluzione

COMMAND CODE	REGISTER NAME	DESCRIPTION	R / W
0x00	ALS_CONF_0	ALS gain, integration time, interrupt, shutdown	W
0x01	ALS_WH	ALS high threshold window setting	W
0x02	ALS_WL	ALS low threshold window setting	W
0x03	Power Saving	Set (15 : 3) 0000 0000 0000 0b	-
0x04	ALS	ALS bits data	R
0x05	WHITE	WHITE bits data	R
0x06	ALS_INT	ALS INT trigger event	R

Tabella 3.4: Registri di comando *VEML7700*

Interfaccia I^2C

La Figura 3.7 mostra la comunicazione I^2C di base con il sensore VEML7700. L'indirizzo slave del sensore è a 7 bit, seguito da un bit per indicare se il comando è di scrittura o lettura. Per un comando di scrittura viene inviato un segnale di start seguito con l'indirizzo slave e il bit di

modalità. Una volta ricevuto dal VEML7700 il segnale di *ACK*, si invia il codice del registro di comando, uno tra quelli della Tabella 3.4 e si invia il dato costituito da 16 bit seguito dal segnale di stop. Per un comando di lettura, bisogna sempre avviare la richiesta con il bit di scrittura, poi viene di nuovo inviato uno start, l'indirizzo slave e il bit di lettura. I due byte successivi contengono la misura della luminosità.

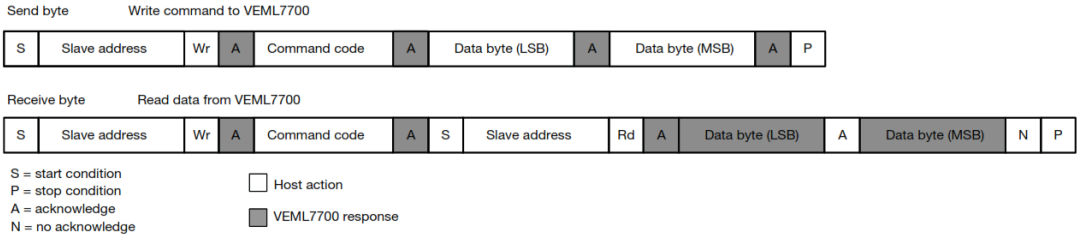


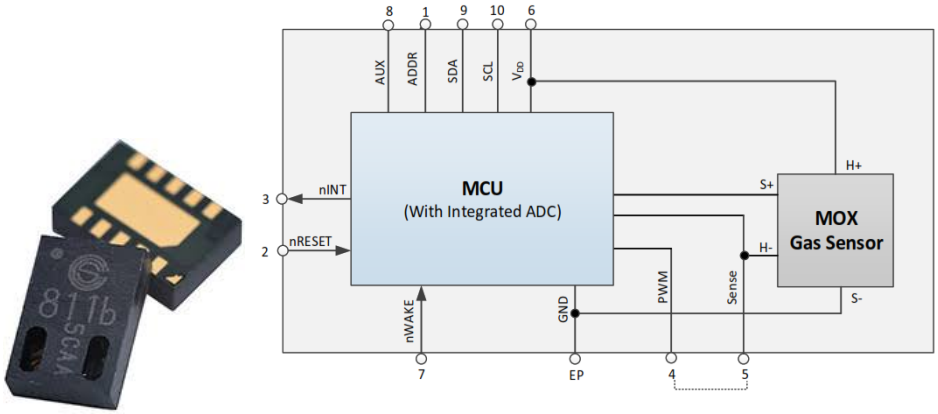
Figura 3.7: Interfaccia I^2C *VEML7700*

3.2.4 Sensore di qualità dell'aria CCS811

Il sensore CCS811 è prodotto da *ams* [23] in grado di rilevare la qualità dell'aria, in particolare, misura la quantità di anidride carbonica CO_2, in *ppm* e il TVOC (*Total Volatile Organic Compounds*), in *ppb*, cioè un parametro che misura tutte quelle sostanze aventi comportamenti fisici e chimici differenti, ma caratterizzati da una certa volatilità, ovvero sono sostanze che evaporano facilmente già ad una bassa temperatura. Negli ambienti indoor sono generati principalmente da materiali edili, isolanti, arredi, rivestimenti, ma anche da materiali naturali, prodotti per l'igiene personale e della casa. CCS811 integra un sensore di gas ossido di metallo (MOX) per rilevare un vasta gamma di composti organici volatili (VOC), un'unità di microcontrollore (MCU), che include un convertitore analogico-digitale e un'interfaccia I^2C. L'MCU integrata gestisce le modalità di azionamento dei sensori e le modalità di dati del sensore misurati durante il rilevamento di VOC, causati principalmente dall'uomo.

Il range di operatività per la CO_2 è da 400 a 8192 ppm, mentre per il TVOC, è da 0 a 1187 ppb. Principali applicazioni del sensore sono gli smartphone, la domotica, l'automazione degli edifici e tanti tipi di accessori, anche indossabili. Il sensore può lavorare in cinque modalità di funzionamento:

- *Modalità 0.* Inattivo, a bassa corrente.
- *Modalità 1.* A potenza costante, esegue le misure ogni secondo.
- *Modalità 2.* Riscaldamento a impulsi, misura ogni dieci secondi.



(a) *Immagine sensore* (b) *Schema a blocchi*

Figura 3.8: Sensore di qualità dell'aria CCS811

- *Modalità 3.* Riscaldamento a impulsi a bassa potenza, misura ogni minuto.
- *Modalità 4.* A potenza costante, misura ogni 250 ms.

Nelle modalità 1, 2 e 3, la concentrazione di CO_2 equivalente e la concentrazione di TVOC viene calcolata per ogni campione. La modalità 1 reagisce più velocemente alla presenza di gas, ma ha una maggiore corrente di esercizio, mentre la modalità 3 reagisce più lentamente alla presenza di gas ma ha il la più bassa corrente media di funzionamento.

La Tabella 3.5 mostra i registri del sensore con una breve descrizione per ognuno. Da notare il registro *ENV_DATA* che permette di memorizzare i valori di temperatura e di umidità misurati da un sensore esterno, usati per compensare le letture di CO_2 e TVOC.

Interfaccia I^2C

Le transazioni I^2C richiedono la selezione di un indirizzo di registro, che corrisponde a uno o più byte e seguito dai dati. A differenza degli altri sensori, il CCS811 ha un pin *nWAKE* che deve essere portato a massa prima di una transazione I^2C, almeno per un tempo *tAWAKE* prima della transazione e mantenuto basso per tutto il tempo. La transazione inizia come al solito con una condizione di start, seguita dall'indirizzo slave, dal bit che indica il tipo di modalità lettura o scrittura e infine dal segnale di acknowledge. Dopodiché viene inviato l'indirizzo del registro desiderato e i dati da scrivere o leggere. Come mostrato in Figura 3.9, in un'operazione di scrittura, si inviano due byte di cui uno è l'indirizzo di registro e l'altro il dato da scrivere. Per quanto riguarda la lettura, come negli altri sensori, durante l'I^2C non è possibile fornire un indirizzo di un registro letto, perciò è necessario prima preparare una scrittura che andrà a selezionarlo, poi si invia di nuovo l'indirizzo slave e infine si ottiene il dato da leggere. Si

Indirizzo	Registro	R / W	Size	Descrizione
0x00	STATUS	R	1 byte	Registro di stato.
0x01	MEAS_MODE	R / W	1 byte	Modalità di misura e condizioni.
0x02	ALG_RESULT_DATA	R	fino a 8 byte	Risultato dell'algoritmo. I 2 byte più significativi contengono una stima del livello di CO2, mentre i 2 byte successivi contengono la stima del TVOC.
0x03	RAW_DATA	R	2 byte	Valori di dati ADC grezzi per la resistenza e la fonte di corrente usato.
0x05	ENV_DATA	W	4 byte	Valori di temperatura e umidità relativa possono essere scritti in questo registro per la compensazione delle misure.
0x06	NTC	R	4 byte	Fornisce la tensione attraverso la resistenza di riferimento e la tensione attraverso la resistenza NTC,da cui il è possibile determinare la temperatura ambiente.
0x10	THRESHOLDS	W	5 byte	Le soglie per il funzionamento quando le interruzioni sono solo generate quando la CO2 supera una soglia.
0x11	BASELINE	R / W	2 byte	Il valore codificato della corrente di base può essere letto. Un valore calcolato precedentemente può essere scritto.
0x20	HW_ID	R	1 byte	ID hardware. Il valore è 0x81.
0x21	HW Version	R	1 byte	Versione hardware. Il valore è 0x1X
0x23	FW_Boot_Version	R	2 byte	Versione di avvio (BOOT) del firmware. I primi 2 byte contengono il numero della versione.
0x24	FW_App_Version	R	2 byte	Versione dell'applicazione firmware. I primi 2 byte contengono il numero della versione.
0xE0	ERROR_ID	R	1 byte	ID di errore. Quando il registro di stato segnala un errore, la fonte si trova in questo registro.
0xFF	SW_RESET	W	4 byte	Se vengono scritti i 4 byte corretti (0x11 0xE5 0xE5 0x72 0x8A) a questo registro in un'unica sequenza, il dispositivo si resetta e torna in modalità BOOT.

Tabella 3.5: Registri delle applicazioni CCS811

possono inoltre leggere o scrivere più byte di dati alla volta. Le due operazioni possono essere anche combinate opzionalmente in una singola transazione utilizzando una condizione di partenza ripetuta.

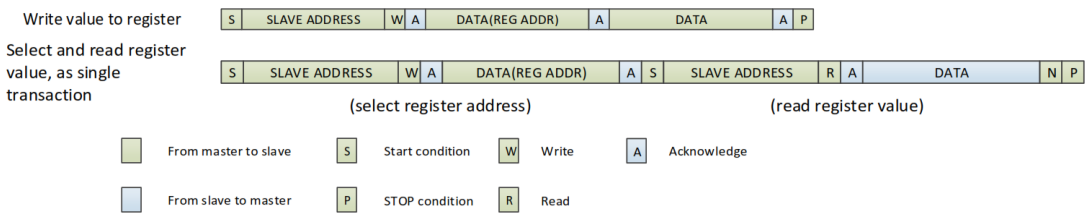


Figura 3.9: Interfaccia I^{2}C CCS811

3.2.5 I^{2}C Expander PCF8575

Come già anticipato, DONET possiede 12 LED RGB che vengono accesi a seconda dell'attività che sta svolgendo. Per poterli controllare con un unico segnale I^{2}C si utilizzano gli I/O expander

PCF8575 realizzati da *Texas Instruments* [24]. Il PCF8575 fornisce un'espansione I/O remota per la maggior parte delle famiglie di microcontrollori ed è dotato di una porta di input e output quasi bidirezionale a 16 bit, come mostrato nello schematico di Figura 3.10. Prima di leggere dal PCF8575 tutte le porte desiderate come ingressi devono essere impostate a livello logico alto.

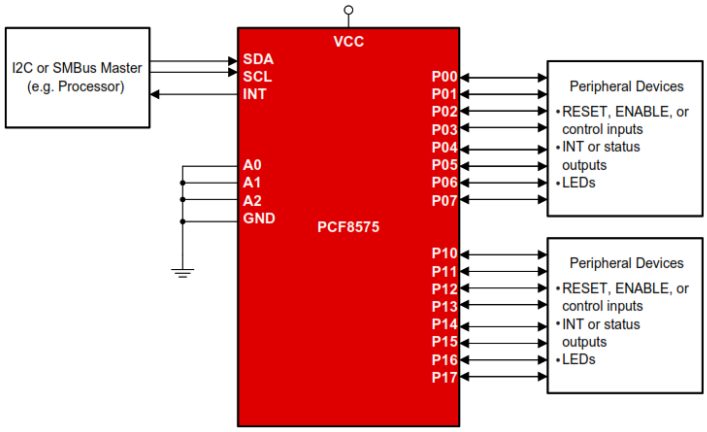


Figura 3.10: Schematico PCF8575

Interfaccia I^2C

La comunicazione I^2C con il dispositivo è avviata dal master inviando un segnale di start seguito dall'indirizzo del PCF8575, incluso il bit di selezione della modalità scrittura o lettura. L'indirizzo viene scelto secondo la Tabella 3.6, dove si possono selezionare a piacere i bit *A2*, *A1* e *A0*, in modo da poter utilizzare più dispositivi insieme. Infatti, in questo progetto sono stati usati tre PCF8575 e ognuno controlla l'accensione di un colore dei LED (uno per il rosso, uno per il verde e uno per il blu). Se l'indirizzo è valido, l'expander risponde con un segnale di ACK. Il byte di dati segue il segnale di ACK: se il bit R/W è alto, allora si esegue una lettura, mentre se è basso si esegue la scrittura. Finita l'operazione, il master invia un segnale di stop.

BYTE	**BIT**							
	7 (MSB)	**6**	**5**	**4**	**3**	**2**	**1**	**0 (LSB)**
I2C slave address	L	H	L	L	A2	A1	A0	R/W
P0x I/O data bus	P07	P06	P05	P04	P03	P02	P01	P00
P1x I/O data bus	P17	P16	P15	P14	P13	P12	P11	P10

Tabella 3.6: Definizione dell'interfaccia PCF8575

3.3 Broker MQTT

Come descritto in Paragrafo 1.5, il broker MQTT ha un ruolo importante nella WSN perché raccoglie i dati rilevati dai nodi wireless e li trasmette a un database centrale. Nel progetto di questa tesi, si è scelto di installare in un computer Raspberry un broker Mosquitto MQTT [1], cioè un single-board computer progettato per ospitare sistemi operativi basati su kernel Linux o RISC OS. Si è scelto Mosquitto perchè è open source ed è il più diffuso. La versione di Raspberry di cui si fa uso è la Raspberry Pi 3 Model B avente le seguenti caratteristiche hardware e software:

- SOC Broadcom BCM2837;
- CPU da 1.2 GHz 64-bit quad-core ARM Cortex-A53;
- Memoria SDRAM: 1 GB LPDDR2 (900 MHz);
- Sistema operativo: Ubuntu 18.04.3 LTS.

Per installare questo servizio in un computer con sistema operativo Ubuntu è sufficiente aprire il terminale ed eseguire i seguenti comandi

```
$ sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
$ sudo apt-get update
$ sudo apt-get install mosquitto
```

Installato il broker MQTT si può installare anche il client

```
$ sudo apt-get install mosquitto-clients
```

Dopo aver eseguito questi comandi il server sarà attivo. Per verificare il suo stato, è possibile digitare il comando

```
$ sudo service mosquitto status
```

Per fermare il server MQTT è sufficiente digitare il comando

```
$ sudo service mosquitto stop
```

Per avviare il server MQTT si esegue

```
$ sudo service mosquitto start
```

3.4 Funzionamento DONET

In questo paragrafo viene descritto in dettaglio il funzionamento del nodo wireless DONET. Si presenta l'algoritmo implementato, il cui codice scritto in linguaggio MicroPython è riportato in Capitolo 5. Viene mostrato anche il processo di configurazione del dispositivo al primo utilizzo.

