

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Contesto e limiti delle onde elettromagnetiche .....	3
1.2	Motivazione dell'approccio acustico .....	4
1.3	Richiami teorici sulla propagazione acustica .....	4
1.4	Motivazioni sperimentali e obiettivi della tesi .....	5
1.5	Contributi attesi .....	5
<b>2</b>	<b>Stato dell'Arte</b>	<b>6</b>
2.1	Tecnologie IoT .....	6
2.2	Comunicazione in ambienti sotterranei .....	6
2.2.1	Radio frequenze .....	7
2.2.2	Onde acustiche .....	8
2.3	Applicazione della letteratura alla tesi .....	9
<b>3</b>	<b>Design del Protocollo</b>	<b>10</b>
3.1	Architettura generale del sistema .....	10
3.2	Livello Fisico .....	10
3.2.1	Ingresso .....	11
3.2.2	Uscita .....	16
3.3	Livello Bit .....	18
3.3.1	Decodifica .....	19
3.3.2	Codifica .....	20
3.4	Livello Link .....	20
3.4.1	Procedura di registrazione .....	21
3.4.2	Indirizzamento e instradamento .....	21
3.4.3	Gestione dei Token (TKN) .....	22
3.4.4	Comandi di controllo e di rete .....	23
3.4.5	Indirizzi di rete .....	23
3.5	Livello Applicazione .....	24
<b>4</b>	<b>Implementazione Hardware</b>	<b>26</b>
4.1	Tecnologie di base .....	26

4.1.1	Panoramica dell'ESP32 .....	26
4.1.2	Periferiche di Input/Output.....	28
4.2	Progettazione digitale .....	32
4.2.1	Collegamento del microfono digitale I <sup>2</sup> S.....	32
4.2.2	Collegamento dell'amplificatore digitale I <sup>2</sup> S .....	34
4.2.3	Sistema di segnalazione LED .....	35
4.2.4	Bottone di Hotspot .....	37
4.2.5	Circuito completo .....	37
4.2.6	Realizzazione del PCB .....	38
<b>5</b>	<b>Implementazione Software: Design del Protocollo</b>	<b>40</b>
5.1	Architettura generale del sistema .....	40
5.2	Emissione dei toni.....	41
5.3	Campionamento e bufferizzazione .....	42
5.4	Trasformata veloce di Fourier .....	44
5.5	Decodifica spettrale .....	45
5.6	Traduzione frequenza-bit .....	47
5.7	Ricevimento e trasmissione dei bit (Packers) .....	50
5.7.1	Ricezione dei bit .....	50
5.7.2	Trasmissione dei bit .....	51
5.8	Instrandamento e protocollo .....	51
5.9	Sensore di movimento .....	52

# Capitolo 1

## Introduzione

### 1.1 Contesto e limiti delle onde elettromagnetiche

Le reti di sensori in ambienti sotterranei pongono sfide peculiari alla comunicazione wireless. La propagazione delle *onde elettromagnetiche* nel sottosuolo soffre di attenuazioni elevate dovute alla permittività, alla conducibilità del terreno e all'umidità, con conseguente riduzione drastica della portata e dell'affidabilità dei link radio [1].

Alcune soluzioni proposte in letteratura includono l'uso di frequenze superhigh ed ultra-high (SHF, UHF) con l'obiettivo di implementare sistemi di tracking e monitoraggio in miniere di carbone, gallerie o condotti [2]. Tuttavia, queste frequenze a causa della loro natura fisica sono soggette a forti perdite di segnale e riflessioni multipath, limitando la copertura a range dai 10 ai 33 metri in condizioni ottimali con l'ausilio di antenne direzionali con un'altezza pari a 1.2 metri l'una; questa tecnologia inoltre mostra tutta la sua vulnerabilità in presenza di curve strette ( $90^\circ$ ) o ostacoli.

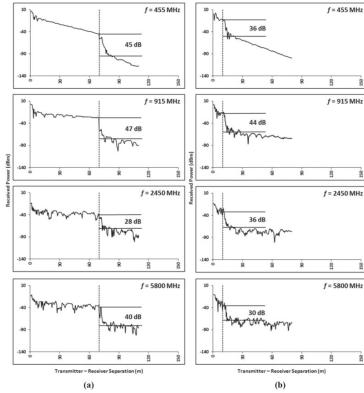


Figura 1.1: Corner Loss  $\sim 30$  dB in un condotto con angolo di  $90^\circ$  per radio frequenze HF/SHF [2].

## 1.2 Motivazione dell'approccio acustico

Da questa problematica nasce la presente tesi, che si pone l’obiettivo di progettare e validare un **protocollo di comunicazione acustica** per reti **master–slave** in ambienti sotterranei, che dovrà essere: **efficiente, robusta ed economica**.

L’idea è sfruttare la *propagazione del suono nell’aria* presente in cavità, condotti o tunnel, trattando l’aria come un canale guida all’interno degli spazi confinati, e più in generale impiegare l’onda acustica come mezzo portante laddove il canale elettromagnetico è troppo penalizzato.

## 1.3 Richiami teorici sulla propagazione acustica

In effetti, l’aria può essere modellata come un fluido compressibile: le variazioni locali di pressione e densità generate da una sorgente si propagano come onde longitudinali, la cui dinamica è descritta dall’equazione delle onde acustiche. La velocità di propagazione, che in condizioni standard è circa 343 m/s, dipende da temperatura, pressione e composizione del gas, secondo la relazione  $c = \sqrt{\gamma p_0 / \rho_0}$ .

Questa visione permette di trattare l’aria non solo come “spazio vuoto”, ma come un vero e proprio *canale fisico*, caratterizzato da attenuazioni dovute a dispersione geometrica, assorbimento atmosferico e riverberi dovuti alle superfici [3–5].

Il fine ultimo della rete sarà quello di permettere lo scambio di dati tra nodi utilizzando frequenze sonore sub-9kHz, appartenenti al range 1-10kHz: queste frequenze sono scelte per il loro compromesso tra portata e qualità del segnale [6]. L’applicazione principale è destinata al monitoraggio post-disastro, considerando

che in altri contesti l'utilizzo di segnali acustici potrebbe risultare disturbante. I sensori saranno, inoltre, in grado di rilevare la presenza di corpi umani o gas tossici a seguito di crolli o esplosioni, e trasmettere queste informazioni a un nodo master situato in superficie o in una zona sicura.

## 1.4 Motivazioni sperimentali e obiettivi della tesi

La letteratura recente indica che, in scenari confinati, segnali acustici a bassa frequenza possono mantenere un rapporto segnale/rumore utilizzabile su distanze dell'ordine delle decine di metri, con modalità di propagazione *lungo condotti* (ad esempio tubazioni o gallerie) e con modelli di attenuazione prevedibili [7].

Queste evidenze motivano la definizione di un protocollo leggero e robusto (rilevazione spettrale, soglie adattive con ausilio di machine learning) che faccia uso di componenti economici e facilmente integrabili (microfoni/codec, amplificatori, altoparlanti) per creare un *layer fisico* acustico e il relativo *protocollo di accesso*.

## 1.5 Contributi attesi

In sintesi, i contributi attesi sono:

1. modellazione e scelta dei parametri del canale acustico in aria in ambienti confinati;
2. progettazione di un protocollo master-slave basato su pattern di frequenze, ACK e gestione del ritardo casuale;
3. implementazione hardware/software a basso costo;
4. validazione sperimentale mediante misure di SPL a diverse distanze, con stima dei livelli attesi per amplificazioni maggiori tramite traslazione dei valori rilevati.