3.4.1 Algoritmo implementato

Il diagramma di flusso di Figura 3.11 illustra la struttura generale del programma. Al primo avvio di DONET è necessario eseguire la configurazione iniziale alla quale si accede tenendo premuto per un tempo di almeno 8 secondi il tasto BOOT del dispositivo. DONET inizia ad attivarsi in modalità WiFi Soft Access Point al quale connettersi attraverso un computer o uno smartphone, permettendo di aprire la pagina web di configurazione in cui inserire i dati richiesti. Una volta avvenuta con successo questa operazione, DONET riceve i parametri inseriti e li memorizza in un file scritto in formato JSON, permettendo di strutturare i dati in modo semplice così il dispositivo è in grado di leggerlo dopo il riavvio e recuperare le informazioni. Le credenziali del WiFi locale vengono invece salvate in un file *.dat*. Una volta riavviato DONET si attiva in modalità WiFi Station utilizzando le credenziali per connettersi alla rete locale impostata e procede con l'inizializzazione di: il bus I^2C, i sensori SHT21, VEML7700 e CCS811 e la comunicazione con il gateway della rete attraverso il protocollo MQTT. A questo punto DONET è pronto a compiere le attività principali, ovvero la lettura dei dati provenienti dai sensori, la connessione al broker MQTT e invio delle misure. Le misure vengono fatte ad ogni intervallo di tempo impostato durante la fase di configurazione del dispositivo, perciò è necessario un conteggio del tempo che gestisce la chiamata dei sensori. DONET misura anche da quanto tempo è connesso, inviando un messaggio ogni 10 secondi al broker MQTT con l'indicazione del tempo trascorso in secondi, minuti, ore e giorni. Il controllo di un aggiornamento OTA lo richiede l'utente quando vuole premendo il tasto BOOT per un tempo superiore a 4 secondi e inferiore a 8. Se l'aggiornamento è presente allora viene avviato il download e l'esecuzione del nuovo codice dopo un riavvio della scheda. DONET possiede 12 LED RGB che si illuminano in base all'attività che sta svolgendo:

- *Rosso fisso*, significa che non è connesso alla rete WiFi o ha perso la connessione
- *Rosso e blu alternati*, indica che è stata persa la connessione MQTT
- *Blu fisso*, segnala lo stato in modalità di configurazione

- *Verde rotante*, indica l'avvenuta lettura e invio al broker MQTT delle misure provenienti dai sensori

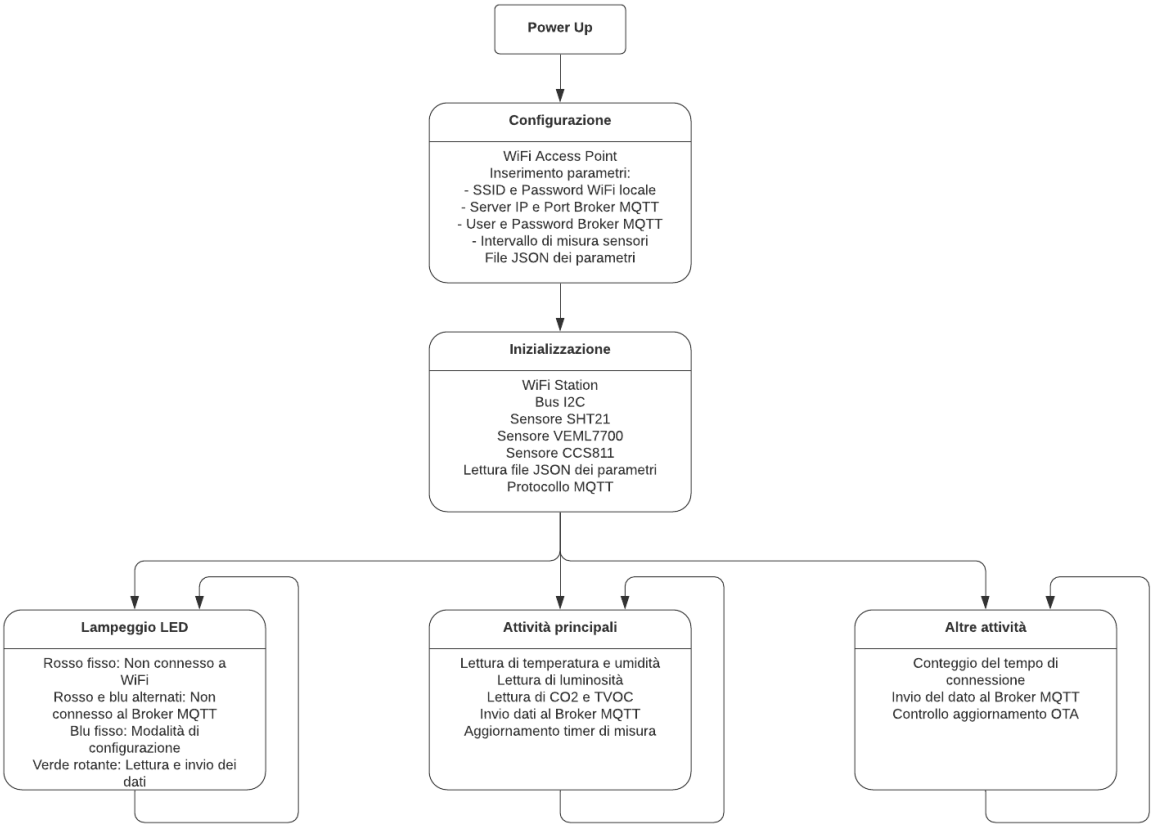


Figura 3.11: Diagramma di flusso dell'algoritmo DONET

3.4.2 Configurazione DONET

La configurazione di DONET viene fatta tramite un computer o uno smartphone e un modem WiFi. Premendo il tasto BOOT per un tempo di almeno 8 secondi, si accede alla modalità di configurazione e si dovrebbe accendere di colore blu fisso. Si utilizza il computer per connettersi alla rete generata dal dispositivo e chiamata *WSNstation - DONET*. In questo modo il chip si attiva come SoftAP, cioè è possibile collegarsi alla rete ma non si può navigare in Internet, perché il chip ESP32 non si connette ulteriormente ad una rete cablata come il router. Per poter accedere alla schermata di configurazione mostrata in Figura 3.12 si accede all'indirizzo *http://192.168.4.1/*. In questa schermata sono mostrate le reti WiFi che il dispositivo ha trovato nelle vicinanze, bisogna quindi selezionare quella alla quale si desideri connettere. In basso sono invece riportati gli altri campi da impostare, ovvero l'indirizzo IP del server MQTT, la porta

MQTT, il nome utente e password del broker e l'intervallo di tempo in cui effettuare le misure dei sensori. Il programma è stato scritto impostando questi valori (tranne le credenziali WiFi) di default nel caso in cui non si dovessero riempire tutti i campi. L'intervallo di default di misura è di 60 secondi. Una volta inviati i dati, DONET dovrebbe connettersi alla rete aprendo una nuova schermata di conferma. A questo punto, il dispositivo esce dalla fase di configurazione ed è pronto ad elaborare.

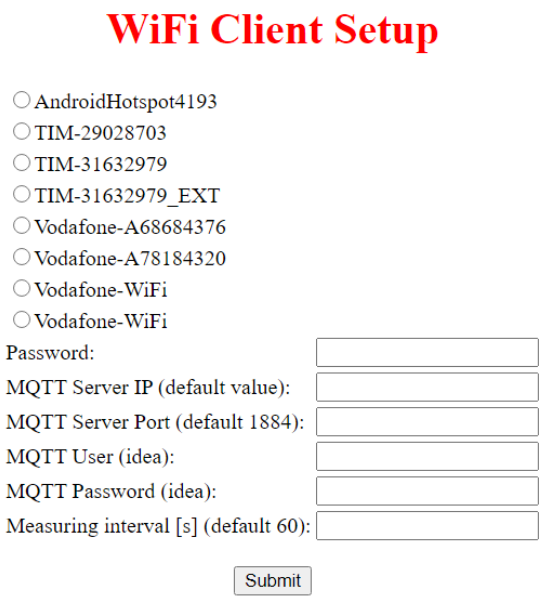


Figura 3.12: Schermata di configurazione DONET

Capitolo 4

MICROPYTHON

MicroPython è un'implementazione lineare ed efficiente della versione 3.x del noto linguaggio Python. Il sistema include un sottoinsieme delle librerie standard del linguaggio, ed è ottimizzato per essere eseguito su microcontrollori, o in generale su sistemi embedded in cui, come noto, le risorse a disposizione sono limitate e devono pertanto essere utilizzate in modo ottimale. Tuttavia, include moduli per accedere a hardware di basso livello, ciò significa che ci sono librerie per accedere e interagire facilmente con i GPIO. L'obiettivo di MicroPython è rendere la programmazione dell'elettronica digitale il più semplice possibile, in modo che possa essere utilizzata da chiunque.

4.1 Caratteristiche di MicroPython

MicroPython è sviluppato nel linguaggio C ed è un progetto interamente open source. Il codice è disponibile a tutti su *GitHub* [25] e chiunque è libero di scaricarlo per studio o per contribuire al miglioramento e perfezionamento. Possiede delle tecniche di codifica avanzate per mantenere le stesse funzionalità di Python pur ottenendo compilati di piccola dimensione, per potersi adattare alle scarse risorse dei microcontrollori. La potenza di MicroPython sta proprio nel compilatore. Infatti questo software ha al suo interno una serie di moduli hardware-specifici (come per esempio *machine*) in modo da poter compilare il codice in Python adeguatamente a seconda del microcontrollore utilizzato.

Inoltre, MicroPython è estremamente compatto: per eseguire il suo codice sono infatti sufficienti 256 kB di memoria flash e 16 kB di memoria RAM. L'obiettivo dichiarato di MicroPython è quello di raggiungere il più elevato grado di compatibilità con la versione standard di Python, in modo tale che sia possibile trasferire sul microcontrollore oppure sul sistema embedded lo stesso codice sviluppato in ambiente desktop. L'interfaccia utente di MicroPython è rappresentata da un prompt interattivo denominato *REPL* (*Read-Eval-Print Loop*). Si tratta in sostanza di una

shell interattiva molto semplice, in grado di ricevere gli input dell'utente, processarli, e restituire il risultato dell'operazione. REPL permette sia di eseguire i comandi (o script) immediatamente, sia di importare ed eseguire script dal filesystem. Consente quindi di connettersi a una scheda ed eseguire rapidamente il codice senza la necessità di compilarlo o caricarlo.

I codici sono espressivi e robusti in modo che i concetti logici vengano implementati con un minor numero di righe. MicroPython in una scheda come la ESP32 ha il supporto periferico di GPIO, ADC, PWM, I^2C/SPI e anche delle connessioni WiFi. Ecco alcune caratteristiche che sono state pubblicate sul sito ufficiale [26]:

- alta configurabilità grazie a molte opzioni di configurazione a tempo di compilazione
- molte architetture supportate (x86, x86-64, ARM, ARM Thumb, Xtensa)
- sintassi semplice e pulita per controllare con il 93% di copertura del codice
- tempo veloce di start-up dal boot per caricare il primo script
- un semplice, veloce e robusto mark-sweep garbage collector per la memoria heap
- lancio dell'eccezione *MemoryError* nel caso dell'esaurimento dell'heap
- lancio dell'eccezione *RuntimeError* quando si è raggiunto lo stack limit
- supporto per il codice Python in esecuzione con un hard interrupt con latenza minima
- tutti gli errori hanno un backtrace e riportano il numero della riga di codice sorgente relativa
- un cross-compiler ed il frozen bytecode, in modo da avere script pre-compilati che non occupano RAM
- un emitter nativo che scrive il codice macchina direttamente piuttosto che il bytecode per la virtual machine

Gli altri linguaggi di programmazione, come Arduino per esempio, vengono utilizzati generalmente per scrivere un programma nella sua interezza, ed una volta corretto, viene compilato ed inviato al microcontrollore per essere eseguito (generalmente in loop). Un linguaggio come il C/C++ è più lento rispetto alla scrittura e al debug del codice, infatti il debug della programmazione per computer che individua e corregge i bug nel codice del programma può costare circa il 20% - 25% di un grande progetto software. Pertanto, è importante scegliere una programmazione adeguata tenendo conto dei costi del progetto.

Python invece, fa uso di un interprete, quindi si può utilizzare per scrivere programmi e lavorare in runtime, lanciando un comando alla volta. Ecco, con MicroPython è possibile scrivere un comando alla volta da console, che viene compilato in tempo reale e inviato al microcontrollore. Confrontando la sua velocità di scrittura e debug, risulta 5-10 volte più veloce dei linguaggi di programmazione tradizionali. La velocità della programmazione del computer è definita dalla relativa risposta quando viene fornito il suo ingresso. Le variabili utilizzate nel linguaggio di programmazione si associano al suo nome e al tipo di dati che saranno diversi. Il controllo del tipo di tali variabili può essere fatto sia durante il tempo di compilazione in contrapposizione al tempo di esecuzione, sia durante il tempo di esecuzione in contrapposizione al tempo di compilazione. I linguaggi di programmazione come C, C++, Java sono detti di tipizzazione statica in quanto il loro controllo di tipo viene fatto durante il tempo di compilazione in contrapposizione al tempo di esecuzione. D'altra parte, i linguaggi di programmazione come Lua, JavaScript, MicroPython sono detti di digitazione dinamica in quanto il loro controllo di tipo viene fatto durante il tempo di esecuzione, in contrapposizione al tempo di compilazione, questo risulterà in una risposta più rapida. La Tabella 4.1 riassume il confronto tra i linguaggi di programmazione MicroPython e Arduino C/C++ [27].

Caratteristiche	**MicroPython**	**Arduino C/C++**
Tipo di linguaggio	Interprete del linguaggio di scrittura	Linguaggio di compilazione
Complessità di linguaggio	Più semplice	Complesso
Implementazione	Nessuna fase di compilazione o di caricamento separata	Necessarie una compilazione e un caricamento
Gestione della memoria	Automatica, allocazione della memoria per una nuova variabile non è necessaria	Utilizza i puntatori per gestire e accedere alla memoria
Sintassi	Codici più puliti e semplici senza l'uso di parentesi e punti e virgola	Parentesi e punti e virgola sono obbligatori
Tipizzazione	Dinamica	Statica
Velocità di scrittura e di debug	5 - 10 volte più veloce	Più lento
Leggibilità	Migliore	Peggiore

Tabella 4.1: Confronto linguaggi di programmazione MicroPython - C/C++

4.2 MicroPython in ESP

In questa sezione sono illustrati i passaggi per programmare un chip ESP con MicroPython, ovvero con il firmware che svolge la funzione di interprete per il MicroPython. Una volta installato il firmware è possibile caricare gli script di codice sulla board ed eseguirli. Per l'installazione di MicroPython è necessario usare un PC con installato *Ubuntu 18.04.3 LTS* oppure si può utilizzare una macchina virtuale come *VirtualBox*. Questa guida inizia dalla configurazione del PC con il software Python 3.x, per poi installare *esptool.py*, necessario per il caricamento del firmware MicroPython.

4.2.1 Configurazione PC

Si apre il terminale e si installa Python sul PC digitando:

```
$ sudo apt-get update
$ sudo apt-get install python3.8
```

Installare il package manager per Python *PIP* con il comando:

```
$ sudo apt-get install python3-pip
```

A questo punto è possibile scaricare il tool *esptool.py* che consente di caricare il firmware e gli script nei chip *Espressif*:

```
$ pip install esptool
```

Il tool sarà installato nella directory degli eseguibili Python predefinita. Prima di proseguire con l'installazione, è necessario assicurarsi che la scheda sia visibile sulla porta USB, cioè che l'interfaccia di connessione seriale sia riconosciuta. Controllare sulla *Gestione dispositivi* di Windows, se non si trova, allora sarà necessario installare il driver. Una volta fatto questo, è necessario annotarsi il nome della porta utilizzata dal PC per la connessione con la board.

4.2.2 Flash ESP32 con MicroPython

Si può pensare a MicroPython come al sistema operativo sul chip ESP, quindi bisogna installarlo prima di poter eseguire qualsiasi programma. Andando nella pagina di download di MicroPython è possibile scaricare il file *.bin* ESP32. Bisogna scegliere l'ultima versione stabile, come indicato dalla freccia in Figura 4.1.

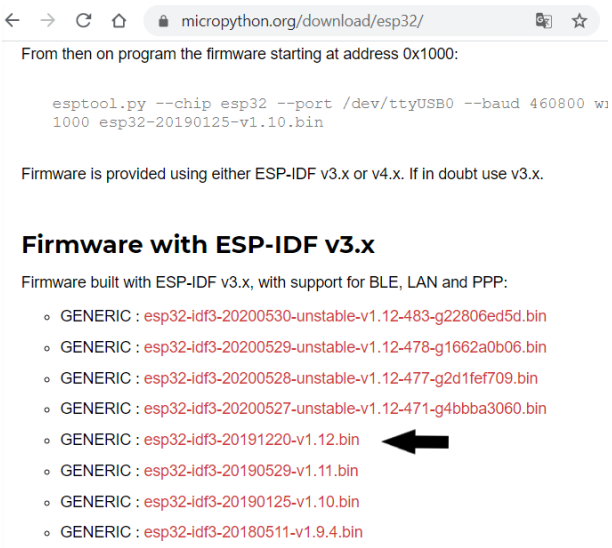


Figura 4.1: Download firmware MicroPython

Prima di eseguire il flashing del firmware è necessario cancellare la memoria flash di ESP32 con il comando:

```
$ esptool.py --chip esp32 erase_flash
```

Per cancellare la flash, il chip deve trovarsi nella modalità *BOOT*. In molte board è presente un tasto per utilizzare questa modalità, basterà tenerlo premuto e poi eseguire il comando. A questo punto è possibile installare il firmware, eseguendo (sempre tenendo premuto il pulsante *BOOT*) il comando:

```
$ esptool.py --chip esp32 --port <porta_seriale> write_flash -z 0x1000 <
    path_firmware>
```

dove alla voce *<porta_seriale>* sarà necessario scrivere il nome della porta seriale a cui è connessa la scheda, mentre in *<path_firmware>* va messo il percorso contenente il file *.bin* corrispondente all'immagine del firmware. MicroPython è stato installato correttamente nel chip ESP32. A questo punto è possibile iniziare a programmare la scheda. Ci sono due importanti file che MicroPython cerca nella root del suo filesystem. Questi file contengono il codice MicroPython che verrà eseguito ogni volta che la scheda viene accesa o resettata. Questi file sono:

- ***boot.py***. Questo file viene eseguito per primo all'accensione/reset e dovrebbe contenere codice di basso livello che imposta la scheda per completare l'avvio. Imposta diverse opzioni di configurazione come le credenziali di rete, l'importazione delle librerie, l'impostazione dei pin, ecc.

- ***main.py***. Viene eseguito dopo *boot.py* e dovrebbe contenere qualsiasi script principale che si voglia eseguire quando la scheda viene accesa o resettata. Lo script *main.py* è usato per eseguire il proprio codice ogni volta che una scheda MicroPython si accende.

Bisogna tenere conto che se si copia all'interno della scheda un altro file Python *.py*, non verrà eseguito a meno che non sia chiamato da *main.py*. Una volta caricato il codice sulla scheda, per eseguirlo bisogna premere il tasto di abilitazione *EN/RESET*.

4.3 Gestione memoria ESP

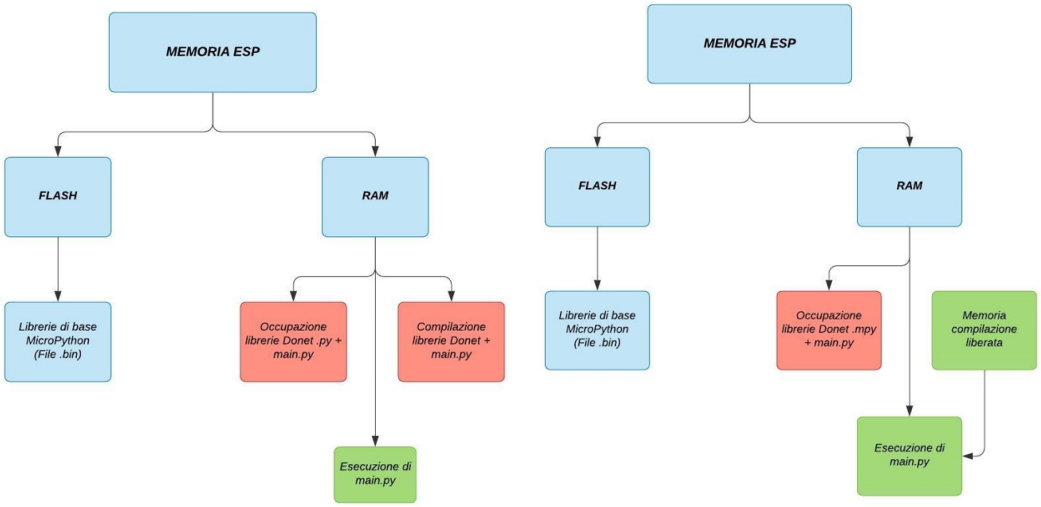
In questo paragrafo è descritto come gestire la memoria dei chip ESP, importante da considerare per eseguire i firmware in MicroPython. I chip ESP8266 ed ESP32 possiedono una memoria flash ROM e una SRAM. Esistono tre metodi di caricamento del codice sulla scheda in MicroPython. In Figura 4.2 sono mostrati dei diagrammi per aiutare a comprendere il funzionamento delle tre modalità.

1. **Moduli .py**. Sono compilati in fase di runtime della scheda e inseriti in RAM. Come si può vedere in Figura 4.2a, il file *.bin* che contiene le librerie di base MicroPython viene caricato nella memoria flash ROM. La RAM è invece suddivisa in tre parti: la prima è quella occupata fisicamente dal codice cioè dalle librerie DONET e dal file *main.py*; la seconda è quella dedicata alla compilazione in runtime del codice; la terza è quella che rimane disponibile per l'esecuzione del programma. In questo modo la RAM viene completamente occupata dall'occupazione del codice e dalla sua compilazione, generando errori durante l'importazione delle librerie nel file *main.py*. Questo metodo si può usare di conseguenza soltanto con codici molto brevi.

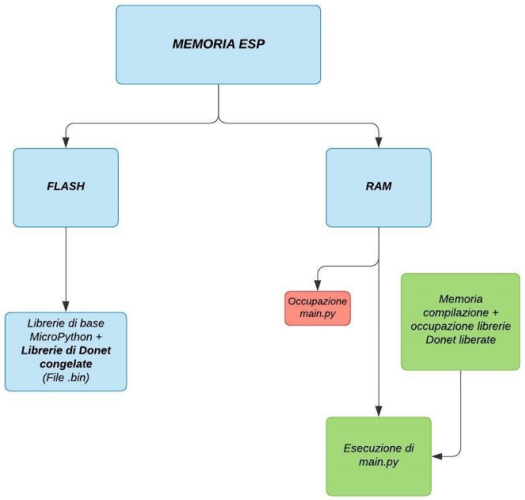
2. **Moduli .mpy**. Sono pre-compilati sul PC e inseriti in RAM (Figura 4.2b). Si utilizza il compilatore esterno *mpy-cross*, che esegue una pre-compilazione del codice sul PC convertendo i file *.py* in *.mpy*, riducendo notevolmente l'occupazione in memoria del codice, ma li salva allo stesso modo in RAM. Si ha quindi un guadagno in memoria RAM di quella parte che prima era destinata alla compilazione del codice che ora viene fatta esternamente. Ma se si lavorasse con codici ancora troppo lunghi, si potrebbero comunque avere problemi di allocazione di memoria durante l'esecuzione del codice.

3. **Moduli congelati**. Sono pre-compilati sul PC ma inseriti in flash. Questo è il metodo da seguire per i codici più lunghi. Il file *.bin* che contiene le librerie di base in MicroPython può essere creato da zero. Questo significa che è possibile generare un nuovo firmware

MicroPython che oltre alla librerie di base ingloba anche quelle che si desiderano. Per costruire questo file è necessario utilizzare la toolchain *esp-open-sdk*, che contiene tutti gli script di integrazione per creare un SDK standalone completo per lo sviluppo di software con i chip ESP. Osservando la Figura 4.2c si può notare che il congelamento dei moduli permette di inglobare le proprie librerie e salvarle non più nella RAM ma nella flash, permettendo di liberare tutta la memoria in RAM. La RAM occupa soltanto il codice del programma soggetto a modifiche (perché i file congelati non sono modificabili) e tutto il resto può essere utilizzato per l'esecuzione del codice.



(a) *Moduli .py* (b) *Moduli .mpy*



(c) *Moduli congelati*

Figura 4.2: Gestione memoria ESP

Scaricando la libreria MicroPython in [25] è possibile seguire la guida per utilizzare il cross-

compilatore *mpy-cross* e convertire i moduli *.py* in *.mpy*. La tecnica dei moduli congelati [28, 29] è stata usata per il chip ESP8266, avendo molta meno memoria disponibile rispetto a ESP32. Queste procedure utilizzano comandi in Linux.

4.4 MicroPython in PyCharm IDE

PyCharm è un ottimo IDE multipiattaforma pensato per lo sviluppo di applicazioni in Python. Dopo la scrittura del codice è possibile affidarsi al debugging tool incluso in PyCharm per verificare la presenza di bug e imperfezioni del codice. In breve sono riportate le feature principali di PyCharm:

- coding assistance/analysis, con code completion, syntax ed error highlighting, linter integration e sistema per il quick fixing
- project/code navigation: vista rapida della struttura dei file e quick jumping tra i file, classi e metodi
- python refactoring: rename, gli extract method, gestione di variabili e costanti, pull up e push down
- supporto nativo per i web frameworks Django, web2py e Flask
- python debugger e unit testing, con code coverage line-by-line
- supporto per il Google App Engine
- controllo di versione integrato con UI per la gestione unificata di Mercurial, Git, Subversion, Perforce e CVS

Quello che più interessa è il *plug-in MicroPython* per PyCharm, che consente di modificare il codice MicroPython e di interagire con i microcontrollori basati su MicroPython utilizzando PyCharm. Supporta i dispositivi ESP8266/32, Pyboard e BBC Micro: bit. PyCharm non è in grado di eseguire il flashing della scheda ESP32 con il firmware MicroPython, quindi prima di tutto bisogna seguire la procedura descritta in Paragrafo 4.2.2.

Per installare il plug-in MicroPython, bisogna aprire PyCharm e selezionare *File > Setting > Plugins*, cercare il plug-in MicroPython e installarlo (Figura 4.3). Ora bisogna creare un nuovo progetto. Come mostrato in Figura 4.4, selezionare *File > New Project > Pure Python* e scegliere la locazione del progetto e l'interprete Python più recente.

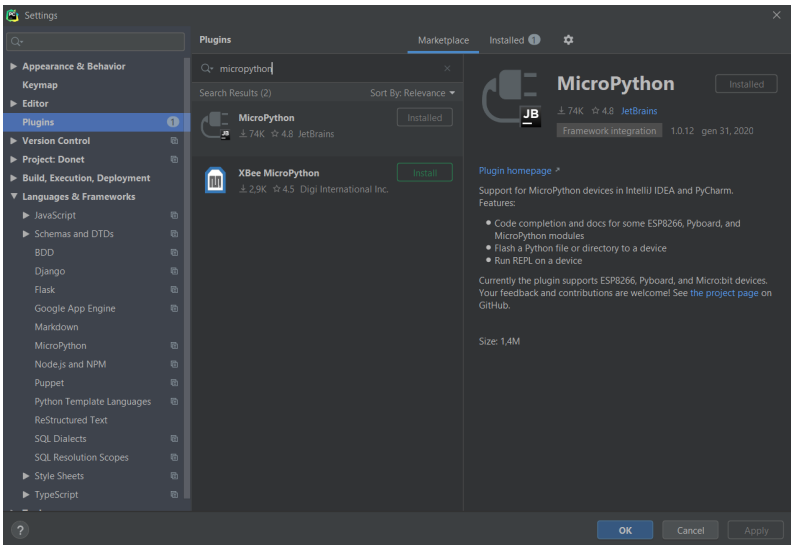


Figura 4.3: Plug-in MicroPython

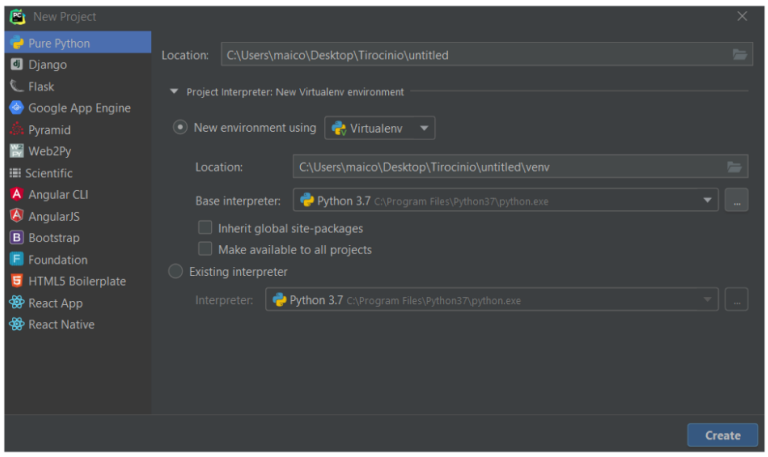


Figura 4.4: Nuovo progetto in PyCharm

Successivamente è necessario abilitare il supporto MicroPython. (Figura 4.5). Andare su *File > Settings > Languages & Frameworks > MicroPython* e specificare il dispositivo e la porta alla quale è collegato. Se si programma una ESP32 selezionare comunque la voce *ESP8266*, funzionerà lo stesso. A questo punto è possibile scrivere uno script Python da eseguire sulla board. Appena si apre un nuovo file, appare un avviso di installazione di alcuni pacchetti richiesti da MicroPython, quindi basta cliccare sul pulsante relativo e verranno installati in automatico. Dopo aver scritto i file Python bisogna caricarli sulla scheda ESP32. Prima di fare ciò è importante sapere che quando si crea un progetto su PyCharm, al suo interno, viene creata una cartella *.idea* per le impostazioni specifiche di PyCharm, che non deve essere caricata sulla board, perchè sarebbe inutile e occuperebbe spazio, perciò bisogna escluderla. Come si vede in Figura 4.6, selezionare *File > Settings > Project: 'nome_progetto' > Project Structure*. Notare che le cartelle escluse sono di colore arancione, quindi si seleziona *Excluded* e poi la cartella che si vuole escludere.

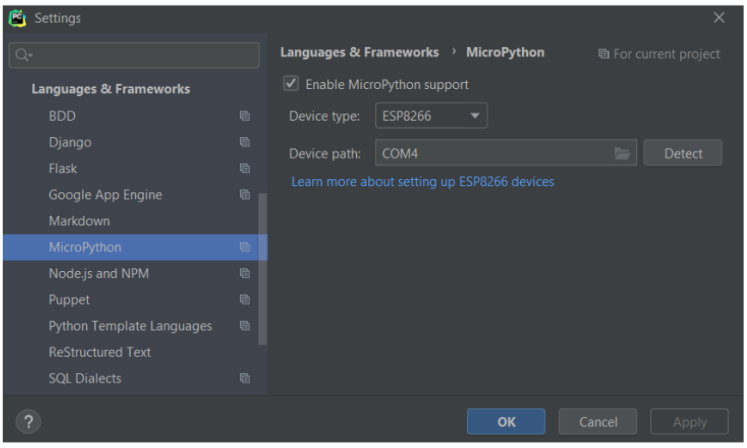


Figura 4.5: Abilitazione MicroPython in PyCharm

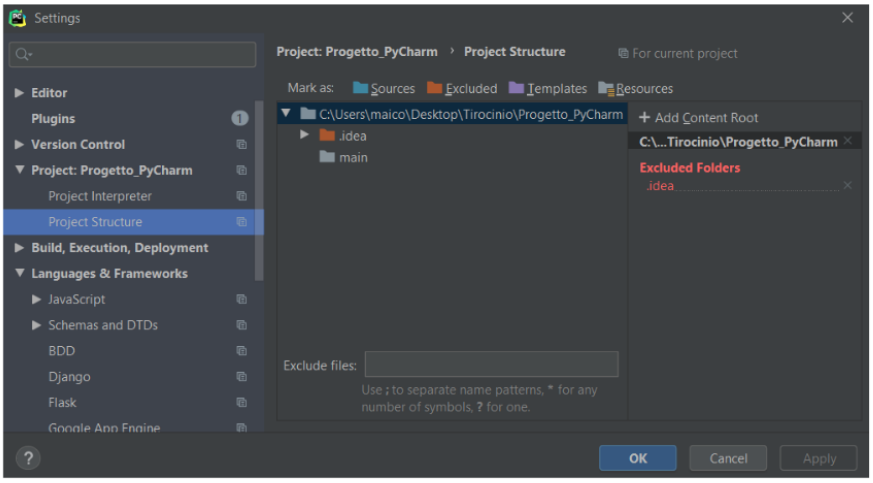


Figura 4.6: Esclusione delle cartelle da non caricare sul chip

Per caricare gli script sulla ESP32, bisogna impostare le configurazioni di esecuzione, come in Figura 4.7, selezionando *Run > Edit Configurations*. Ora cliccare su + per aggiungere una nuova configurazione e scegliere il tipo, in questo caso si seleziona *MicroPython*. Si sceglie un nome per la configurazione e si seleziona il file da caricare, oppure è possibile selezionare un'intera cartella. Si può aprire questa impostazione anche dal menu di scelta rapida in alto a destra nella schermata principale. Infine si preme il tasto *Run* per caricare il progetto. Un altro metodo per caricare codice nel chip è tramite comando dal Prompt. Ci si posiziona nella cartella dove sono inseriti i file da caricare e si esegue il comando *ampy* come mostrato di seguito:

```
$ ampy --port COM3 put nome_file/cartella
```

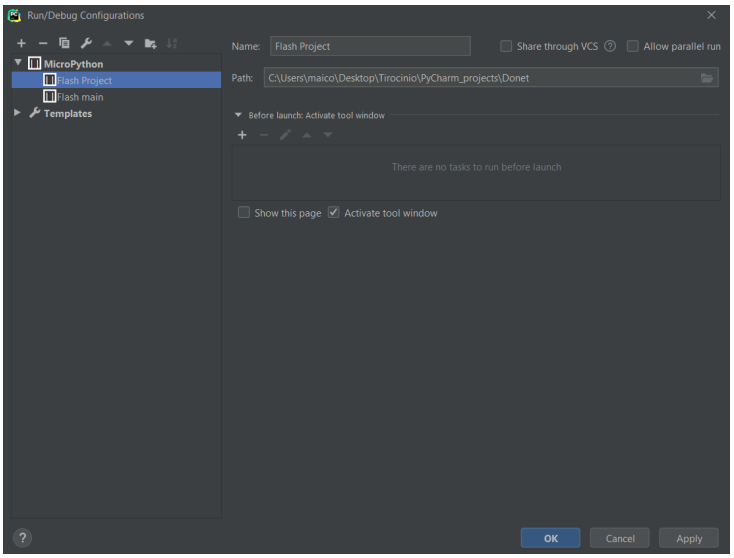


Figura 4.7: Configurazione di esecuzione (Run)

Gli esperimenti interattivi svolgono un ruolo importante nello sviluppo di Python, ma sono ancora più importanti con i microcontrollori che di solito non hanno schermi per mostrare possibili errori. Il plug-in MicroPython consente di eseguire rapidamente una console REPL Python interattiva. Si apre questo terminale da *Tools* > *MicroPython* > *MicroPython REPL*. Quindi si esegue il codice su ESP32 premendo il tasto *EN* della board e attraverso questa shell è possibile visualizzare tutto quello che fa e si leggono gli eventuali errori. In Figura 4.8 è mostrata la schermata del REPL.