# **Capitolo 2**

## **Stato dell'Arte**

### **2.1 Tecnologie IoT**

L'Internet of Things (IoT) rappresenta una rete di dispositivi fisici, veicoli incorporati con elettronica, software, sensori e connettività di rete. Questi dispositivi sono progettati per raccogliere e scambiare dati, consentendo di comunicare con l'ambiente circostante. Negli anni diverse tecnologie sono state usate per far comunicare questi dispositivi, la maggior parte di queste si basano su radio frequenze, come LoRa, NB-IoT, ZigBee, Wi-Fi, BLE, fortemente indirizzate ad un uso outdoor o indoor.

Tuttavia, esistono scenari in cui la propagazione elettromagnetica incontra limiti strutturali, ad esempio in ambienti sotterranei, gallerie, miniere o condotti, dove l'attenuazione del segnale cresce a causa della composizione del terreno e dell'umidità. In questi contesti, l'affidabilità della comunicazione wireless tradizionale risulta fortemente compromessa.

### **2.2 Comunicazione in ambienti sotterranei**

La comunicazione in ambienti sotterranei rappresenta una sfida significativa a causa delle caratteristiche uniche di questi ambienti. Le onde elettromagnetiche, comunemente utilizzate per la comunicazione wireless, soffrono di attenuazioni elevate dovute alla permittività, alla conducibilità del terreno e all'umidità, con conseguente riduzione drastica della portata e dell'affidabilità dei link radio [1]. In letteratura è possibile individuare diverse soluzioni proposte per affrontare queste sfide.

In letteratura si riconoscono due linee principali di ricerca:

- l'uso di radio frequenze molto basse (VLF), medie (MF) o più alte (UHF, SHF), adattate a condizioni specifiche;
- l'uso di onde acustiche, che sfruttano la propagazione meccanica del suono attraverso solidi e fluidi.

### 2.2.1 Radio frequenze

Le ricerche sulle onde elettromagnetiche in gallerie e miniere hanno mostrato limiti significativi: la propagazione è fortemente influenzata dalla geometria degli ambienti e dalle proprietà elettriche del mezzo. Gli studi del U.S. National Institute for Occupational Safety and Health's (NIOSH) Office of Mine Safety and Health Research (OMSHR) [2], avvenuti in seguito a crolli di alcune miniere di carbone in U.S. nel 2006. Questi hanno piantato le fondamenta per l'uso di onde elettromagnetiche SHF/UHF in ambienti sotterranei.

Attraverso una serie di antenne direzionali dalla misura di 1.2 metri e ricevitori collocati prima in linea retta ad una distanza crescente, e poi in presenza di curve a  $30^\circ$ ,  $60^\circ$ ,  $90^\circ$  e  $180^\circ$  in condotti sotterranei: hanno dimostrato che, in condizioni ottimali, la copertura può raggiungere i 33 metri, ma la presenza di curve strette, biforcazioni o ostacoli riduce drasticamente la portata [2].

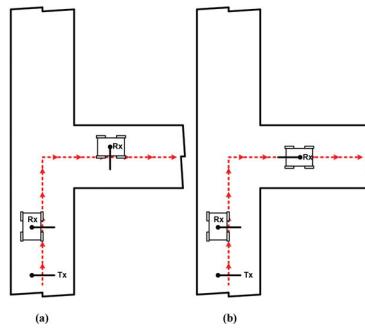


Figura 2.1: Mappa Tunnel U.S. National Institute for Occupational Safety and Health's [2].

Mentre Akyildiz et al. [1] descrivono le difficoltà intrinseche alla propagazione sotterranea a causa di permittività e conducibilità elevate del terreno.

Più in generale, le tecnologie wireless pensate per ambienti sotterranei devono fare i conti con una ridotta penetrazione delle frequenze tradizionalmente usate per IoT. Per questo, l'uso di frequenze estremamente basse (VLF, tra 3–30 kHz) è stato studiato in scenari di emergenza e applicazioni militari, ma presenta limiti di banda e di miniaturizzazione delle antenne [8].

## 2.2.2 Onde acustiche

Un’alternativa promettente alla comunicazione elettromagnetica in ambienti sotterranei è l’impiego di *onde acustiche* (meccaniche), che si propagano attraverso solidi e fluidi (aria, acqua, fanghi di perforazione). A differenza delle onde EM, la propagazione acustica è spesso meno sensibile alla conducibilità elettrica del mezzo e può seguire percorsi guidati (ad es. lungo tubazioni o gallerie), con migliore resilienza a curve e biforazioni [9–11].

**Modello di canale (cenni).** Nei **solidi** (roccia, calcestruzzo, metalli, polimeri), la propagazione è dominata da onde elastiche (longitudinali e di taglio) e, in geometrie guidate (tubi, piastre, travi), da *guided waves* (p. es. onde di Lamb o creep waves) con attenuazione che cresce fortemente con la frequenza e con il disaccoppiamento modale [12].

Nei **fluidi** in condotti (acqua/aria), il canale è tipicamente a bassa frequenza (centinaia di Hz–pochi kHz), con riflessioni multiple e dispersione; in vasche o condotti lunghi si osservano fenomeni di riverbero e frequenze di taglio dei modi guida [9].

Nel **suolo sfuso** (soil), la trasmissione è di tipo acustico/seismico: l’attenuazione dipende da tessitura, umidità e compattazione, ed è marcata oltre poche decine di metri; lo *sweet spot* operativo è in genere sotto i 2–5 kHz per massimizzare SNR a parità di potenza [13, 14].

**Hardware e trasduttori.** Sono impiegati trasduttori piezoelettrici o magnetostrettivi accoppiati meccanicamente al mezzo (collari su tubi, inserti su rocce/pareti, piastre di accoppiamento).

Nei tubi metallici, coppie di attuatori/ricevitori clamp-on permettono comunicazione non invasiva (senza rompere il tubo), sfruttando onde di taglio o di Lamb [10, 12].

Su polimeri (es. tubi MDPE), l’accoppiamento è più debole e la banda utile è più bassa, ma resta praticabile [11].

Nei fluidi, si usano idrofoni/speaker impermeabilizzati; nel suolo, trasduttori sismici compatti e miniaturizzati sono stati dimostrati in prototipi a bassa potenza [13].

**Tecniche di modulazione e codifica.** Dato il canale fortemente dispersivo e soggetto a multi-percorso, si adottano modulazioni robuste: FSK/MFSK e BFSK a bassa frequenza per collegamenti affidabili a bassa velocità; OFDM con stime del canale per massimizzare efficienza in condotti lunghi; e codifica con interleaving e rateless/LDPC per gestire *burst errors* [9, 11]. In tubazioni e solidi guidati, è utile

la *modal selection* (scegliere il modo meno attenuato) e la *sub-carrier spacing* adeguata alle *delay spreads* misurate [11, 12].