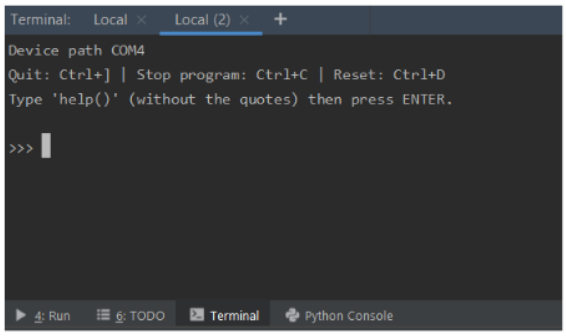


Figura 4.8: MicroPython REPL

Capitolo 5

FIRMWARE DONET

In questo capitolo è spiegato nel dettaglio l'implementazione dell'algoritmo del dispositivo DONET in linguaggio MicroPython. Il codice è stato suddiviso in più librerie:

- **sht21.py**, per il controllo del sensore di temperatura e umidità SHT21
- **veml7700.py**, per il controllo del sensore di luminosità VEML7700
- **ccs811.py**, per il controllo del sensore di qualità dell'aria CCS811
- **wifimgr.py**, per la gestione della connessione WiFi
- **umqttsimple.py**, per la connessione al protocollo di comunicazione MQTT
- **ota_updater.py**, per l'aggiornamento OTA
- **led_donet.py**, per il controllo dei LED del dispositivo
- **donet.py**, per l'algoritmo DONET

5.1 Libreria sht21.py

Il codice per il controllo del sensore di temperatura e umidità relativa [30] è racchiuso all'interno della classe *SHT21*. Si definisce la funzione ***__init__()*** mostrata in Codice 5.1 che riceve in ingresso l'indirizzo *i2c* del master (chip ESP32) e inizializza l'indirizzo *address* dello slave (sensore), il buffer *data* dove memorizzare le letture e il buffer *command* per inserire il comando di lettura.

```
def __init__(self, i2c):
    self._i2c = i2c
    self._address = SHT21_I2C_ADDRESS
    self._data = bytearray(3)
    self._command = bytearray(1)
```

Codice 5.1: Inizializzazione del sensore SHT21

Sono poi implementate le funzioni ***read_temperature()*** e ***read_humidity()*** (Codice 5.2) di lettura della temperatura e dell'umidità relativa, come spiegato in Paragrafo 3.2.2. Per prima cosa viene inviato al sensore tramite la funzione *i2c.writeto()* il comando desiderato tra quelli elencati in Tabella 3.2, si è scelta la modalità *No Hold Command*. Si attende un tempo di 86 ms per la temperatura e di 30 ms per l'umidità come indicato in Tabella 3.3 per permettere al sensore di restituire il dato. Si esegue la lettura tramite *i2c.readfrom()*, ricordando che bisogna leggere tre byte di dati, dei quali i primi due rappresentano la misurazione mentre il terzo è riferito al checksum per controllare gli errori. Se il checksum determinato dalla *_calculate_checksum()* è corretto viene chiamata la funzione per utilizzare il dato misurato e calcolare il valore della temperatura o umidità relativa attraverso le formule indicate sul datasheet.

```
def read_temperature(self):
    """ Reads the temperature from the sensor. Note that this call blocks
    for ~86ms to allow the sensor to return the data. """

    self._command[0] = MEASUREMENT_CMD['eTempNoHoldCmd'] & 0xFF
    self._i2c.writeto(self._address, self._command)
    time.sleep_ms(TEMPERATURE_WAIT_TIME)
    self._data = self._i2c.readfrom(self._address, 3)
    if self._calculate_checksum(self._data, 2) == self._data[2]:
        return self._get_temperature_from_buffer(self._data)

def read_humidity(self):
    """ Reads the humidity from the sensor. Note that this call blocks
    for ~30ms to allow the sensor to return the data. """

    self._command[0] = MEASUREMENT_CMD['eRHumidityNoHoldCmd'] & 0xFF
    self._i2c.writeto(self._address, self._command)
    time.sleep_ms(HUMIDITY_WAIT_TIME)
    self._data = self._i2c.readfrom(self._address, 3)
    if self._calculate_checksum(self._data, 2) == self._data[2]:
        return self._get_humidity_from_buffer(self._data)
```

Codice 5.2: Lettura delle misure del sensore SHT21

Il Codice 5.3 mostra la funzione ***__calculate__checksum()*** per il calcolo del checksum, utilizzando il polinomio fornito dal datasheet

$$P(x) = x^8 + x^5 + x^4 + 1 \tag{5.1}$$

```
def _calculate_checksum(data, number_of_bytes):
    """ 5.7 CRC Checksum using the polynomial given in the datasheet. """

    polynomial = 0x131  # //P(x)=x^8+x^5+x^4+1 = 100110001
    crc = 0
    # calculates 8-Bit checksum with given polynomial
    for byteCtr in range(number_of_bytes):
        crc ^= data[byteCtr]
        for bit in range(8, 0, -1):
            if crc & 0x80:
                crc = (crc << 1) ^ polynomial
            else:
                crc = (crc << 1)
    return crc
```

Codice 5.3: Calcolo del checksum in SHT21

Infine, il Codice 5.4 mostra le funzioni ***__get__temperature__from__buffer()*** e ***__get__humidity__from__buffer()*** che ricevono in ingresso il dato misurato per calcolare rispettivamente la temperatura e l'umidità, applicando l'Equazione 3.1 e l'Equazione 3.2. La funzione della temperatura restituisce due uscite: *temperature* è il valore della temperatura approssimato che DONET invia al broker MQTT, mentre *t_co2* è lo stesso valore senza approssimazioni, che viene fornito al sensore di qualità dell'aria CCS811 per la compensazione delle misurazioni di CO_2 e TVOC. Anche la funzione per l'umidità restituisce due uscite, *rel_humidity* approssimata e *hum_co2* non approssimata.

```
def _get_temperature_from_buffer(data):
    """This function reads the first two bytes of data and
    returns the temperature in C by using the following function:
    T = -46.85 + (175.72 * (ST/2^16))
    where ST is the value from the sensor"""

    t = (data[0] << 8) + data[1]
    t &= STATUS_BITS_MASK  # zero the status bits
    t *= 175.72
    t /= 1 << 16  # divide by 2^16
    t -= 46.85
    delta_t = (t * 11) / 100
    t_co2 = t - delta_t
    temperature = round(t_co2, 1)
    return temperature, t_co2
def _get_humidity_from_buffer(data):
    """This function reads the first two bytes of data and returns
    the relative humidity in percent by using the following function:
    RH = -6 + (125 * (SRH / 2 ^16))
    where SRH is the value read from the sensor"""

    rh = (data[0] << 8) + data[1]
    rh &= STATUS_BITS_MASK  # zero the status bits
    rh *= 125.0
    rh /= 1 << 16  # divide by 2^16
    rh -= 6
    delta_h = (rh * 11) / 100
    hum_co2 = rh + delta_h
```

```
        rel_humidity = round(hum_co2, 1)
        return rel_humidity, hum_co2
```

Codice 5.4: Calcolo della temperatura e dell'umidità relativa

5.2 Libreria veml7700.py

Il codice per il controllo del sensore di luminosità [31] è racchiuso all'interno della classe *VEML7700*. Si definisce la funzione ***___init___()*** mostrata in Codice 5.6 che riceve in ingresso l'indirizzo *i2c* del master (chip ESP32), il valore del tempo di integrazione *it* e il guadagno *gain*, introdotti in Paragrafo 3.2.3. In questa funzione vengono configurati tutti i registri che vengono utilizzati, a seconda dei parametri *it* e *gain* inseriti. Infatti, per selezionare i giusti bit da scrivere sui registri sono state costruite le mappe *CONF_VALUES* e *GAIN_VALUES* mostrate in Codice 5.5 che vengono lette durante l'inizializzazione del sensore.

```
CONF_VALUES = {25: {1/8: bytearray([0x00, 0x13]), 1/4: bytearray([0x00, 0x1B]), 1: bytearray([0x00, 0
    x01]), 2: bytearray([0x00, 0x0B])},
               50: {1/8: bytearray([0x00, 0x12]), 1/4: bytearray([0x00, 0x1A]), 1: bytearray([0x00, 0
                   x02]), 2: bytearray([0x00, 0x0A])},
               100: {1/8: bytearray([0x00, 0x10]), 1/4: bytearray([0x00, 0x18]), 1: bytearray([0x00, 0
                   x00]), 2: bytearray([0x00, 0x08])},
               200: {1/8: bytearray([0x40, 0x10]), 1/4: bytearray([0x40, 0x18]), 1: bytearray([0x40, 0
                   x00]), 2: bytearray([0x40, 0x08])},
               400: {1/8: bytearray([0x80, 0x10]), 1/4: bytearray([0x80, 0x18]), 1: bytearray([0x80, 0
                   x00]), 2: bytearray([0x80, 0x08])},
               800: {1/8: bytearray([0xC0, 0x10]), 1/4: bytearray([0xC0, 0x18]), 1: bytearray([0xC0, 0
                   x00]), 2: bytearray([0xC0, 0x08])}}

GAIN_VALUES = {25: {1/8: 1.8432, 1/4: 0.9216, 1: 0.2304, 2: 0.1152},
               50: {1/8: 0.9216, 1/4: 0.4608, 1: 0.1152, 2: 0.0576},
               100: {1/8: 0.4608, 1/4: 0.2304, 1: 0.0288, 2: 0.0144},
               200: {1/8: 0.2304, 1/4: 0.1152, 1: 0.0288, 2: 0.0144},
               400: {1/8: 0.1152, 1/4: 0.0576, 1: 0.0144, 2: 0.0072},
               800: {1/8: 0.0876, 1/4: 0.0288, 1: 0.0072, 2: 0.0036}}
```

Codice 5.5: Mappe dei valori di configurazione e di guadagno del sensore VEML7700

Viene poi inizializzato l'indirizzo I^2C del sensore e si utilizza la funzione *get()* per prelevare i valori di configurazione dei registri e di guadagno dalle mappe e vengono memorizzati, rispettivamente, in *conf_values* e *gain*. Infine vengono scritti i valori di configurazione nei registri attraverso la funzione *i2c.writeto_mem()*.

```
class VEML7700:

    def __init__(self, i2c, it, gain):

        self.i2c = i2c
        self.address = VEML7700_I2C_ADDRESS
```

```
        self.it = it
        self.gain = gain
        if i2c is None:
            raise ValueError('An I2C object is required.')

        conf_values_for_it = CONF_VALUES.get(it)
        gain_values_for_it = GAIN_VALUES.get(it)
        if conf_values_for_it is not None and gain_values_for_it is not None:
            conf_value_for_gain = conf_values_for_it.get(gain)
            gain_value_for_gain = gain_values_for_it.get(gain)
            if conf_value_for_gain is not None and gain_value_for_gain is not None:
                self.conf_values = conf_value_for_gain
                self.gain = gain_value_for_gain
            else:
                raise ValueError('Wrong gain value. Use 1/8, 1/4, 1, 2')
        else:
            raise ValueError('Wrong integration time value. Use 25, 50, 100, 200, 400, 800')

        # load calibration data
        self.i2c.writeto_mem(self.address, REGISTER['ALS_CONF_0'], self.conf_values)
        self.i2c.writeto_mem(self.address, REGISTER['ALS_WH'], interrupt_high)
        self.i2c.writeto_mem(self.address, REGISTER['ALS_WL'], interrupt_low)
        self.i2c.writeto_mem(self.address, REGISTER['POW_SAV'], power_save_mode)
```

Codice 5.6: Inizializzazione del sensore VEML7700

In Codice 5.7 è mostrata la funzione ***read_lux()*** per la lettura del valore di luminosità. Viene applicato un ritardo di 40 ms. Si ricordi che il dato misurato dal sensore è contenuto in due byte di dati nel registro *ALS*, che vengono letti attraverso *i2c.readfrom_mem_into()*. La lettura viene elaborata, moltiplicata per il guadagno e salvata nella variabile *lux* in uscita dalla funzione.

```
    def read_lux(self):
        """ Reads the data from the sensor and returns the data.
                Returns:
                    the number of lux detect by this captor.
            """
        # The frequency to read the sensor should be set greater than
        # the integration time (and the power saving delay if set).
        # Reading at a faster frequency will not cause an error, but
        # will result in reading the previous data

        self.lux = bytearray(2)
        time.sleep_ms(40)

        self.i2c.readfrom_mem_into(self.address, REGISTER['ALS'], self.lux)
        self.lux = self.lux[0] + self.lux[1] * 256
        self.lux = self.lux * self.gain
        return round(self.lux, 1)
```

Codice 5.7: Lettura della luminosità

5.3 Libreria ccs811.py

Il codice per il controllo del sensore di qualità dell'aria [32] è racchiuso all'interno della classe *CCS811*. Si definisce la funzione ***___init___()*** mostrata in Codice 5.8 che riceve in ingresso l'indirizzo *i2c* del master (chip ESP32). Vengono inizializzate l'indirizzo I^2C *addr* del sensore, la variabile *register* costituita da quattro byte, che contiene il dato letto (due byte per la CO_2 e due per il TVOC), la variabile *status* in cui è salvato lo stato del sensore. La variabile *error_id* serve per verificare gli errori, mentre in *mode* si imposta la modalità di funzionamento del sensore, tra quelle elencate in Paragrafo 3.2.4. Dopo il controllo del corretto collegamento all'I^2C, viene verificato che sul registro *HW_ID* sia scritto il valore *0x81*, che garantisce il corretto funzionamento. Poi si va a leggere il registro *STATUS* per ricavare lo stato del sensore e controllare che il bit 4 *APP_VALID* sia portato alto. Viene cambiata la modalità da quella di boot a quella di applicazione con un operazione di scrittura senza dati (*0xF4*).

```
class CCS811(object):
    """CCS811 gas sensor. Measures eCO2 in ppm and TVOC in ppb"""

    def __init__(self, i2c):
        self.i2c = i2c
        self.addr = CCS811_I2C_ADDRESS
        self.register = bytearray(4)
        self.status = bytearray(1)
        self.error_id = bytearray(1)
        self.mode = 1  # Constant power mode; measurement every second

        # Check if sensor is available at i2c bus address
        devices = i2c.scan()
        if self.addr not in devices:
            raise ValueError('CCS811 not found. Please check wiring. Pull nWake to ground.')

        # See Figure 23 in datasheet: Bootloader Register Map
        # Check HW_ID register - correct value 0x81
        hardware_id = self.i2c.readfrom_mem(self.addr, REGISTER['HW_ID'], 1)
        if hardware_id[0] != 0x81:
            raise ValueError('Wrong Hardware ID.')

        # Check Status Register to see if valid application present-
        self.status = self.i2c.readfrom_mem(self.addr, REGISTER['STATUS'], 1)

        # See Figure 12 in datasheet: Status register: Bit 4: App valid
        if not (self.status[0] >> 4) & 0x01:
            raise ValueError('Application not valid.')

        # Application start. Write with no data to App_Start (0xF4)
        self.i2c.writeto(self.addr, bytearray([0xF4]))

        # Set drive mode 1 - see Figure 13 in datasheet: Measure Mode Register
        self.i2c.writeto_mem(self.addr, REGISTER['MEAS_MODE'], bytearray([0x18]))
```

Codice 5.8: Inizializzazione del sensore CCS811

Il Codice 5.9 mostra la funzione ***put_envdata()*** che riceve in ingresso i valori di umidità

humidity e temperatura *temp* misurati dal sensore SHT21 per calibrare le misure di CO_2 e TVOC. Questi dati sono elaborati e memorizzati nel registro *ENV_DATA*.

```
def put_envdata(self, humidity, temp):
    """Write temeperature and relative humidity values measured by an external
    sensor in Environment Data Register for compensate gas readings due to
    their changes"""

    envregister = bytearray(4)
    envregister[0] = int(humidity) << 1
    t = int(temp//1)
    tf = temp % 1
    t_h = (t+25) << 9
    t_l = int(tf*512)
    t_comb = t_h | t_l
    envregister[2] = t_comb >> 8
    envregister[3] = t_comb & 0xFF
    self.i2c.writeto_mem(self.addr, REGISTER['ENV_DATA'], envregister)
```

Codice 5.9: Calibrazione temperatura e umidità del sensore CCS811

Per la lettura del sensore è stata scritta la funzione ***read_co2_tvoc()*** mostrata in Codice 5.10. Prima di tutto viene letto lo stato per prelevare il bit 0 di errore e verificare che sia basso. Se il bit 3 di *status* è alto, allora è presente un dato da leggere, perciò viene letto il registro *ALG_RESULT_DATA* e memorizzato in *register*. I quattro byte letti sono:

- byte 0 - CO_2 high byte
- byte 1 - CO_2 low byte
- byte 2 - TVOC high byte
- byte 3 - TVOC low byte

I byte sono salvati in variabili separate e poi uniti in due misure costituite da due byte ciascuna, costituendo le uscite della funzione *eco2* e *tvoc* con le letture da restituire a DONET. Viene infine chiamata la funzione *put_envdata()* per aggiornare i dati della temperatura e umidità relativa misurati dal sensore SHT21 per calibrare le letture successive.

```
def read_co2_tvoc(self, humidity, temperature):
    """ Returns true if new data was downloaded. Values in .eCO2 and .tVOC"""

    self.status = self.i2c.readfrom_mem(self.addr, REGISTER['STATUS'], 1)
    # bit 0 in the status register: error
    if self.status[0] & 0x01:
        self.error_id = self.i2c.readfrom_mem(self.addr, REGISTER['ERROR_ID'], 1)
        raise ValueError('Found error number: %d' % self.error_id[0])
    # bit 3 in the status register: data_ready
    if (self.status[0] >> 3) & 0x01:
        # datasheet Figure 14: Algorithm Register Byte Order
        self.register = self.i2c.readfrom_mem(self.addr, REGISTER['ALG_RESULT_DATA'], 4)
```

```
            co2_hb = self.register[0]
            co2_lb = self.register[1]
            tvoc_hb = self.register[2]
            tvoc_lb = self.register[3]
            eco2 = ((co2_hb << 8) | co2_lb)
            tvoc = ((tvoc_hb << 8) | tvoc_lb)
            CCS811.put_envdata(self, humidity, temperature)
            return eco2, tvoc
```

Codice 5.10: Lettura di CO_2 e TVOC

5.4 Libreria wifimgr.py

La libreria *wifimgr.py* si occupa della gestione del WiFi del dispositivo DONET, cioè dell'attivazione del chip in modalità Access Point e Stazione. Inoltre crea la pagina web raggiungibile all'indirizzo per la configurazione dei parametri del dispositivo e un file in cui sono salvate le credenziali della rete WiFi locale alla quale connettersi. Di seguito sono riportate solo le parti principali della libreria, utilizzando come riferimento quella realizzata in [33].