**Prestazioni riportate in letteratura.** Risultati sperimentali rappresentativi includono:

- **Suolo (through-soil).** Collegamenti fino a  $\sim 50$  m con  $\sim 20$  bps utilizzando portanti a bassa frequenza e trasduttori compatti; dimostrazione di comunicazione digitale robusta in campi prova agricoli [13]. Survey recenti confermano che, per distanze  $> 10\text{--}20$  m, le velocità realistiche sono dell'ordine di 1–100 bps, con forte dipendenza dal contenuto d'acqua [14, 15].
- **Tubi metallici (acciaio).** Trasmissione di dati via onde elastiche lungo tubazioni esistenti in impianti industriali; dimostrate immagini e pacchetti a centinaia di bps–pochi kbps su decine–centinaia di metri a bassa potenza, sfruttando onde di taglio e di Lamb [10, 12].
- **Tubi in polimero (MDPE).** Misure analitiche/numeriche e in campo mostrano che con segnali a bassa frequenza si ottengono distanze utili per telemetria (decine–centinaia di metri) con bitrate modesto (decine–centinaia di bps) [11, 16].
- **Condotti idrici reali.** Revisione dei test in reti idriche urbane indica fattibilità di reti acustiche in-pipe per monitoraggio/IoT, con vincoli di potenza, sincronizzazione e instradamento; lo stato dell'arte è pre-commerciale ma in rapida evoluzione [9].
- **Fluido non convenzionale: fango di perforazione.** Sistemi *while drilling* usano la colonna di fluido come canale acustico per telemetria a bassa velocità (bps–decine di bps) su centinaia di metri, con elevata robustezza [17].

## 2.3 Applicazione della letteratura alla tesi

La tesi si occuperà quindi della creazione di un protocollo di comunicazione acustica per sistemi distribuiti in ambienti sotterranei, basandosi sui risultati e le tecniche di cui sopra, è possibile quindi dedurre alcuni requisiti chiave: Il protocollo opera nella banda 1–9 kHz. Questa scelta è coerente con la letteratura, che evidenzia come nei condotti fluidi l'impiego di frequenze basse consenta di estendere la portata e ridurre l'attenuazione, fenomeno particolarmente critico per le frequenze più elevate [9]

# Capitolo 3

## Design del Protocollo

### 3.1 Architettura generale del sistema

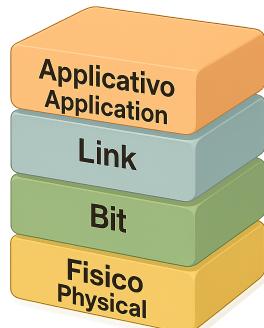


Figura 3.1: Livelli del Protocollo

Il protocollo è strutturato in quattro livelli principali, ognuno con una funzione specifica.

### 3.2 Livello Fisico

Il livello fisico è responsabile della trasmissione dei bit precedentemente composti dal livello Bit. Questo sfrutta uno speaker per la trasmissione e un microfono per la ricezione dei bit che vengono trasmessi mediante coppie superimposte di frequenze.

### 3.2.1 Ingresso

Il microfono I2S (descritto nel [Capitolo 4](#)) è collegato al microcontrollore ESP32 tramite il bus I2S, sfruttando uno dei due canali disponibili. Il segnale audio viene campionato a 48 kHz con una risoluzione di 16 bit, per poi essere gestito mediante due array in swapping, concetto che verrà approfondito successivamente.

La scelta della frequenza di campionamento deriva dal **teorema di Nyquist-Shannon** [18], secondo il quale un segnale può essere ricostruito senza ambiguità se la frequenza di campionamento  $f_s$  è almeno doppia rispetto alla massima frequenza del segnale  $f_{\max}$ :

$$f_s \geq 2f_{\max}.$$

Ne consegue che la massima frequenza rappresentabile è

$$f_{\text{Nyquist}} = \frac{f_s}{2}.$$

Nel nostro caso:

$$f_s = 48 \text{ kHz} \quad \Rightarrow \quad f_{\text{Nyquist}} = 24 \text{ kHz}.$$

Poiché l'orecchio umano percepisce frequenze fino a circa 20 kHz [19], la scelta di 48 kHz garantisce la copertura dell'intero spettro udibile, con un margine di sicurezza di 4 kHz. Frequenze prossime al limite teorico di Nyquist risulterebbero invece difficili da catturare senza aliasing, a causa dei limiti pratici dei filtri anti-alias.

Per l'elaborazione, i campioni vengono raccolti in blocchi di lunghezza  $N = 512$ . Con la frequenza di campionamento fissata:

$$T_{\text{blocco}} = \frac{N}{f_s} = \frac{512}{48 \cdot 10^3} \approx 0.01066 \text{ s}.$$

Ogni blocco di dati rappresenta quindi un intervallo di circa 10.7 ms di segnale audio. Questa durata, successivamente a una attenta analisi effettuata sul software Audacity, si è rivelata sufficiente per poter catturare in modo affidabile i toni.



Figura 3.2: Verifica della cattura del tono a 9kHz con campionamento a 48kHz nel periodo 0-0.010 secondi, utilizzando finestra di Hann su 512 elementi

Una volta acquisito il blocco, viene calcolata la **trasformata veloce di Fourier (FFT)** [20] per analizzare lo spettro in frequenza. La complessità della FFT cresce come  $N \log_2 N$ , e per  $N = 512$  si hanno circa:

$$512 \cdot \log_2(512) = 512 \cdot 9 = 4608$$

operazioni complesse.

Sul microcontrollore ESP32 [21], operante a 120 MHz, questo si traduce in un tempo di elaborazione di circa 0.42 ms per blocco, ovvero  $\sim 0.83 \mu\text{s}$  per campione. In termini di carico computazionale:  $\sim 20$  moltiplicazioni,  $\sim 29$  addizioni/sottrazioni e un'operazione di radice quadrata per campione, pari a  $\sim 99$  cicli di clock.

Questo significa che, a fronte di una finestra temporale di 10.7 ms, l'FFT viene calcolata in meno del 5% del tempo disponibile, consentendo un'elaborazione in tempo reale anche senza multi-threading.

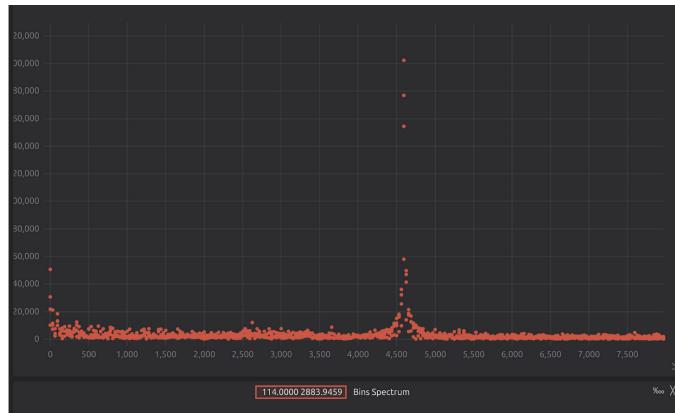


Figura 3.3: Grafico contentente lo spettro di frequenza calcolato tramite FFT su un blocco di 512 campioni acquisiti a 48 kHz.