La prima funzione ad essere chiamata è ***connect_ap()*** che attiva il chip in modalità WiFi SoftAP (Codice 5.11), creando un socket d'ascolto per le richieste in arrivo e inviare il testo HTML in risposta, dando quindi la possibilità di accedere tramite l'indirizzo *http://192.168.4.1/* alla pagina web di configurazione dei parametri di DONET da un qualsiasi dispositivo che si connette alla rete generata dal chip ESP32. Oltre a questo, l'esecuzione attende il collegamento alla pagina web in modo da chiamare le funzioni *handle_root()* e *handle_configure()*.

```
def connect_ap(port=80):
    global server_socket, data_input

    addr = socket.getaddrinfo('0.0.0.0', port)[0][-1]

    stop()

    wlan_sta.active(True)
    wlan_ap.active(True)

    wlan_ap.config(essid=ap_ssid, authmode=ap_authmode)

    server_socket = socket.socket()
    server_socket.bind(addr)
    server_socket.listen(5)

    print('Connect to WiFi ssid ' + ap_ssid)
    print('and access the ESP via your favorite web browser at 192.168.4.1.')
    print('Listening on:', addr)
```

Codice 5.11: Attivazione modalità Access Point

La funzione ***handle_root()*** mostrata incompleta in Codice 5.12 si occupa della visualizzazione nella pagina web della schermata di configurazione di Figura 3.12. Viene eseguito uno scan delle reti WiFi nelle vicinanze alle quali è possibile accedere, poi all'interno della funzione *client.sendall()* viene scritto l'HTML della pagina da inviare al socket e mostrare all'utente, inserendo il titolo e i parametri da impostare:

- SSID WiFi
- password WiFi
- indirizzo IP broker MQTT
- numero porta broker MQTT
- user broker MQTT
- password broker MQTT
- intervallo di misura dei parametri ambientali in secondi

```
def handle_root(client):
    wlan_sta.active(True)
    networks = wlan_sta.scan()
    ssids = sorted(ssid.decode('utf-8') for ssid, *_ in networks)
    send_header(client)
    client.sendall("""\
        <html>
        </html>
    """)
    client.close()
```

Codice 5.12: Creazione pagina web di configurazione

Una volta impostati i parametri e inviato il comando, viene richiamata la funzione ***handle_configure()*** mostrata in Codice 5.13, con l'obiettivo di prelevare i dati inseriti e memorizzarli all'interno della variabile *data_input*, costruita come una lista di elementi. Da questa lista vengono prelevati ssid e password della WiFi perché vengono salvati nel file *wifi.dat* che il dispositivo utilizza per recuperarli e connettersi quando riavviato. Gli altri parametri vengono poi passati al chip successivamente. A questo punto si tenta di accedere alla rete (modalità STA) chiamando la funzione ***do_connect()***, mostrata in Codice 5.14: se il risultato è positivo, restituisce *True*, crea una nuova pagina in cui si avvisa l'utente che la connessione è avvenuta correttamente e si salva le credenziali nel *wifi.dat* con le funzioni *read_profiles()* e *write_profiles()*; se il risultato è negativo, avvisa della mancata connessione e restituisce *False*. In Codice 5.13 è mostrato il prelievo dei dati e la risposta di connessione affermativa.

```
def handle_configure(client, request):
    global data_input

    patterns = ['ssid=([^&]*)', 'password=([^&]*)', 'mqttIP=([^&]*)',
                'mqttP=([^&]*)', 'mqttU=([^&]*)', 'mqttPwd=([^&]*)', 'inter=(.*)']
    for pattern in patterns:
        match = re.search(pattern, request)
        try:
            data_input.append(match.group(1).decode("utf-8").replace("=", ""))
        except OSError:
            data_input.append(match.group(1).replace("=", ""))

    ssid = data_input[0]
    password = data_input[1]
    data_input[0:2] = []

    if do_connect(ssid, password):
        response = """\
            <html>
                <center>
                    <br><br>
                    <h1 style="color: #5e9ca0; text-align: center;">
                        <span style="color: #ff0000;">
                            ESP successfully connected to WiFi network %(ssid)s.
                        </span>
                    </h1>
                    <br><br>
                </center>
            </html>
        """ % dict(ssid=ssid)
        send_response(client, response)
        try:
            profiles = read_profiles()
        except OSError:
            profiles = {}
        profiles[ssid] = password
        write_profiles(profiles)

        time.sleep(5)

        return True
```

Codice 5.13: Salvataggio dei parametri di configurazione

```
def do_connect(ssid, password):
    wlan_sta.active(True)
    print('Trying to connect to %s...' % ssid)
    wlan_sta.connect(ssid, password)
    for retry in range(100):
        connected = wlan_sta.isconnected()
        if connected:
            break
        time.sleep(0.1)
        print('.', end='')
    if connected:
        print('\nConnected. Network config: ', wlan_sta.ifconfig())
    else:
        print('\nFailed. Not Connected to: ' + ssid)
    return connected
```

Codice 5.14: Connessione alla rete WiFi locale in modalità STA

Quanto descritto finora è quello che succede alla prima accensione di DONET. Ai successivi riavvii del dispositivo, per potersi connettere alla rete aventi le credenziali salvate, viene chiamata la funzione ***get_connection()*** mostrata in Codice 5.15. Viene prima di tutto letto il *wifi.dat* per recuperare le credenziali e viene fatto uno scan delle WiFi disponibili nelle vicinanze. Se l'ssid salvato corrisponde a uno trovato dallo scan, allora chiama *do_connect()* inserendo ssid e password e si connette.

```
wlan_ap = network.WLAN(network.AP_IF)
wlan_sta = network.WLAN(network.STA_IF)
def get_connection():
    """return a working WLAN(STA_IF) instance or None"""
    connected = False
    try:
        # Read known network profiles from file
        profiles = read_profiles()

        # Search WiFis in range
        wlan_sta.active(True)
        networks = wlan_sta.scan()

        for ssid, bssid, channel, rssi, authmode, hidden in sorted(networks, key=lambda x: x[3],
            reverse=True):
            ssid = ssid.decode('utf-8')
            encrypted = authmode > 0
            print("ssid: %s chan: %d rssi: %d authmode: %s" % (ssid, channel, rssi, AUTHMODE.get(
                authmode, '?')))
            if encrypted:
                if ssid in profiles:
                    password = profiles[ssid]
                    connected = do_connect(ssid, password)
                else:
                    print("skipping unknown encrypted network")
            else:  # open
                connected = do_connect(ssid, None)
            if connected:
                break

    except OSError as e:
        print("exception", str(e))

    return wlan_sta if connected else None
```

Codice 5.15: Connessione alla rete WiFi ai successivi riavvii di DONET

5.5 Libreria umqttsimple.py

Questa libreria che si trova in [34] si occupa della connessione del sistema embedded con il broker attraverso il protocollo di comunicazione MQTT. Le sue funzioni sono definite all'interno della classe *MQTTClient*, a partire da ***__init__()*** di Codice 5.16 che riceve in ingresso il client id, l'indirizzo del server, la porta, l'user e la password.

```
class MQTTClient:

    def __init__(self, client_id, server, port, user, password, keepalive=0,
                 ssl=False, ssl_params={}):
        self.client_id = client_id
        self.sock = None
        self.server = server
        self.port = port
        self.ssl = ssl
        self.ssl_params = ssl_params
        self.pid = 0
        self.cb = None
        self.user = user
        self.pswd = password
        self.keepalive = keepalive
        self.lw_topic = None
        self.lw_msg = None
        self.lw_qos = 0
        self.lw_retain = False
```

Codice 5.16: Inizializzazione MQTT

Di seguito sono mostrate solo le funzioni utilizzate nel progetto, ovvero ***connect()*** in Codice 5.17, ***disconnect()*** in Codice 5.18 e ***publish()*** in Codice 5.19.

```
    def connect(self, clean_session=True):
        self.sock = socket.socket()
        addr = socket.getaddrinfo(self.server, self.port)[0][-1]
        self.sock.connect(addr)
        if self.ssl:
            import ussl
            self.sock = ussl.wrap_socket(self.sock, **self.ssl_params)
        premsg = bytearray(b"\x10\0\0\0\0\0")
        msg = bytearray(b"\x04MQTT\x04\x02\0\0")

        sz = 10 + 2 + len(self.client_id)
        msg[6] = clean_session << 1
        if self.user is not None:
            sz += 2 + len(self.user) + 2 + len(self.pswd)
            msg[6] |= 0xC0
        if self.keepalive:
            assert self.keepalive < 65536
            msg[7] |= self.keepalive >> 8
            msg[8] |= self.keepalive & 0x00FF
        if self.lw_topic:
            sz += 2 + len(self.lw_topic) + 2 + len(self.lw_msg)
            msg[6] |= 0x4 | (self.lw_qos & 0x1) << 3 | (self.lw_qos & 0x2) << 3
            msg[6] |= self.lw_retain << 5

        i = 1
        while sz > 0x7f:
            premsg[i] = (sz & 0x7f) | 0x80
            sz >>= 7
            i += 1
        premsg[i] = sz

        self.sock.write(premsg, i + 2)
        self.sock.write(msg)
        self._send_str(self.client_id)
        if self.lw_topic:
            self._send_str(self.lw_topic)
```

```
            self._send_str(self.lw_msg)
        if self.user is not None:
            self._send_str(self.user)
            self._send_str(self.pswd)
        resp = self.sock.read(4)
        assert resp[0] == 0x20 and resp[1] == 0x02
        if resp[3] != 0:
            raise MQTTException(resp[3])
        return resp[2] & 1
```

Codice 5.17: Connessione MQTT

```
    def disconnect(self):
        self.sock.write(b"\xe0\0")
        self.sock.close()
```

Codice 5.18: Disconnessione MQTT

```
    def publish(self, topic, msg, retain=False, qos=0):
        pkt = bytearray(b"\x30\0\0\0")
        pkt[0] |= qos << 1 | retain
        sz = 2 + len(topic) + len(msg)
        if qos > 0:
            sz += 2
        assert sz < 2097152
        i = 1
        while sz > 0x7f:
            pkt[i] = (sz & 0x7f) | 0x80
            sz >>= 7
            i += 1
        pkt[i] = sz
        #print(hex(len(pkt)), hexlify(pkt, ":"))
        self.sock.write(pkt, i + 1)
        self._send_str(topic)
        if qos > 0:
            self.pid += 1
            pid = self.pid
            struct.pack_into("!H", pkt, 0, pid)
            self.sock.write(pkt, 2)
        self.sock.write(msg)
        if qos == 1:
            while 1:
                op = self.wait_msg()
                if op == 0x40:
                    sz = self.sock.read(1)
                    assert sz == b"\x02"
                    rcv_pid = self.sock.read(2)
                    rcv_pid = rcv_pid[0] << 8 | rcv_pid[1]
                    if pid == rcv_pid:
                        return
        elif qos == 2:
            assert 0
```

Codice 5.19: Pubblicazione MQTT

5.6 Libreria ota_updater.py

La libreria *ota_updater.py* è quella che si occupa dell'aggiornamento OTA del firmware di DONET. All'interno della libreria sono definite tre classi:

- *OTAUpdater*, che confronta la versione del firmware del repository GitHub con quella installata nel chip ed esegue l'aggiornamento OTA
- *HttpClient*, che genera il socket d'ascolto per le richieste http
- *Response*, utilizzata da *HttpClient* per inviare le risposte

Di seguito si descrive solo il codice relativo alla classe *OTAUpdater*, per maggiori informazioni consultare [35]. Come ogni classe, si inizia con la ***__init__()*** mostrata in Codice 5.20, che riceve in ingresso il link del repository GitHub *github_repo* e attraverso il campo *headers* viene passato il token del repository per l'autorizzazione all'accesso. Viene poi inizializzata la classe *HttpClient*, mentre *github_repo* viene modificato sostituendo la stringa *https://github.com* con *https://api.github.com/repos* per utilizzare l'API GitHub. La variabile *main_dir* indica invece il nome della cartella (che è stata chiamata *main*) da cercare nel repository in cui si trova il nuovo firmware da scaricare.

```
class OTAUpdater:

    def __init__(self, github_repo, module='', main_dir='main', headers={}):
        self.http_client = HttpClient(headers=headers)
        self.github_repo = github_repo.rstrip('/').replace('https://github.com', 'https://api.github.
            com/repos')
        self.main_dir = main_dir
        self.module = module.rstrip('/')
```

Codice 5.20: Inizializzazione OTAUpdater

In Codice 5.21 sono mostrate le funzioni ***get_version()*** e ***get_latest_version()*** per determinare il numero, rispettivamente, della versione attuale presente in DONET e dell'ultima versione rilasciata nel repository.

```
    def get_version(self, directory, version_file_name='.version'):
        if version_file_name in os.listdir(directory):
            f = open(directory + '/' + version_file_name)
            version = f.read()
            f.close()
            return version
        return '1.0'

    def get_latest_version(self):
        latest_release = self.http_client.get(self.github_repo + '/releases/latest')
        version = latest_release.json()['tag_name']
```

```
        latest_release.close()
        return version
```

Codice 5.21: Controllo versioni firmware

In Codice 5.22 sono mostrate le funzioni ***download_all_files()*** e ***download_file()*** utilizzate per il download dei file dal repository.

```
    def download_all_files(self, root_url, version):
        file_list = self.http_client.get(root_url + '?ref=refs/tags/' + version)
        for file in file_list.json():
            if file['type'] == 'file':
                download_url = file['download_url']
                download_path = self.modulepath('next/' + file['path'].replace(self.main_dir + '/', '')
                    )
                self.download_file(download_url.replace('refs/tags/', ''), download_path)
            elif file['type'] == 'dir':
                path = self.modulepath('next/' + file['path'].replace(self.main_dir + '/', ''))
                os.mkdir(path)
                self.download_all_files(root_url + '/' + file['name'], version)

        file_list.close()

    def download_file(self, url, path):
        print('\tDownloading: ', path)
        with open(path, 'w') as outfile:
            try:
                response = self.http_client.get(url)
                outfile.write(response.text)
            finally:
                response.close()
                outfile.close()
                gc.collect()
```

Codice 5.22: Download dei file

Quando si vuole controllare la presenza di un aggiornamento OTA, si chiama la funzione ***check_for_update_to_install_during_next_reboot()*** mostrata in Codice 5.23. Chiama le funzioni *get_version()* e *get_latest_version()* per determinare, rispettivamente, la versione attuale *current_version* in DONET e l'ultima *latest_version* presente su GitHub. Se *latest_version* è maggiore di *current_version* significa che è presente un nuovo aggiornamento da scaricare e crea una nuova cartella chiamata *next* nella memoria SRAM del chip ESP32, aggiornando il numero della versione *latest_version*.

```
    def check_for_update_to_install_during_next_reboot(self):
        current_version = self.get_version(self.modulepath(self.main_dir))
        latest_version = self.get_latest_version()

        print('Checking version... ')
        print('\tCurrent version: ', current_version)
        print('\tLatest version: ', latest_version)
        if latest_version > current_version:
            print('New version available, will download and install on next reboot')
```

```
            try:
                os.mkdir(self.modulepath('next'))
            except OSError:
                pass
            with open(self.modulepath('next/.version_on_reboot'), 'w') as versionfile:
                versionfile.write(latest_version)
                versionfile.close()
```

Codice 5.23: Verifica dell'aggiornamento

Ora è possibile avviare il download dell'aggiornamento, chiamando la funzione ***download_and_install_update_if_available()***, che controlla la presenza della cartella *next* e chiama la funzione ***__download_and_install_update()*** per scaricare l'aggiornamento. Queste due funzioni sono riportate in Codice 5.24.

```
    def download_and_install_update_if_available(self):
        if 'next' in os.listdir(self.module):
            if '.version_on_reboot' in os.listdir(self.modulepath('next')):
                latest_version = self.get_version(self.modulepath('next'), '.version_on_reboot')
                print('New update found: ', latest_version)
                self._download_and_install_update(latest_version)
        else:
            print('No new updates found...')
            machine.reset()

    def _download_and_install_update(self, latest_version):
        self.download_all_files(self.github_repo + '/contents/' + self.main_dir, latest_version)
        self.rmtree(self.modulepath(self.main_dir))
        os.rename(self.modulepath('next/.version_on_reboot'), self.modulepath('next/.version'))
        os.rename(self.modulepath('next'), self.modulepath(self.main_dir))
        print('Update installed (', latest_version, '), will reboot now')
        machine.reset()
```

Codice 5.24: Avvio del download

5.7 Libreria led_donet.py

In questa libreria sono raccolte tutte le funzioni per controllare l'accensione dei LED di DONET. All'interno della classe *PCF8575* è stato messo il codice per la gestione dell'I/O expander ed è mostrato in Codice 5.25. C'è la ***__init__()*** che riceve l'indirizzo dello slave al quale accedere. Vengono poi definiti l'indirizzo *i2c* del master (ESP32) e dichiarata la variabile *port* che contiene i due byte di dati da leggere o scrivere. In questo progetto gli expander sono usati per scrivere dati ai LED in modo da controllare la loro accensione, quindi in Codice 5.25 sono mostrate solo le funzioni che permettono la scrittura.

```
class PCF8575:
    def __init__(self, address):
        self._i2c = I2C(scl=Pin(I2C_SCL_GPIO), sda=Pin(I2C_SDA_GPIO))
```

```
        self._address = address
        self._port = bytearray(2)
    def port(self, value):
        self._port[0] = value & 0xff
        self._port[1] = (value >> 8) & 0xff
        self._write()
    def _write(self):
        self._i2c.writeto(self._address, self._port)
```

Codice 5.25: Gestione dell'I/O expander PCF8575

Il codice è stato scritto in modo da poter controllare i seguenti parametri dei LED:

- PATTERN, ovvero se l'accensione del LED deve essere fissa, lampeggiante o rotante
- SPEED, per selezionare la velocità di accensione durante la rotazione dei LED
- COLOR, cioè il colore di accensione, in cui oltre a rosso, verde e blu è possibile sommare i bit dei colori tra loro e ottenerne altri come nero, bianco, giallo, magenta e ciano

Il Codice 5.26 mostra la funzione ***io_exp_write()*** che riceve in ingresso l'indirizzo *adr8* dello slave a 8 bit in cui scrivere e il dato *data16* a 16 bit. Viene chiamata e inizializzata la classe *PCF8575* ed eseguita la scrittura.

```
def io_exp_write(adr8, data16):
    # code to program PCF8575 I/O expanders

    data16 = ~ data16
    pcf = PCF8575(adr8)
    pcf.port = data16
    return True
```

Codice 5.26: Scrittura dati negli expander PCF8575

Per dire all'ESP32 quale colore dei LED accendere è stata scritta la funzione ***write_leds()*** che riceve in ingresso la bit mask, cioè il dato da 16 bit in cui ogni bit corrisponde a un LED, la presenza dello *0* indica che deve essere spento, mentre l'*1* indica che deve accendersi. In Codice 5.27 è mostrata il codice per accendere il LED di rosso, quindi gli altri colori sono uguali. La variabile *g_led* indica il colore desiderato, mentre *g_led_bitmask* contiene le bitmask dell'ultima configurazione effettuata prima della nuova chiamata di *write_leds()*. Se *g_led* corrisponde a rosso, allora si salva la bitmask in una variabile di appoggio, altrimenti questa variabile è settata a zero perché significa che quel colore deve restare spento. Dopodiché se la bitmask è diversa rispetto a quella precedente, si chiama la funzione *io_exp_write()* per scrivere il dato, che viene poi memorizzato in *g_led_bitmask*. Se la bitmask è uguale a quella precedente non c'è bisogno di scrivere lo stesso dato.

```
def write_leds(bit_mask):
    global g_led, g_led_bitmask

    if g_led['color'] & LED_COLOR['RED']:
        _bit_mask = bit_mask
    else:
        _bit_mask = LED_BITMASK_OFF
    if _bit_mask != g_led_bitmask['RED']:
        io_exp_write(I2C_IOEXP['RED'], _bit_mask)
        g_led_bitmask['RED'] = _bit_mask
```

Codice 5.27: Selezione del colore dei LED da accendere

La funzione ***setup_leds()*** mostrata in Codice 5.28 riceve in ingresso il colore *led_color* e lo schema *_led_scheme* da impostare ai LED. La variabile *g_led_scheme* indica le ultime impostazioni dello schema selezionate prima della chiamata di questa funzione. Se questi parametri sono gli stessi di quelli precedenti allora non viene fatto nulla e si esce dalla funzione, altrimenti, aggiorna i parametri a seconda dello schema desiderato. Se si sceglie un accensione fissa dei LED, allora si richiama la funzione *write_leds()* con *LED_BITMASK_ON*, ovvero imposta a tutti *0x0FFF* la bitmask da scrivere.

```
def setup_leds(led_color, _led_scheme):
    global g_led, g_led_scheme

    if led_color == g_led['color'] and _led_scheme == g_led_scheme:
        print("** skip identical LED setup")
        return

    if _led_scheme['speed'] == LED_SPEED['SLOW']:
        g_led['period'] = 48
    elif _led_scheme['speed'] == LED_SPEED['MEDIUM']:
        g_led['period'] = 24
    elif _led_scheme['speed'] == LED_SPEED['FAST']:
        g_led['period'] = 12

    g_led['pattern_div'] = g_led['period'] // 12
    g_led['phase'] = 0
    g_led['loops'] = 0

    if _led_scheme['pattern'] != LED_PATTERN['FIXED']:
        g_led['prev_color'] = g_led['color']
    g_led['color'] = led_color
    g_led_scheme = _led_scheme

    if _led_scheme['pattern'] == LED_PATTERN['FIXED']:
        write_leds(LED_BITMASK_ON)
```

Codice 5.28: Setup dello schema dei LED

Se invece lo schema da impostare è il lampeggio o rotazione dei LED, la funzione ***advance_leds()*** in Codice 5.29 imposta le bitmask a seconda della richiesta. Il parametro *loops* dello schema indica il numero di giri con il quale si devono accendere i LED.

```
def advance_leds():
    global g_led, g_led_scheme

    if g_led_scheme['pattern'] == LED_PATTERN['FIXED']:
        return

    if g_led_scheme['pattern'] == LED_PATTERN['BLINK']:
        if g_led['phase'] < (g_led['period'] / 2):
            write_leds(LED_BITMASK_ON)
        else:
            write_leds(LED_BITMASK_OFF)
    elif g_led_scheme['pattern'] == LED_PATTERN['ROTATE_CW']:
        write_leds(g_ledBitmaskCW[g_led['phase'] // g_led['pattern_div']])
    elif g_led_scheme['pattern'] == LED_PATTERN['ROTATE_CCW']:
        write_leds(g_ledBitmaskCCW[g_led['phase'] // g_led['pattern_div']])

    g_led['phase'] += 1
    if g_led['phase'] == g_led['period']:
        if g_led_scheme['loops'] != 0:
            g_led['loops'] += 1
            if g_led['loops'] == g_led_scheme['loops']:
                led_scheme = LED_SCHEME
                led_scheme['pattern'] = LED_PATTERN['FIXED']
                led_scheme['speed'] = LED_SPEED['ANY']
                led_scheme['loops'] = 0
                setup_leds(g_led['prev_color'], led_scheme)
        g_led['phase'] = 0
```

Codice 5.29: Setup avanzato dei LED

5.8 Libreria donet.py

Questa libreria è la principale del programma, perché è quella che contiene le funzioni per la gestione di tutte le attività di DONET. Le funzioni sono contenute all'interno della classe chiamata *DONET*, quindi dopo la chiamata di questa libreria viene eseguita l'inizializzazione del dispositivo con la funzione ***___init___()***. In Codice 5.30 viene eseguita l'inizializzazione delle componenti hardware di DONET nelle seguenti variabili:

- *USR_BUTTON* è il pin impostato come ingresso al quale è collegato il pulsante BOOT
- *nWAKE* è il pin impostato come uscita che corrisponde al segnale di wake del sensore CCS811 da portare al livello basso prima di eseguire una transazione I^2C
- *i2c* è la variabile in cui viene inizializzata la comunicazione I^2C, selezionando il pin per i dati (SDA) e quello per il clock (SCL)
- *sht21* è quella in cui viene inizializzato il sensore SHT21
- *veml7700* è la variabile in cui viene inizializzato il sensore VEML7700

- *ccs811* serve per l'inizializzazione del sensore CCS811
- *led_scheme* è la variabile in cui viene salvato la struttura dello schema da applicare ai LED
- *g_usrPrevBtnState* è un bit che indica lo stato del pulsante BOOT nell'istante precedente, cioè se è stato premuto oppure no
- *g_usrBtnPressMillis* è la variabile in cui viene salvato l'istante di tempo in cui viene premuto il tasto BOOT
- *g_measure_counter* è il contatore utilizzato per sapere quando è il momento di eseguire le misure
- *g_millisSinceConnected* tiene conto da quanto tempo DONET è collegato
- *ota* in cui viene inizializzata la classe *OTAUpdater*
- *config_done* è una variabile booleana che avverte se la configurazione del dispositivo è stata fatta oppure no (inizialmente impostato a *False*)

```
class DONET:

    def __init__(self):

        # Hardware setup
        self.USR_BUTTON = Pin(USR_BUTTON_GPIO, Pin.IN)

        self.nWAKE = Pin(CCS811_nWAKE, Pin.OUT)
        time.sleep_us(100)
        self.nWAKE.value(1)

        self.i2c = I2C(scl=Pin(I2C_SCL_GPIO), sda=Pin(I2C_SDA_GPIO))
        self.sht21 = sht21.SHT21(i2c=self.i2c)
        self.veml7700 = veml7700.VEML7700(i2c=self.i2c, it=100, gain=2)
        self.nWAKE.value(0)
        time.sleep_us(100)
        self.ccs811 = ccs811.CCS811(i2c=self.i2c)
        time.sleep_us(100)
        self.nWAKE.value(1)

        self.led_scheme = LED_SCHEME

        self.g_usrPrevBtnState = USR_BTN_NOT_PRESSED
        self.g_usrBtnPressMillis = 0
        self.g_measure_counter = 0

        self.g_millisSinceConnected = 0

        self.ota = ota_updater.OTAUpdater('https://github.com/maicolleti/OTAPrivate',
                                          headers={'Authorization': 'token {}'.format(token)})

        self.config_done = False
```

Codice 5.30: Hardware setup DONET

La funzione ***config_mode()*** mostrata in Codice 5.31 è quella che gestisce la configurazione iniziale di DONET. Quando è chiamata, accende i LED di blu fisso attraverso la funzione *setup_leds()* e attiva l'ESP32 in modalità WiFi SoftAP con la funzione *connect_ap()*, quindi si avvia la procedura di configurazione descritta in Paragrafo 3.4.2 e al termine usa le credenziali WiFi per connettersi alla rete con *get_connection()*. Dopodiché bisogna creare un file in formato JSON per salvare i parametri inseriti dall'utente durante la configurazione in modo da poterli recuperare ogni volta che DONET viene riavviato. Per prima cosa si ricevono i parametri attraverso la funzione ***receive_input_parameters()***, definita nella libreria *wifimgr.py*, che semplicemente passa i dati e vengono salvati nella variabile *input_parameters*. A questo punto viene definito il dizionario *_py_obj_dict* in cui verranno salvati i dati in un formato etichetta-valore, utile per la conversione in JSON. Sono quindi inizializzati le seguenti variabili:

- *g_mqttServerIP* che contiene il server IP del broker MQTT
- *g_mqttServerPort* per la porta del broker MQTT
- *g_mqttUser* per il nome utente del broker MQTT
- *g_mqttPassword* per la password del broker MQTT
- *g_measureIntervalSec* per l'intervallo in cui devono essere fatte le misure

Per ognuno di questi parametri, si controlla se è stato inserito dall'utente oppure no: se è inserito, allora viene selezionato quel valore, altrimenti viene preso il valore di default impostato nel codice. Ora i dati possono essere convertiti in JSON attraverso la funzione *dumps()* della libreria *json* e viene creato il file *config_json.txt* dove memorizzarli. Infine, si possono spegnere i LED e resettare la scheda.

```
def config_mode(self):

    print('**** board entering config mode')
    self.led_scheme['pattern'] = LED_PATTERN['FIXED']
    led_donet.setup_leds(LED_COLOR['BLUE'], self.led_scheme)
    wifimgr.connect_ap()
    self.wlan_sta = wifimgr.get_connection()
    if self.wlan_sta is None:
        print("Could not initialize the network connection.")
        time.sleep(1)
        reset()
    input_parameters = wifimgr.receive_input_parameters()

    _py_obj_dict = {}

    if input_parameters[0] == "":
        self.g_mqttServerIP = k_mqttServerDefault
    else:
        self.g_mqttServerIP = input_parameters[0]
```

```
        _py_obj_dict['mqttServerIP'] = self.g_mqttServerIP

        if input_parameters[1] == "":
            self.g_mqttServerPort = k_mqttPortDefault
        else:
            self.g_mqttServerPort = int(input_parameters[1])
        _py_obj_dict['mqttServerPort'] = self.g_mqttServerPort

        if input_parameters[2] == "":
            self.g_mqttUser = k_mqttUserDefault
        else:
            self.g_mqttUser = input_parameters[2]
        _py_obj_dict['mqttUser'] = self.g_mqttUser

        if input_parameters[3] == "":
            self.g_mqttPassword = k_mqttPasswordDefault
        else:
            self.g_mqttPassword = input_parameters[3]
        _py_obj_dict['mqttPassword'] = self.g_mqttPassword

        if input_parameters[4] == "":
            self.g_measureIntervalSec = k_measureIntervalDefault
        else:
            self.g_measureIntervalSec = int(input_parameters[4])
        _py_obj_dict['measureIntervalSec'] = self.g_measureIntervalSec

        _json_file = json.dumps(_py_obj_dict)

        _f = open('config_json.txt', 'w')
        _f.write(_json_file)
        print('** custom parameters saved OK')
        _f.close()

        led_donet.setup_leds(LED_COLOR['BLACK'], self.led_scheme)

        print('** RESTART IN ONE SECOND...')
        time.sleep(1)
        reset()
```

Codice 5.31: Modalità di configurazione DONET

All'interno della *___init___()* è presente un'altra sezione di codice (Codice 5.32), racchiusa all'interno delle funzione *try-except*. Questa parte è quella che permette il recupero dei dati dopo il riavvio della scheda. Il motivo per il quale è racchiuso da queste funzioni, è perché questo codice può essere eseguito solamente dopo la configurazione di DONET, quindi durante la prima accensione del codice il *try-except* permette di evitare un *AttributeError*. Quindi viene letto il file *config_json.txt* percorrendo la procedura opposta a quella di creazione dello stesso file: si legge il file con la funzione *read()* e si converte *json_file* che contiene i dati nel formato JSON, in un dizionario *py_obj_dict* attraverso la funzione *loads()* della libreria *json*. Vengono memorizzati i dati e per l'intervallo di misura si verifica se il valore inserito dall'utente si trova all'interno dei limiti impostati nel codice, con un minimo di 30 secondi e un massimo di 1 ora. Si imposta *config_done* a *True* e la fase di inizializzazione del dispositivo è terminata.

```
        try:
            self.wlan_sta = wifimgr.get_connection()
            self.g_macAddress = binascii.hexlify(self.wlan_sta.config('mac'), ':').decode()
            self.g_mqttClientID = 'WSN_' + self.g_macAddress[9:11] + self.g_macAddress[12:14] + self.
                g_macAddress[15:17]
            self.g_wifiStatus = self.wlan_sta.status()

            try:
                f = open('config_json.txt')
                print(f.readlines())
            except f is None | FileNotFoundError:
                print('File config_json.txt not found or failed opening, do reset')
                time.sleep(1)
                reset()
            finally:
                f.close()

            # fetch MQTT setup from JSON config file
            with open('config_json.txt') as f:
                json_file = f.read()
                py_obj_dict = json.loads(json_file)

            self.g_mqttServerIP = py_obj_dict['mqttServerIP']
            print('**** %s: %s' % ('mqttServerIP', self.g_mqttServerIP))
            self.g_mqttServerPort = py_obj_dict['mqttServerPort']
            print('**** %s: %s' % ('mqttServerPort', self.g_mqttServerPort))
            self.g_mqttUser = py_obj_dict['mqttUser']
            print('**** %s: %s' % ('mqttUser', self.g_mqttUser))
            self.g_mqttPassword = py_obj_dict['mqttPassword']
            print('**** %s: %s' % ('mqttPassword', self.g_mqttPassword))

            self.g_mqttClient = umqttsimple.MQTTClient(self.g_mqttClientID, self.g_mqttServerIP, self.
                g_mqttServerPort,
                                                       self.g_mqttUser, self.g_mqttPassword)

            self.g_measureIntervalSec = py_obj_dict['measureIntervalSec']
            if self.g_measureIntervalSec < k_measureIntervalMin:
                self.g_measureIntervalSec = k_measureIntervalMin
            elif self.g_measureIntervalSec > k_measureIntervalMax:
                self.g_measureIntervalSec = k_measureIntervalMax
            self.g_measure_period = 1000 * self.g_measureIntervalSec
            print('**** %s: %u' % ('measureIntervalSec', self.g_measureIntervalSec))

            self.config_done = True
        except AttributeError:
            pass
```

Codice 5.32: Inizializzazione parametri DONET

In Codice 5.33 è mostrata la funzione ***download_and_install_new_update()***, chiamata per controllare nel repository GitHub la presenza di un aggiornamento del firmware via OTA.

```
    def download_and_install_new_update(self):
        self.ota.check_for_update_to_install_during_next_reboot()
        self.ota.download_and_install_update_if_available()
        led_donet.setup_leds(LED_COLOR['BLUE'], self.led_scheme)
        self.end_of_loop(LED_SCHEME_MILLIS)
```