**FFT, bin e asse delle frequenze** L'algoritmo FFT non calcola direttamente lo spettro in frequenza continua, bensì restituisce  $N$  valori discreti  $X[k]$  detti *bin*, con indici

$$k = 0, 1, \dots, N - 1.$$

Questi bin rappresentano i coefficienti della trasformata discreta e contengono l'informazione spettrale su griglie di frequenza equispaziate. Per  $N = 512$  e frequenza di campionamento  $F_s = 48$  kHz, ciascun bin corrisponde alla frequenza

$$f_k = k \frac{F_s}{N}, \quad \Delta f = \frac{F_s}{N} = \frac{48\,000}{512} = 93.75 \text{ Hz}.$$

Dunque l'asse  $x$  del grafico, che mostra le frequenze in Hz, è in realtà una *ridenominazione* dei bin calcolati dall'FFT. Ogni valore sull'asse delle ascisse non è una misura continua, ma la mappatura del bin discreto  $k$  nella corrispondente frequenza  $f_k$ .

**Simmetria dello spettro** Poiché il segnale è reale, vale la simmetria

$$X[N - k] = \overline{X[k]} \quad \Rightarrow \quad |X[N - k]| = |X[k]|.$$

Per questo motivo lo spettro viene spesso mostrato soltanto fino al bin  $N/2$ , ossia da  $k = 0$  a  $k = 256$ . In particolare, il bin  $k = 256$  corrisponde alla frequenza massima non ambigua, cioè la frequenza di Nyquist:

$$f_{256} = 256 \cdot \frac{48\,000}{512} = 24 \text{ kHz.}$$

Durante la creazione di questo protocollo sono emerse diverse difficoltà legate alla presenza di rumore ambientale, questa condizione ha reso necessario l'uso di tecniche di filtraggio.

### 3.2.1.1 Filtraggio e riconoscimento dei toni

Una parte cruciale del processo di design del protocollo è stata la scelta delle frequenze da utilizzare per la trasmissione dei dati. I picchi che vengono selezionati dalla fase di filtraggio devono appartenere a un insieme di frequenze predefinite, così da poter essere riconosciuti dal **Livello Bit**.

La seguente tabella riporta l'insieme delle frequenze adottate, distinguendo quelle di tipo *carrier* da quelle destinate al trasferimento dei dati:

Frequenza [Hz]	Tipo
1000	Carrier
1400	Data
1800	Data
2200	Data
2600	Data
3000	Data
3400	Data
3800	Data
4200	Data
4600	Data
5000	Data

Frequenza [Hz]	Tipo
5400	Data
5800	Data
6200	Data
6600	Data
7000	Data
7400	Data
7800	Data
8200	Data
8600	Carrier
9000	Carrier

Tabella 3.1: Frequenze e tipi di segnale

Il filtraggio avviene nel dominio della frequenza, dopo l'applicazione della trasformata FFT a 512 punti. A differenza dei filtri digitali convenzionali (IIR/FIR), la selezione dei toni non si basa su maschere statiche ma su un insieme di procedure che rendono il sistema adattivo al rumore e preciso nell'identificazione dei picchi. Inizialmente, quando il protocollo era ancora nelle sue fasi embrionali, era stato deciso di utilizzare una semplice soglia fissa (filtro passa-basso) per discriminare i picchi delle frequenze predefinite dal rumore. Questa soluzione, tuttavia, si è rivelata inefficace: i microfoni presentano una risposta in frequenza che degrada sulle alte frequenze, rendendo i livelli in dB di queste ultime attenuati al punto da non superare la soglia prefissata. Inoltre, il rumore ambientale non è mai costante ma varia nel tempo e nello spettro, il che rendeva difficile definire un limite statico in grado di funzionare in tutte le condizioni.

In un'evoluzione successiva si è quindi ipotizzato l'impiego di soglie fisse ma differenziate per ciascuna frequenza, così da compensare la risposta non piatta del microfono.

Per calcolare tali soglie è stato utilizzato un algoritmo di regressione lineare, in grado di stimare la retta che meglio approssima l'andamento della soglia minima:

$$y = \beta + \beta x + \varepsilon \rightarrow y = -301.751324 \times x + 48531.689491$$

dove  $x$  rappresenta il numero del bin. La [Figura 3.4](#) mostra lo spettro in frequenza di tre toni (basso, medio e alto) e mette in evidenza l'attenuazione introdotta dalla risposta del microfono.



Figura 3.4: Grafico contenente lo spettro di frequenza di tre frequenze (bassa, media, alta), utilizzato per identificare l'attenuazione in frequenza del microfono.

Il sistema attuale ha superato queste limitazioni introducendo una catena di elaborazione più robusta. In primo luogo viene stimato il livello medio di rumore calcolando la media delle ampiezze spettrali:

$$\text{noise\_floor} = \frac{1}{N} \sum_{k=0}^{N-1} |X[k]|$$

dove  $X[k]$  è il modulo del  $k$ -esimo bin. A partire da questa misura viene definita una soglia dinamica, proporzionale al rumore, che permette di adattarsi alle condizioni del segnale: un picco viene considerato valido solo se la sua ampiezza supera di almeno otto volte il livello medio di rumore. Per ridurre ulteriormente i falsi positivi, ogni frequenza candidata deve corrispondere a un **massimo locale**, ossia il **bin identificato** deve avere ampiezza superiore rispetto ai sei **bin adiacenti a sinistra e a destra**.

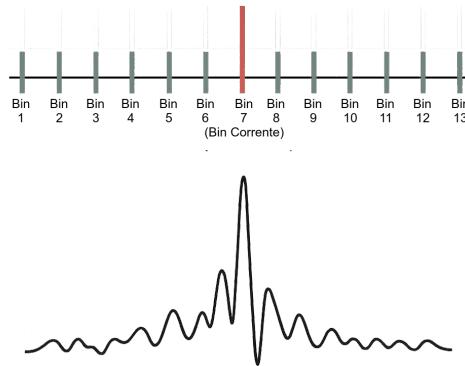


Figura 3.5: Picco Locale nell'intorno dei sei Bins

**Interpolazione parabolica** A questo punto, per migliorare la risoluzione in frequenza oltre i limiti del singolo bin FFT, viene applicata un'**interpolazione parabolica** basata sui valori  $X[k - 1]$ ,  $X[k]$  e  $X[k + 1]$ , secondo la formula

$$p = \frac{1}{2} \frac{\alpha - \gamma}{\alpha - 2\beta + \gamma},$$

dove  $\alpha = |X[k - 1]|$ ,  $\beta = |X[k]|$  e  $\gamma = |X[k + 1]|$ . La frequenza stimata diventa così  $f_k + p \Delta f$ , con  $\Delta f = 93.75$  Hz, ottenendo una risoluzione sub-bin.

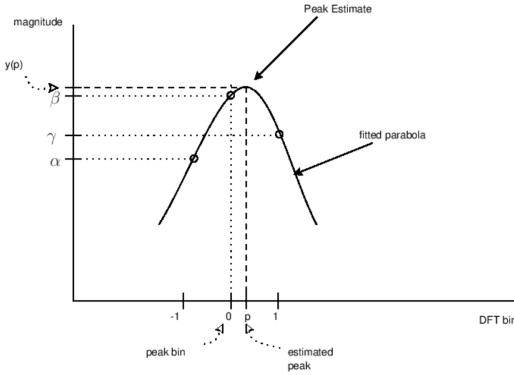


Figura 3.6: Interpolazione Parabolica [22]

È inoltre disponibile, sebbene disattivato di default, un modulo di regressione lineare che consente di compensare eventuali pendenze dello spettro in condizioni particolarmente critiche. Infine, i **picchi confermati vengono incapsulati in una struttura dati (`struct_tone_frequencies`) e inoltrati al Livello Bit**, che li utilizza per ricostruire i bit associati ai canali *master*, *slave* e *config*.