Codice 5.33: Controllo aggiornamento OTA

La funzione ***handle_usr_button()*** in Codice 5.34 si occupa delle funzioni associate al tasto BOOT del dispositivo. La pressione del pulsante permette la chiamata della funzione *download_and_install_new_update()* per il controllo OTA quando premuto per un tempo da 4 a 8 secondi, mentre per un tempo superiore agli 8 secondi viene chiamata la funzione *config_mode()* per avviare la procedura di configurazione di DONET.

```
def handle_usr_button(self):

    while True:
        if self.config_done is not True:
            current_btn_state = self.USR_BUTTON.value()
            if current_btn_state != self.g_usrPrevBtnState:
                time.sleep_ms(10)
                current_btn_state = self.USR_BUTTON.value()
                if current_btn_state == USR_BTN_PRESSED:
                    print("** USR BUTTON PRESSED")
                    self.g_usrBtnPressMillis = time.time() * 1000
                if current_btn_state == USR_BTN_NOT_PRESSED:
                    millis_now = time.time() * 1000
                    if k_longUsrPressMillis <= (millis_now - self.g_usrBtnPressMillis) <
                        k_veryLongUsrPressMillis:
                        print('** LONG USR PRESS - call ota updater')
                        if not self.wlan_sta.isconnected():
                            self.wlan_sta = wifimgr.get_connection()
                        self.download_and_install_new_update()
                        self.led_scheme['pattern'] = LED_PATTERN['FIXED']
                        time.sleep(1)
                        reset()
                        led_donet.setup_leds(LED_COLOR['BLACK'], self.led_scheme)
                    elif (millis_now - self.g_usrBtnPressMillis) >= k_veryLongUsrPressMillis:
                        print('** VERY LONG USR PRESS - enter config mode')
                        if self.wlan_sta is None:
                            pass
                        else:
                            if self.wlan_sta.isconnected():
                                self.wlan_sta.disconnect()
                        self.config_mode()
                    else:
                        self.config_done = True
                        return
                self.g_usrPrevBtnState = current_btn_state
        else:
            current_btn_state = self.USR_BUTTON.value()
            if current_btn_state == USR_BTN_PRESSED:
                print("** USR BUTTON PRESSED")
                self.config_done = False
            return
```

Codice 5.34: Controllo del tasto BOOT

In Codice 5.35 è riportata la funzione ***publish_data()*** che si occupa della pubblicazione delle letture dei sensori al broker MQTT attraverso messaggi in formato JSON. Riceve in ingresso la lista *json_list* delle letture. Nella variabile *values* viene costruito il messaggio da pubblicare utilizzando la funzione *OrderedDict()* appartenente alla libreria *collections* di MicroPython, per ordinare i dati nell'ordine con la quale vengono inseriti nel codice, quindi come si desidera

visualizzare nel broker. Nell'ultima versione del firmware, il messaggio è costituito dal MAC address di DONET, seguito dalle misure di temperatura, umidità relativa, luminosità, CO_2 e TVOC. Ogni campo è costituito dal nome della grandezza seguito dal valore misurato. A questo punto si avvia la connessione al broker MQTT con la funzione *connect()* della libreria *umqttsimple.py* e viene pubblicato il messaggio con la funzione *publish()*, fornendogli in ingresso il topic *measure_mpy* del broker MQTT nel quale pubblicare il messaggio, e il messaggio *values* convertito in JSON. Infine si disconnette DONET dal broker. Il codice della pubblicazione del messaggio è racchiuso in un *try-except* in modo da poter riprovare la pubblicazione in caso di errore e mostrandolo all'utente lampeggiando i LED di blu e rosso alternati.

```
def publish_data(self, json_list):
    values = OrderedDict([('mac.address', '{}'.format(json_list[0])),
                          ('temperature.celsius', '{}'.format(str(json_list[1]))),
                          ('relative.humidity', '{}'.format(str(json_list[2]))),
                          ('light.lux', '{}'.format(str(json_list[3]))),
                          ('co2.ppm', '{}'.format(str(json_list[4]))),
                          ('tvoc.ppb', '{}'.format(str(json_list[5])))])
    try:
        self.g_mqttClient.connect()
        self.g_mqttClient.publish(k_measureTopic, json.dumps(values))
        self.g_mqttClient.disconnect()
        print('MQTT[%s]: %s ' % (k_measureTopic, json.dumps(values)))
        return True
    except OSError:
        self.led_scheme['pattern'] = LED_PATTERN['FIXED']
        self.led_scheme['speed'] = LED_SPEED['ANY']
        self.led_scheme['loops'] = 0
        led_donet.setup_leds(LED_COLOR['BLUE'], self.led_scheme)
        time.sleep_ms(300)
        led_donet.setup_leds(LED_COLOR['RED'], self.led_scheme)
        time.sleep_ms(300)
        led_donet.setup_leds(LED_COLOR['BLUE'], self.led_scheme)
        time.sleep_ms(300)
        led_donet.setup_leds(LED_COLOR['RED'], self.led_scheme)
        time.sleep_ms(300)
        self.publish_data(json_list)
```

Codice 5.35: Pubblicazione dati al broker MQTT

La ***end_of_loop()*** in Codice 5.36, come intuibile dal nome, è la funzione che viene chiamata alla fine del loop principale, che attribuisce a DONET un ritardo di *delay_millis* millisecondi e chiama la funzione *advance_leds()* per impostare lo schema da applicare ai LED.

```
def end_of_loop(_delay_millis):
    time.sleep_ms(_delay_millis)
    led_donet.advance_leds()
```

Codice 5.36: Fine del loop principale

L'ultima funzione della libreria è la ***main_loop()*** che viene chiamata in un ciclo infinito dal *main.py* caricato nel chip ed esegue tutte le attività di DONET chiamando a sua volta le funzioni fin qui descritte. Come si può vedere in Codice 5.37 si definisce il *delay_millis*, cioè il tempo in millisecondi (scelto a 50 ms) da aggiungere ad ogni ciclo al contatore *g_measure_counter*, così il chip saprà il momento della lettura dei dati. Viene chiamata la funzione *handle_usr_button()* in modo da riconoscere gli eventi da attivare quando il tasto è premuto. Il codice è diviso da un *if-else* che controlla se DONET è connesso al WiFi oppure no. Quando non è connesso, vengono accesi i LED di rosso fisso e tenta la riconnessione terminando con la *end_of_loop()*. Se invece è connesso, verifica prima di tutto se è il momento di misurare le grandezze ambientali, quindi chiama le funzioni *read_temperature()*, *read_humidity()*, *read_lux()*, *read_co2_tvoc()* per ricevere le misure. Si crea la lista *data* costituita dalle variabili *g_macAddress*, *temp*, *hum*, *lux*, *co2*, *tvoc* e lo pone in ingresso alla *publish_data()* per pubblicare le misure attraverso il protocollo MQTT. Completata la pubblicazione si azzera *g_measure_counter* e si accendono i LED di verde rotante, ma solo per valori di luminosità superiori a 1 lux per evitare l'emissione di luce durante le ore notturne. L'ultima parte del codice è dedicata al calcolo del tempo di connessione di DONET. Si incrementa *g_millisSinceConnected* di *delay_millis*. Il tempo di connessione è calcolato solo se *g_millisSinceConnected* è un tempo uguale o multiplo di 10 secondi. Dal tempo in millisecondi si ricavano i secondi, i minuti, le ore e i giorni, quindi viene creato il messaggio *dbg_string* e pubblicato nel topic *debug_mpy* del broker MQTT. Per finire, si aggiorna il contatore del tempo di misura *g_measure_counter* e si procede con *end_of_loop()*.

```
def main_loop(self):

    delay_millis = LED_SCHEME_MILLIS
    self.handle_usr_button()

    if self.wlan_sta.isconnected() is not True:
        led_donet.setup_leds(LED_COLOR['RED'], self.led_scheme)
        ssid = self.wlan_sta.config('essid')
        print('** try to connect to saved SSID: %s' % ssid)
        self.wlan_sta = wifimgr.get_connection()
        self.g_wifiStatus = self.wlan_sta.status()
        self.led_scheme['pattern'] = LED_PATTERN['FIXED']
        if self.wlan_sta.isconnected() is not True:
            delay_millis = 1000
            led_donet.setup_leds(LED_COLOR['BLACK'], self.led_scheme)
            self.end_of_loop(delay_millis)
        led_donet.setup_leds(LED_COLOR['BLACK'], self.led_scheme)
        self.end_of_loop(delay_millis)
    else:
        if self.g_measure_counter == self.g_measure_period:
            # do measures and publish
            temp, temp_co2 = self.sht21.read_temperature()
            hum, hum_co2 = self.sht21.read_humidity()
            lux = self.veml7700.read_lux()
            self.nWAKE.value(0)
            time.sleep_us(100)
            co2, tvoc = self.ccs811.read_co2_tvoc(hum_co2, temp_co2)
```

```
            time.sleep_us(100)
            self.nWAKE.value(1)
            data = [self.g_macAddress, temp, hum, lux, co2, tvoc]
            ok = self.publish_data(data)
            if ok is not True:
                print('publish error...')
            if lux > 1.0:
                self.led_scheme['pattern'] = LED_PATTERN['ROTATE_CW']
                self.led_scheme['speed'] = LED_SPEED['FAST']
                self.led_scheme['loops'] = 3
                led_donet.setup_leds(LED_COLOR['GREEN'], self.led_scheme)
            self.g_measure_counter = 0

        self.g_millisSinceConnected += delay_millis
        if not self.g_millisSinceConnected % 10000:
            secs_since_connected = self.g_millisSinceConnected // 1000
            mins_since_connected = secs_since_connected // 60
            secs_since_connected -= mins_since_connected * 60
            hours_since_connected = mins_since_connected // 60
            mins_since_connected -= hours_since_connected * 60
            days_since_connected = hours_since_connected // 24
            hours_since_connected -= days_since_connected * 24
            if days_since_connected:
                dbg_string = 'connected for {}d {}h {}m {}s'.format(days_since_connected,
                                                                    hours_since_connected,
                                                                    mins_since_connected,
                                                                    secs_since_connected)
            elif hours_since_connected:
                dbg_string = 'connected for {}h {}m {}s'.format(hours_since_connected,
                                                                mins_since_connected,
                                                                secs_since_connected)
            else:
                dbg_string = 'connected for {}m {}s'.format(mins_since_connected,
                                                            secs_since_connected)
            pub_string = '{}: {}'.format(self.g_mqttClientID, dbg_string)
            self.g_mqttClient.connect()
            self.g_mqttClient.publish(k_debugTopic, pub_string)
            self.g_mqttClient.disconnect()
            print('MQTT[%s]: %s ' % (k_debugTopic, pub_string))

        self.g_measure_counter += delay_millis

        self.end_of_loop(delay_millis)
```

Codice 5.37: Loop principale DONET

Capitolo 6

RISULTATI SPERIMENTALI

Il firmware descritto in Capitolo 5 è stato inizialmente implementato nel dispositivo DONET con il chip ESP8266 e si sono riscontrate numerose difficoltà dovute all'allocazione di memoria del codice. Infatti, ESP8266 ha una SRAM di soli 36 kB e l'intero pacchetto di firmware DONET in formato *.py* occupa 58 kB di memoria, il che significa che va ben oltre l'occupazione disponibile del chip, senza considerare la memoria necessaria alla compilazione in runtime ed esecuzione del codice. Si è provato anche a compilare sul PC i moduli, utilizzando il compilatore di MicroPython *mpy-cross*, riducendo le dimensioni delle librerie in formato *.mpy* a 28 kB, ma anche in questo modo non si ha abbastanza memoria per l'allocazione dinamica durante l'esecuzione. Si è passati allora alla tecnica dei moduli congelati, compilando esternamente al chip i moduli e inglobandoli alle librerie di base MicroPython, spostando quindi le librerie DONET dalla SRAM alla flash ROM e liberando quasi tutto lo spazio. In questo modo si è riusciti ad eseguire quasi tutto il codice DONET, escludendo la parte dell'aggiornamento OTA e la relativa libreria. Questo è il massimo risultato ottenibile programmando la scheda in linguaggio MicroPython, dimostrando che il chip ESP8266 non è adatto a sostenere il programma e confermando l'idea di sostituirlo con la versione avanzata ESP32 di memoria SRAM pari a 520 kB.

È stato riprogettato l'hardware di DONET, sostituendo il chip con ESP32 e i risultati sono stati ottimali. Infatti, DONET è riuscito ad eseguire l'intero firmware nel formato *.py*, permettendo anche l'aggiornamento OTA. Il dispositivo è in grado di connettersi correttamente al WiFi, a misurare le grandezze di temperatura, umidità relativa, luminosità, CO_2, TVOC e a pubblicare i dati nel broker MQTT. Infatti, collegandosi tramite un software come *MQTT.fx* al broker dell'azienda, è possibile visualizzare i messaggi inviati da DONET, sottoscrivendosi ai topic *measure_mpy* e *debug_mpy*. La Figura 6.1 mostra la struttura dei messaggi in arrivo al broker MQTT nel topic *measure_mpy* (Figura 6.1a) e *debug_mpy* (Figura 6.1b). Il topic *measure_mpy* è quello in cui si visualizzano i dati relativi alle misure effettuate dal dispositivo, mentre in *debug_mpy* si può controllare da quanto tempo DONET è connesso.

{"mac.address": "4c:11:ae:fd:f2:50", "temperature.celsius": "28.1", "relative.humidity": "54.6", "light.lux": "28.3", "co2.ppm": "442", "tvoc.ppb": "6"}

(a) *Topic measure_mpy*

WSN_fdf250: connected for 2h 12m 50s

(b) *Topic debug_mpy*

Figura 6.1: Messaggi in arrivo al broker MQTT

Di seguito vengono mostrate alcune misure svolte da DONET all'interno di una camera da letto di un'abitazione. Per poter osservare le letture dei dati è possibile accedere al sito web *http://donet.it* la cui pagina iniziale è mostrata in Figura 6.2 e si crea un account per collegare l'indirizzo MAC del dispositivo in possesso.

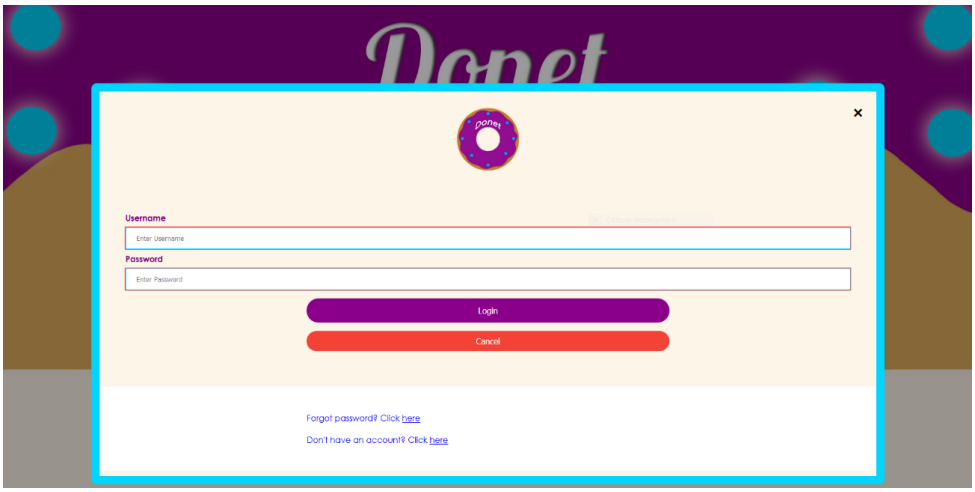


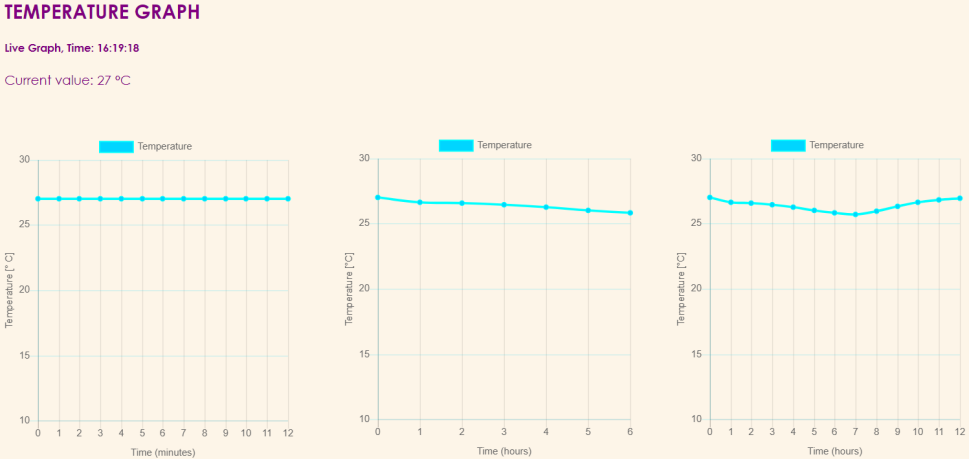
Figura 6.2: Sito web DONET

Accedendo alle misure di DONET nel sito viene visualizzata una schermata con tutti i grafici degli andamenti delle grandezze fisiche misurate, in particolare:

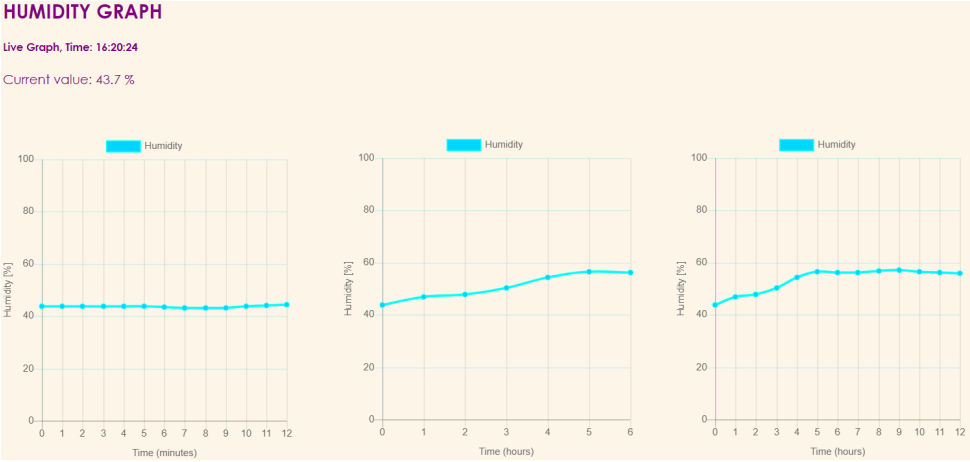
- la temperatura in Figura 6.3a
- l'umidità in Figura 6.3b
- la luminosità in Figura 6.2c
- l'anidride carbonica in Figura 6.2d
- i composti organici volatili totali in Figura 6.2e

Per ognuna di queste grandezze sono mostrati l'ora, il valore misurato durante l'ultima lettura e tre grafici: il primo mostra la variazione del parametro negli ultimi 12 minuti, il secondo nelle

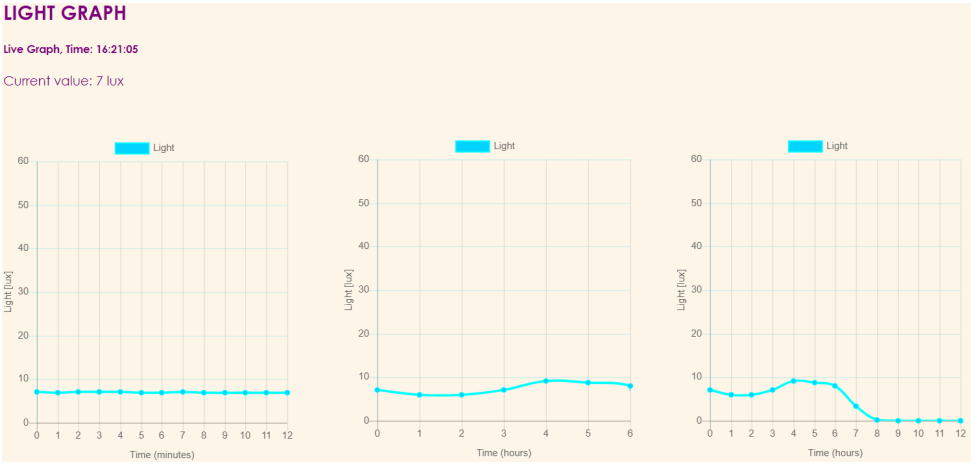
ultime 6 ore, mentre il terzo nelle ultime 12 ore. In Figura 6.2, sono mostrati gli andamenti nella fascia oraria dalle 4 alle 16. In una prima analisi delle misure ottenute si è notato che il sensore di rilevamento della temperatura è soggetto al calore dissipato dal microcontrollore e dall'antenna incorporata, alterando i valori aumentandoli di circa 2-3°Crispetto ad altri dispositivi di misurazione, perciò è stato applicato al codice una correzione dell'11%, calcolato dal confronto dei dati. Osservando invece le misure della luminosità, è facilmente intuibile che i valori sono troppo bassi. Questo difetto è dovuto al case di DONET che copre il sensore abbassando i valori, infatti, togliendo la parte superiore del case, si ottengono misure molto simili a quelle misurate da uno smartphone.



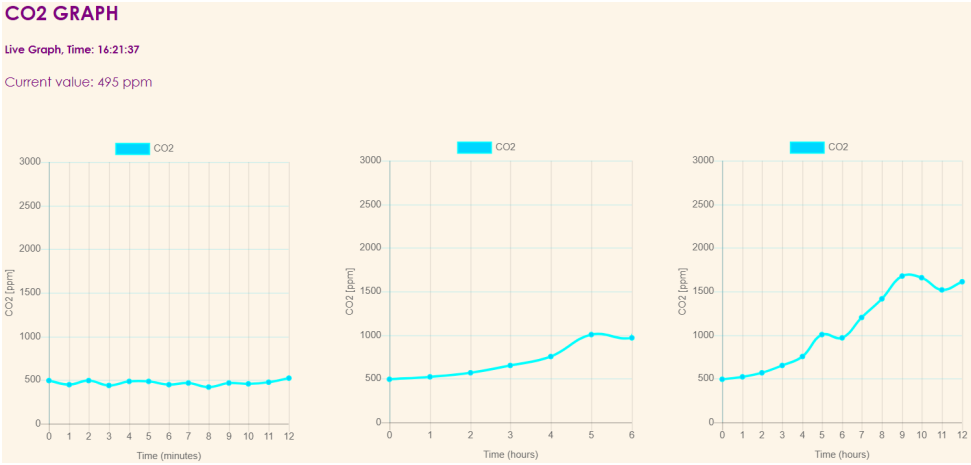
(a) *Temperatura in °C*



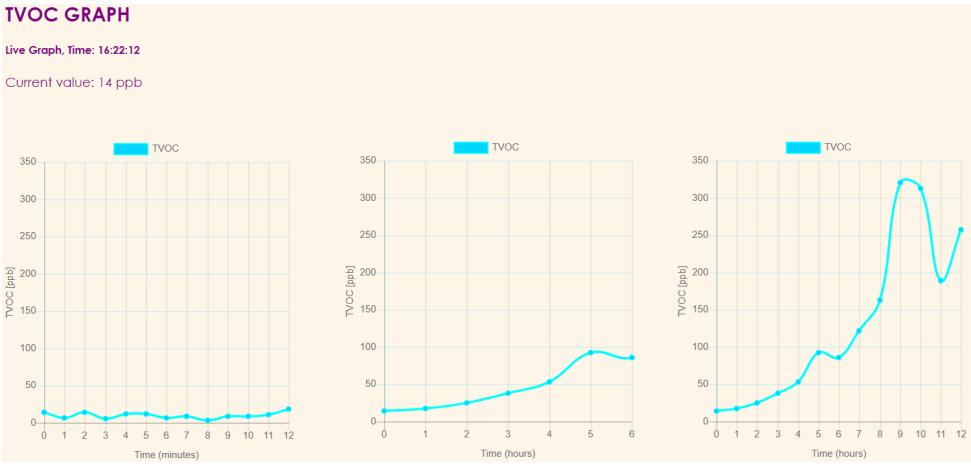
(b) *Umidità relativa*



(c) *Luminosità in lux*



(d) *CO_2 in ppm*



(e) *TVOC in ppb*

Figura 6.2: Andamenti delle grandezze fisiche rilevate

Per testare il funzionamento dell'algoritmo per l'aggiornamento da remoto via OTA, sono state realizzate tre versioni di firmware per il dispositivo. Le librerie *wifimgr.py*, *umqttsimple.py*,

ota_updater.py, *led_donet.py* che non sono modificate durante gli aggiornamenti, sono caricate sulla scheda assieme alla prima versione del firmware. La differenza tra le versioni è che ognuna aggiunge a DONET la funzionalità di un sensore. La libreria *donet.py* che gestisce tutto l'algoritmo del dispositivo, ad ogni versione ovviamente cambia a causa della gestione del sensore che si aggiunge.

- **Versione 1.0**. Fornisce a DONET le librerie *sht21.py* e *donet_1.0.py* permettendo la misura della temperatura e dell'umidità relativa.
- **Versione 2.0**. Aggiunge a DONET la libreria *veml7700.py* e sostituisce *donet_1.0.py* con *donet_2.0.py*, permettendo anche la misura della luminosità.
- **Versione 3.0**. Aggiunge a DONET la libreria *ccs811.py* e sostituisce *donet_2.0.py* con *donet_3.0.py*, permettendo anche la misura della CO_2 e del TVOC.

Dopo aver caricato sul chip la versione 1.0 del firmware e completando la procedura di configurazione, DONET inizia a inviare al broker MQTT messaggi contenenti le misure di temperatura e umidità. Tenendo premuto per 4 secondi il pulsante BOOT, il microcontrollore avvia la procedura OTA, verificando il numero dell'ultima versione del firmware presente nel repository GitHub e lo confronta con quello attuale. Se è maggiore, allora avvia il download della versione 2.0, come nella Figura 6.3a, dove sono mostrati i messaggi che il chip stampa in un monitor seriale durante l'aggiornamento. DONET è quindi in grado di aggiungere alle misure quelle della luminosità della stanza. Chiedendo un nuovo controllo dell'aggiornamento, il dispositivo scarica e installa la versione 3.0 come mostrato in Figura 6.3b aggiungendo anche la misura di CO_2 e TVOC.

```
** USR BUTTON PRESSED
** LONG USR PRESS - call ota updater
Checking version...
        Current version:  1.0
        Latest version:  2.0
New version available, will download and install on next reboot
New update found:  2.0
        Downloading:  next/__init__.py
        Downloading:  next/donet.py
        Downloading:  next/sht21.py
        Downloading:  next/veml7700.py
Update installed ( 2.0 ), will reboot now
```

(a) *Download versione 2.0*

```
** LONG USR PRESS - call ota updater
Checking version...
        Current version:  2.0
        Latest version:  3.0
New version available, will download and install on next reboot
New update found:  3.0
        Downloading:  next/__init__.py
        Downloading:  next/ccs811.py
        Downloading:  next/donet.py
        Downloading:  next/sht21.py
        Downloading:  next/veml7700.py
Update installed ( 3.0 ), will reboot now
```

(b) *Download versione 3.0*

Figura 6.3: Monitor seriale durante l'aggiornamento OTA

CONCLUSIONI E SVILUPPI FUTURI

In questa tesi si è visto come realizzare un firmware per un dispositivo che funge da nodo di una Wireless Sensor Network, in grado di misurare alcune grandezze fisiche all'interno di ambienti indoor. In particolare, è stato utilizzato il dispositivo DONET progettato dall'azienda *IDEA Soc. Coop.* in grado di monitorare la temperatura, l'umidità relativa, la luminosità, l'anidride carbonica e i composti organici volatili nella stanza in cui è installato. Il dispositivo è connesso alla rete Internet tramite un router WiFi locale, pubblicando i dati attraverso il protocollo di comunicazione MQTT in un gateway di raccolta, il quale si occupa di inviare le misure a un database e a un sito web, per controllare gli andamenti delle grandezze. DONET si è evoluto, introducendo un servizio di aggiornamento del firmware Over The Air in modo da distribuire gli aggiornamenti da remoto senza dover intervenire in ogni singolo dispositivo della rete wireless. OTA si serve di un repository privato su GitHub nel quale caricare le versioni del codice, quindi DONET è in grado di accedere attraverso un token di autenticazione e scaricare gli aggiornamenti. Inoltre, è stato verificato che il chip installato nel dispositivo non è adatto a sostenere questa nuova implementazione, per cui è stata fatta un'ulteriore progettazione a livello hardware, installando un chip più potente e con più memoria. È stato anche cambiato il linguaggio di programmazione del firmware, preferendo MicroPython al C/C++, poiché è un linguaggio di alto livello, permettendo di scrivere un codice più corto, più semplice, più veloce e con una leggibilità più fluida.

DONET è un dispositivo in fase di sviluppo da migliorare sia dal punto di vista hardware e software. Per quanto riguarda le componenti hardware attuali, è necessario spostare il sensore di temperatura, in quanto le misurazioni sono alterate dalla dissipazione di calore del microcontrollore e dell'antenna. Anche il sensore di luminosità deve essere spostato perché l'attuale case del dispositivo lo copre abbassando le letture. La soluzione migliore potrebbe essere la progettazione di un nuovo case. Si vorrebbero aggiungere nuovi sensori per la misura di altre grandezze fisiche, come un microfono per il rumore ambientale, le polveri sottili (PM 2.5 e PM 10), la formaldeide, il fumo. È previsto anche l'aggiunta di un display per la visualizzazione delle letture o per la segnalazione di messaggi di alert. Con questi sensori nuovi si potrebbe valutare la realizzazione

di più versioni di DONET differenti dal punto di vista hardware ma allo stesso tempo identici, predisponendo la versione base ad ospitare comunque tutti i sensori, mentre per il firmware, può risultare importante l'implementazione dell'aggiornamento OTA. Si potrebbe anche valutare la modifica del sistema di alimentazione utilizzando una batteria che riesca ad alimentare DONET per un tempo soddisfacente. Per quanto riguarda le migliorie nel software, le idee sono tante. Si sta lavorando sull'integrazione di DONET con Alexa e Google Home. Si vorrebbe integrare il dispositivo anche con elementi domotici della casa, per poi implementare algoritmi di machine learning e di intelligenza artificiale, offrendo un'ulteriore analisi dei dati raccolti dai dispositivi come per esempio la valutazione della qualità dell'aria e dei consumi energetici. Infine, si potrebbe migliorare la piattaforma IoT, con l'aggiunta di funzionalità in più nell'applicazione web e la realizzazione di un'applicazione per smartphone.

Bibliografia

[1] A. Rubechini. Progettazione di una Wireless Sensor Network per la gestione del benessere termico igrometrico in un edificio. Master's thesis, Università Politecnica delle Marche, 2020.

[2] Wireless sensor network. `https://it.emcelettronica.com/wireless-sensor-network`, 2019.

[3] M. A. Matin and M. M. Islam. *Wireless Sensor Networks - Technology and Protocols*. IntechOpen, 2012.

[4] F. De Stefani. Sistema di monitoraggio ambientale tramite WSN. Master's thesis, Università degli Studi di Pavia, 2008.

[5] F. Spagnolo. Progettazione di un sistema di monitoraggio ambientale tramite rete di sensori wireless. Master's thesis, Università degli Studi di Padova, 2011.

[6] A. Pérez, D. Rivas, M. Huerta, R. Clotet, R. Gonzalez, R. Alvizu, and T. Vivas. QoS in Wireless Sensor Networks: A Survey. Simòn Bolivar University, 2009.

[7] MQTT: Il Protocollo di comunicazione dell'IoT. `http://www.itdistribuzione.com/portale/html/NewsPage.html?idNews=2954`, 2015.

[8] MQTT. `https://www.internet4things.it/iot-library/mqtt-cose-e-come-funziona-il-protocollo-alla-base-delliot/`, 2020.

[9] Aggiorniamo ESPertino con l'OTA. `https://it.emcelettronica.com/aggiorniamo-espertino-con-lota`, 2018.

[10] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter. Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles. *IEEE Communications Magazine*, 58(2):35–41, 2020.

[11] X. He, M. Papa, and R. Gamble. Extending Over-the-Air Libraries to Secure ESP8266 Updates. In *2019 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–6, 2019.

[12] K. Kerliu, A. Ross, G. Tao, Z. Yun, Z. Shi, S. Han, and S. Zhou. Secure Over-The-Air Firmware Updates for Sensor Networks. In *2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops (MASSW)*, pages 97–100, 2019.

[13] D. Frisch, S. Reibmann, and C. Pape. An Over the Air Update Mechanism for ESP8266 Microcontrollers. Fulda University of Applied Sciences, Fulda, Germany, 2017.

[14] H. Chandra, E. Anggadjaja, P. S. Wijaya, and E. Gunawan. Internet of Things: Over-the-Air (OTA) firmware update in Lightweight mesh network protocol for smart urban development. In *2016 22nd Asia-Pacific Conference on Communications (APCC)*, pages 115–118, 2016.

[15] Over-the-Air (OTA) Updates in Embedded Microcontroller Applications: Design Trade-Offs and Lessons Learned. `https://www.analog.com/en/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html#`, 2018.

[16] API GitHub. `https://developer.github.com/v3/`, 2020.

[17] OTA in ESP32. `https://medium.com/@ronald.dehuysser/micropython-ota-updates-and-github-a-match-made-in-heaven-45fde670d4eb`, 2018.

[18] Espressif Systems. *ESP-IDF Programming Guide*, 2020.

[19] ESP32 vs ESP8266 – Pros and Cons. `https://makeradvisor.com/esp32-vs-esp8266/`, 2020.

[20] A. Maier, A. Sharp, and Y. Vagapov. Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things. In *2017 Internet Technologies and Applications (ITA)*, pages 143–148, 2017.

[21] Sensirion The Sensor Company. *Datasheet SHT21*, 2010.

[22] Vishay Semiconductors. *Datasheet VEML7700*, 2016.

[23] ams. *Datasheet CCS811*, 2016.

[24] Texas Instruments. *Datasheet PCF8575*, 2005.

[25] The MicroPython project. `https://github.com/micropython/micropython`, 2020.

[26] MicroPython. `http://www.micropython.org/`, 2020.

[27] R. K. Kodali and K. S. Mahesh. Low cost ambient monitoring using ESP8266. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 779–782, 2016.

[28] MicroPython – Compilazione MicroPython per ESP8266. `https://www.microdev.it/wp/it/2018/06/25/micropython-compilazione-micropython-per-esp8266/`, 2018.

[29] Tutorial: Getting Started with MicroPython on ESP32, M5Stack, and ESP8266. `https://lemariva.com/blog/2020/03/tutorial-getting-started-micropython-v20`, 2020.

[30] SHT21 GitHub Library. `https://github.com/jaques/sht21_python`, 2017.

[31] VEML7700 GitHub Library. `https://github.com/palouf34/veml7700`, 2019.

[32] CCS811 GitHub Library. `https://github.com/Notthemarsian/CCS811`, 2018.

[33] WiFi Manager GitHub Library. `https://github.com/tayfunulu/WiFiManager`, 2018.

[34] MQTT GitHub Library. `https://github.com/micropython/micropython-lib/tree/master/umqtt.simple`, 2018.

[35] OTA ESP32 GitHub Library. `https://github.com/rdehuyss/micropython-ota-updater`, 2018.