### 3.2.2 Uscita

Il livello fisico si occupa anche della trasmissione dei dati, convertendo coppie di frequenze ricevute dal Livello Bit in segnali audio. Questa operazione viene eseguita mediante la sintesi digitale di due toni sinusoidali, che vengono sommati, sistemati in fase, sistemati in ampiezza ed infine inviati attraverso il bus I2S ad un amplificatore di potenza (descritto nel [Capitolo 4](#)).

**Sintesi dei toni** L'emissione di una coppia di frequenze superimposte è un'operazione che coinvolge l'utilizzo di diverse tecniche di sintesi digitale.

In primo luogo viene calcolato il numero necessario di campioni che compongono la **sinusoide**, per far ciò è necessario utilizzare la stessa frequenza di campionamento adottata per l'acquisizione, ovvero 48 kHz, ed la durata d'emissione che è stata fissata a 0.024 s. La scelta di questa durata deriva dalla finestra temporale di 10.7 ms utilizzata per l'analisi FFT, quindi 24 ms è un compromesso che consente di avere un segnale sufficientemente lungo per essere percepito e demodulato, evitando così che la **finestra di ascolto (FFT)** possa essere in mezzo tra un tono e l'altro, rendendo così difficile la demodulazione. Con questa durata, quindi, si avrà sempre una serie di 512 campioni in ingresso che conterranno, sicuramente, i toni.

Il numero di campioni necessari per la sintesi è quindi

$$N = f_s \cdot T \quad (3.1)$$

$$N = 48\,000 \text{ Hz} \cdot 0.024 \text{ s} = 1152 \text{ campioni} \quad (3.2)$$

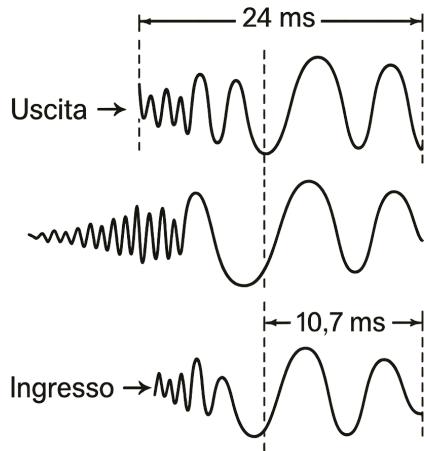


Figura 3.7: Finestra d'ascolto della FFT sul segnale emesso

A seguito di diversi test, dove è stato riscontrato un "fruscio" tra l'emissione di un tono e l'altro, è stato implementato un sistema di allineamento di fase. Questo funziona attraverso una variabile che tiene traccia della fase dell'ultimo campione emesso, in questo modo il primo campione del nuovo tono sarà sempre in fase con l'ultimo campione del tono precedente. Ciò consente di evitare discontinuità nel segnale che si traducono in rumore udibile.

La sintesi di ciascuna sinusoida avviene quindi secondo la formula

$$x[n] = A \cdot \frac{\sin(\phi_1) + \sin(\phi_2)}{2} \quad (3.3)$$

formula semplificata dove  $A$  è l'ampiezza del segnale,  $\phi_1$  e  $\phi_2$  sono le fasi delle due sinusoidi calcolate come

$$\phi = \frac{2\pi f}{f_s} \quad (3.4)$$

### 3.3 Livello Bit

Il Livello Bit si occupa di convertire le frequenze ricevute dal Livello Fisico in bit, e viceversa. Il pacchetto con cui il livello Bit comunica con il Livello Fisico è la seguente struttura dati

Master	Slave			Config				
Frequenza 1	Frequenza 2	Frequenza 3	Frequenza 1	Frequenza 2	Frequenza 3	Frequenza 1	Frequenza 2	Frequenza 3

Tabella 3.2: Struttura di comunicazione tra Livello Bit e Livello Fisico

Ogni gruppo (master, slave, config) rappresenta un gruppo a sè stante, che può essere utilizzato per diversi scopi, questo avviene al fine di evitare conflitti tra i nodi, in quanto il ruolo è già indicato dalle frequenze che usano. Il livello Fisico quindi restituisce per ogni gruppo una struttura dati contenente 2 frequenze, se le 3 frequenze fossero tutte presenti allo stesso momento si avrebbe, allo stato attuale della tecnologia un errore Multi-Tone. In ogni gruppo, le frequenze seguono una logica definita come segue:

- La prima frequenza rappresenta quale **Signal Code** utilizzare dal lato sinistro della tabella che segue
- La seconda frequenza è la portante ed è sempre la stessa per ogni gruppo, questa serve successivamente per la decodifica
- La terza frequenza rappresenta quale **Signal Code** utilizzare dal lato destro della tabella che segue

Frequenza	Signal Code	Portante	Signal Code
1000			
1400	(0) Bits: 0		
1800	(1) Bits: 00		
2200	(2) Bits: 000		
2600	(3) Bits: 0000		
3000	(4) Bits: 00000		
3400	(5) Bits: 000000		
3800	(6) Bits: 0000000		
4200	(7) Bits: 0000000 0000000		
4600	(8) Bits: 0000000 0000000 000000		
5000			(9) Bits: 1
5400			(10) Bits: 11
5800			(11) Bits: 111
6200			(12) Bits: 1111
6600			(13) Bits: 11111
7000			(14) Bits: 111111
7400			(15) Bits: 1111111
7800			(16) Bits: 1111111 1111111
8200			(17) Bits: 1111111 1111111 1111111
8600		Config Carrier	
9000		Slave Carrier	

Tabella 3.3: Mappatura frequenze, codici e portanti

L'utilizzo di questo schema di codifica, viene dopo l'esecuzione di diversi test, in quanto in origine i dati venivano trasmessi utilizzando 3 frequenze per ogni gruppo:

- La prima frequenza rappresentava se era uno 0
- La seconda frequenza era la portante
- La terza frequenza rappresentava se era un 1

L'utilizzo della [Tabella 3.3](#) ha permesso di comprimere i pacchetti di dati a livello di bit, in quanto ora è possibile trasmettere più bit con una sola frequenza.

Questo ha permesso di aumentare la velocità di trasmissione fino a 5(codes per second), nel caso più favorevole **30bps**, che seppur bassa è comunque un miglioramento rispetto alla versione precedente che raggiungeva al massimo 5bps.

Inoltre, grazie a questa codifica, è possibile trasmettere il codice 8 (21 volte 0) con estrema facilità, questo corrisponde alla ripetizione del carattere ASCII NUL per tre volte, che, nel protocollo corrisponde all' End of Packet (EOP).

### 3.3.1 Decodifica

In fase di decodifica, come preannunciato, il Livello Bit riceve dal Livello antecedente una struttura dati contenente fino a 2 frequenze per ogni gruppo (master,

slave, config). Questo associa la frequenza data al corrispondente **Signal Code**, attraverso

$$s[n] = \frac{f - base}{step} = \frac{f - 1000}{400} \quad (3.5)$$

Dove **base** è la frequenza minima (1000Hz), **step** è la differenza tra una frequenza e l'altra (400Hz) e **f** è la frequenza ricevuta.

Questo permette di ottenere il **Signal Code** che può essere compreso tra 0 e 17, che viene successivamente convertito in bit attraverso la [Tabella 3.3](#).

Durante la fase di conversione tra Signal Code e bit, nel momento in cui viene identificato il Codice 8 (21 volte 0) viene interpretato come End of Packet (EOP), avviene quindi la **memorizzazione del pacchetto in bits nell'apposito buffer**, differenziato per tipo di pacchetto (master, slave, config), in attesa che il Livello Link li svuoti.

### 3.3.2 Codifica

La codifica avviene in maniera opposta alla decodifica, il Livello Bit riceve dal Livello Link un buffer di caratteri ASCII da trasmettere. Ogni carattere ASCII (7 bits) viene convertito in bit, successivamente i bit vengono convertiti in **Signal Code** attraverso la [Tabella 3.3](#). Infine i **Signal Code** vengono convertiti in frequenze attraverso la formula inversa

$$f = base + (code * step) = 1000 * (code * 400) \quad (3.6)$$

Dove **base** è la frequenza minima (1000Hz), **step** è la differenza tra una frequenza e l'altra (400Hz) e **code** è il Signal Code da convertire.

Il buffer di frequenze viene successivamente inviato al Livello Fisico per la trasmissione.

## 3.4 Livello Link

Il **Livello Link** riceve le sequenze ASCII dal Livello Bit e realizza l'incapsulamento logico dei messaggi. Le comunicazioni vengono suddivise in tre canali distinti: *config*, *master* e *slave*. Ogni pacchetto segue lo schema generale:

**ID:destinatario{OP}k{mittente}**

dove i campi rappresentano, rispettivamente, l'ID del destinatario, l'operazione da eseguire e l'ID del mittente.

I pacchetti ASCII in arrivo dal Livello Bit vengono inseriti in una coda FIFO di Input e letti dal Livello Link. Una volta estratti, vengono interpretati in base al canale di appartenenza e al comando ricevuto. Analogamente, quando il Livello Link deve inviare dati, i pacchetti vengono preparati ed inseriti nella coda FIFO di Output, in attesa di essere trasformati in frequenze dal Livello Bit.

### 3.4.1 Procedura di registrazione

Quando un nodo si collega per la prima volta, o ha perso il proprio ID, deve eseguire la registrazione sul canale *config*. In questa fase, un nodo senza ID (e non in modalità *hotspot*) invia una richiesta all'hotspot, generando un identificativo provvisorio di **3 cifre alfanumeriche**. La richiesta assume la forma:

`ID:0000{REQ}l{idGenerato}`

Il carattere l sostituisce la k perché il nodo non dispone ancora di un ID valido. L'hotspot, ricevuta la richiesta, assegna un ID definitivo di **4 cifre numeriche univoche** e risponde con:

`ID:idAssegnato{SET}k{0000}`

Il nodo registra l'ID in memoria non volatile e conferma con:

`ID:0000{OK}k{idAssegnato}`

### 3.4.2 Indirizzamento e instradamento

Ogni pacchetto ricevuto viene analizzato per verificare l'indirizzo di destinazione:

- Se l'ID non coincide con quello locale, il nodo entra in modalità *parrot*: memorizza il pacchetto nella coda FIFO di Output e lo ritrasmette, in attesa di un acknowledgment. In assenza di ACK, il pacchetto viene ritrasmesso sullo stesso canale (ad esempio, un messaggio ricevuto in frequenze *master* viene riemesso in frequenze *master* anche se il nodo non è un hotspot).
- Se invece l'ID corrisponde a quello del nodo o all'indirizzo broadcast 1111, il pacchetto viene elaborato in base al comando ricevuto. Se è richiesta una risposta, questa viene generata tramite gli *handle* interni del Link e inserita nella coda di Output *slave*, pronta per essere trasmessa in frequenze *slave*.

Per garantire ordine e separazione dei flussi, il Livello Link mantiene tre code circolari dedicate ai canali *master*, *slave* e *config*, ognuna delle quali gestisce il traffico relativo.

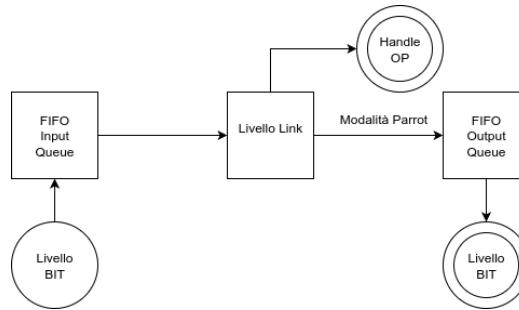


Figura 3.8: Schema del flusso dei pacchetti tra FIFO Input, Livello Link e FIFO Output

### 3.4.3 Gestione dei Token (TKN)

Un aspetto centrale riguarda la possibilità, per i nodi non-hotspot, di trasmettere i pacchetti contenuti nella propria coda di Output solo quando l'hotspot concede loro un *token* (operazione TKN).

Quando un nodo riceve un pacchetto TKN in frequenze *master*, acquisisce il diritto di trasmissione per un tempo massimo di **1 minuto**. Durante questo intervallo, il nodo può:

- inviare un pacchetto con operazione EXT per richiedere l'estensione del token. In tal caso l'hotspot risponde con un acknowledgment (OP:OK) e il tempo si prolunga di un ulteriore minuto;
- inviare un semplice OP:OK per confermare la gestione del token entro il minuto stabilito.

Se entro il tempo limite il nodo non invia alcun messaggio, il token decade automaticamente e l'hotspot lo assegna ad un altro dispositivo. È dunque responsabilità del nodo non-hotspot calcolare i tempi necessari per le proprie trasmissioni future e decidere se richiedere estensioni o accordarsi con l'hotspot per la durata del token.

Questa rappresenta attualmente la parte più delicata del protocollo: al momento della stesura il funzionamento è ancora limitato, ma in prospettiva si prevede la possibilità che anche i nodi non-hotspot possano generare autonomamente token, così da estendere la rete su più livelli e aumentare la flessibilità del sistema. Questa

apertura costituisce un’opportunità interessante, benché richieda un’implementazione più solida per evitare l’accavallamento di comunicazioni su canali multipli quando più nodi tentano di gestire i token in parallelo.

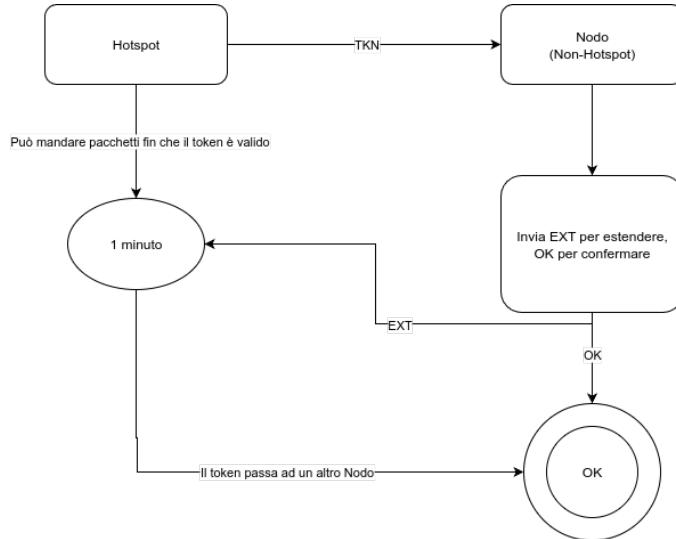


Figura 3.9: Schema del flusso dei Token tra hotspot e nodi non-hotspot

### 3.4.4 Comandi di controllo e di rete

Il Livello Link gestisce alcune operazioni speciali:

- **ABORT**: comando broadcast (ID: 1111) inviato dall’hotspot e ripetuto due volte in frequenze *master* da tutti i nodi non-hotspot. Non richiede ACK e serve a interrompere le operazioni, svuotando i buffer di Input e Output. Viene tipicamente generato quando la piattaforma *Blynk.io* invia il comando *abort* all’hotspot.
- **OK**: rappresenta l’ACK e viene inviato dai nodi non-hotspot all’hotspot per confermare la ricezione di un pacchetto. Solo in questo momento l’hotspot può eliminare il pacchetto dalla sua coda di Output *master*.

Infine, il Livello Link supporta un comando di rete dedicato, **conns**, che la piattaforma *Blynk.io* invia all’hotspot per ottenere la lista dei nodi attualmente connessi.

### 3.4.5 Indirizzi di rete

Gli indirizzi usati dal protocollo sono riassunti nella seguente tabella.

Indirizzo	Descrizione
0000	ID dell'hotspot
1111	Indirizzo broadcast (tutti i nodi)
0001 – 1110	Range di indirizzi assegnabili ai nodi

Tabella 3.4: Indirizzi di rete del Livello Link

### 3.5 Livello Applicazione

Dato che si conoscono già gli usi futuri di questo protocollo e non è destinato al grande pubblico, il dispositivo *hotspot* che riceve in ingresso dal loop **Blynk** un comando nella forma

Comando -> IdDestinatario

si occupa, con un codice "saldato" al livello Link, sia dell'incapsulamento dell'identificativo del destinatario nel formato ID:IdDestinatario, sia della traduzione semantica attraverso la seguente tabella.

Comando	Operazione	Non-Hotspot	Hotspot
movement_sensor_on	MOVEMENT_ON	Esegue misurazione PIR per 5s; restituisce MOVEMENT_YES o MOVEMENT_NO	ACK->OP:OK
movement_sensor_on_[tempo_in_ms]	MOVEMENT_ON_[tempo_in_ms]	Esegue misurazione PIR per l'intervallo specificato; restituisce MOVEMENT_YES o MOVEMENT_NO	ACK->OP:OK

Tabella 3.5: Mappatura comandi Blynk, operazioni interne e azioni eseguite dai nodi

Questa tabella mostra come l'hotspot agisca da traduttore e compressore semantico: ogni comando testuale viene ridotto a un codice operativo interno e ogni dispositivo non-hotspot può eseguire le relative funzioni senza dover gestire sintassi testuali complesse.

Il ritorno dell'esecuzione è anch'esso standardizzato (nel caso illustrato, MOVEMENT\_YES / MOVEMENT\_NO), mentre l'hotspot conferma localmente con un messaggio di tipo ACK->OP:OK, garantendo così sia l'affidabilità della comunicazione che la compatibilità futura con ulteriori estensioni già previste.

Quindi l'utente finale che sia l'industria mineraria, soccorritori o tecnici possono digitare un comando tramite l'interfaccia **Blynk.io**, corrisponde ad un'operazione più compatta che viene effettivamente interpretata dai nodi della rete.

La compressione avviene quindi su due livelli: non solo a livello di codifica dei bit, ma anche a livello semantico, poiché i comandi accettati non appartengono a categorie generiche di protocolli di uso comune (web come HTML/PHP, o trasferimento file come PNG/JPG nei protocolli FTP/HTTP), bensì ad un insieme chiuso e

proprietario di istruzioni standardizzate.

In questo modo l'interpretazione da parte dei dispositivi risulta rapida e priva di ambiguità, mentre l'interfaccia utente mantiene la leggibilità e la semplicità d'uso.

```
<_____  
| HotSpot Device ON |  
\_____|  
Segnale ricevuto: {REO}\{860  
Invio: ID:860(SET:0001)\{0000  
Segnale ricevuto: ID:0000\{OK\}\{0001  
Invio: Nuovo dispositivo registrato con successo, ID: 0001  
> conns  
< 0001  
> h  
< Comando non riconosciuto digita help per tutti i comandi  
> help  
< Comandi disponibili:  
CONNs: lista dispositivi connessi  
ABORT: interrompe la trasmissione corrente  
movement_sensor_on[_durata_ms]->ID: attiva il sensore di movimento (default 5000 ms)  
REQ->ID, SET->ID, OK->ID, MOVEMENT->ID, EXT->ID: invia il comando indicato al dispositivo  
HELP: mostra questo messaggio  
> movement_sensor_on>0001  
< Invio: ID:0001(MOVEMENT:ON)\{0000  
Segnale ricevuto: ID:0000\{OK\}\{0001  
Invio: ID:0001\{OK\}\{0000  
Risposta movimento da 0001: YES  
Segnale ricevuto: ID:0000\{MOVEMENT:YES\}\{0001  
Invio: ID:0001\{OK\}\{0000  
Risposta movimento da 0001: YES  
> abort  
< Invio: ID:1111\{ABORT\}\{0000  
Invio: ID:1111\{ABORT\}\{0000  
> movement_sensor_on_20000->0001  
< Invio: ID:0001(MOVEMENT:ON)\{0000  
Invio: ID:0001\{OK\}\{0000  
Segnale ricevuto: ID:0000\{MOVEMENT:YES\}\{0001  
Invio: ID:0001\{OK\}\{0000  
Risposta movimento da 0001: YES  
> abort  
< Invio: ID:1111\{ABORT\}\{0000  
Invio: ID:1111\{ABORT\}\{0000
```

Figura 3.10: Utilizzo del Livello Applicazione tramite l'interfaccia Blynk.io

# Capitolo 4

## Implementazione Hardware

### 4.1 Tecnologie di base

In questo capitolo tratteremo tutte le tecnologie utilizzate nel progetto, come i microcontrollori e i dispositivi di output, concentrando ci sull'ESP32.

#### 4.1.1 Panoramica dell'ESP32

##### 4.1.1.1 Generalità

I microcontrollori (MCU) sono circuiti integrati che combinano nello stesso chip un microprocessore con una serie di componenti periferici come: memoria, timer, convertitori analogico-digitali e soprattutto pin di I/O.

A differenza di un microprocessore (CPU), i microcontrollori non sono pensati per svolgere onerose attività di calcolo ma per interagire il più possibile con l'ambiente esterno, gestendo numerosi scambi fra le periferiche di Input/Output.

L'ESP32, sviluppato da Espressif Systems, rappresenta un microcontrollore avanzato con architettura dual-core Tensilica LX6 a 32 bit (alcune varianti hanno anche core single-core o LX7), integrando al suo interno Wi-Fi e Bluetooth a basso consumo. Questa caratteristica lo rende particolarmente adatto per applicazioni IoT (Internet of Things), domotica e sistemi embedded complessi (Espressif, 2024 [23]).

##### 4.1.1.2 Famiglie di ESP32

I microcontrollori sono suddivisi in famiglie a seconda del processore e delle periferiche integrate. Nel caso dell'ESP32, troviamo diverse varianti come ESP32-WROOM-32, ESP32-WROVER, ESP32-C3, ESP32-S2 ed ESP32-S3. Queste versioni condividono la stessa architettura di base ma differiscono per risorse

di memoria, numero di GPIO disponibili, supporto a periferiche aggiuntive o funzionalità avanzate come acceleratori AI.

La scelta del modulo da utilizzare deve essere fatta non sulla base delle prestazioni massime, ma dell'ottimalità rispetto al progetto in questione (Patti, 2024 [24]).

#### 4.1.1.3 Componenti hardware

Ogni ESP32 integra al suo interno non solo la CPU, ma anche le memorie e una grande quantità di periferiche digitali e analogiche, oltre ai moduli di comunicazione wireless (Wi-Fi 802.11 b/g/n e Bluetooth 4.2/5.0). Questa elevata integrazione consente di ridurre i costi, semplificare il design e garantire maggiore velocità di accesso ai dati.

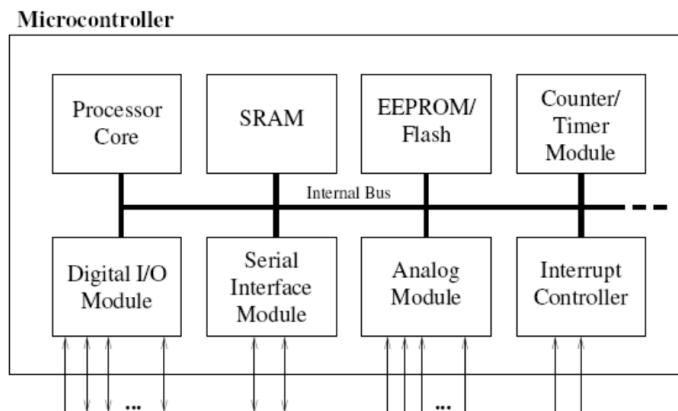


Figura 4.1: Componenti hardware principali di un microcontrollore ESP32.

#### 4.1.1.4 Memorie

Le memorie di un microcontrollore possono essere di tipo volatile e non volatile. Le volatili includono SRAM e DRAM; le non volatili comprendono ROM, EEPROM, Flash e, in alcuni casi, NVRAM.

L'ESP32 tipicamente dispone di:

- 448 KB di ROM, che contiene il bootloader e alcune librerie di base;
- 520 KB di SRAM interna, divisa tra cache, stack e heap per le applicazioni;
- Flash esterna fino a 16 MB (a seconda del modulo), utilizzata per memorizzare codice e file system;

- opzionalmente, PSRAM esterna fino a 8 MB, utile per applicazioni che richiedono elaborazioni multimediali o server web embedded.

Rispetto ad altre schede, l'ESP32 si distingue per la disponibilità di memoria e per la presenza di connettività wireless integrata, rendendolo una soluzione più completa per progetti IoT e sistemi embedded avanzati (Patti, 2024 [24]; Espressif, 2024 [23]).

### 4.1.2 Periferiche di Input/Output

Le periferiche di Input/Output (I/O) sono componenti essenziali che permettono al microcontrollore di interagire con il mondo esterno. Per la realizzazione del progetto sono state utilizzate diverse periferiche, in questa sezione se ne approfondiscono le loro caratteristiche.

Tutta la componentistica impiegata in questo progetto è stata selezionata dal catalogo **Adafruit Industries**. La scelta è motivata dall'elevata qualità dei loro moduli, dalla cura del layout dei circuiti stampati e dalla ricca documentazione tecnica a corredo, che garantiscono affidabilità in fase di prototipazione e semplicità di integrazione. L'utilizzo di componenti a marchio Adafruit riduce i rischi di incompatibilità o malfunzionamenti tipici dei cloni economici e assicura un supporto didattico di livello professionale.

#### 4.1.2.1 Microfono MEMS I2S digitale Adafruit – SPH0645LM4H

Il breakout Adafruit (cod. 3421) monta il microfono MEMS digitale **Knowles SPH0645LM4H-B**, bottom-port, con uscita I<sup>2</sup>S [25, 26]. È un dispositivo *slave* che richiede dal master le linee di clock (BCLK, LRCLK) [27]. Fornisce direttamente campioni PCM a 24 bit (con 18 bit effettivi), evitando stadi analogici esterni [28].



Figura 4.2: Microfono MEMS I2S digitale Adafruit SPH0645LM4H-B.

## Caratteristiche principali.

- Alimentazione: 1.62 V a 3.6 V, logica 3.3 V, tipico consumo 600  $\mu$ A [29].
- Interfaccia: I<sup>2</sup>S, pin BCLK, LRCLK, DOUT, SEL, più 3V e GND.
- Formato dati: parole a 24 bit (MSB first, complemento a due), 18 bit utili, oversampling ratio (OSR) = 64 [28].
- Prestazioni: SNR tip. 65 dB(A), AOP 120 dB SPL, sensibilità  $\sim -26$  dBFS @ 94 dB SPL [28].

**Pinout e collegamenti.** Due moduli possono essere usati in stereo condividendo BCLK, LRCLK, DOUT; SEL basso/alto definisce L/R. In modalità mono basta un solo modulo con SEL a GND (Left di default) [25].

**Interfaccia I<sup>2</sup>S.** Il microfono trasmette in formato *Philips*: ogni semiperiodo di LRCLK contiene una parola da 24 bit. La relazione fondamentale è  $BCLK = 64f_s$  [27]. Tipici: 3.072 MHz per 48 kHz, 2.8224 MHz per 44.1 kHz.

## Catena del segnale interna.

1. Trasduttore MEMS capacitivo: diaframma + backplate  $\rightarrow$  capacità  $C(t)$ .
2. Amplificatore a carica: traduce  $\Delta C$  in tensione.
3. Modulatore  $\Sigma\Delta$ : noise-shaping fuori banda audio.
4. Filtro di decimazione: riduce a  $f_s$  mantenendo 18 bit effettivi.
5. Serializer: prepara le parole I<sup>2</sup>S.

**Fondamenti fisici.** La pressione  $p(t)$  agisce sul diaframma (massa  $m$ , molla  $k$ , smorzamento  $b$ ) [30]:

$$m\ddot{x} + b\dot{x} + kx = S p(t)$$

La variazione di capacità è  $\Delta C/C_0 \approx x/d_0$ , con risonanza a  $\omega_0 = \sqrt{k/m}$ . Il rumore complessivo deriva da contributi termico-meccanici, elettronici e quantizzazione. Il back-volume e la porta acustica modellano la risposta in frequenza (roll-off alle basse, ondulazioni agli alti) [30].

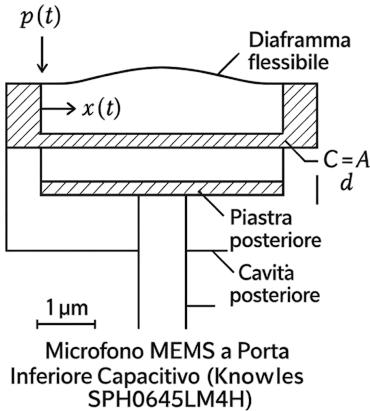


Figura 4.3: Sezione verticale del microfono Adafruit MEMS.

**Limiti.** Non tollera 5 V, banda garantita  $\sim 20\text{ Hz} - 10\text{ kHz}$  (utilizzabile fino a  $\sim 15\text{ kHz}$  secondo Adafruit) [26], OSR fisso (meno flessibile di PDM).

**Relazione dBFS-SPL.**  $94\text{ dB SPL} = 1\text{ Pa} \Rightarrow \approx -26\text{ dBFS RMS}$  [28]. L'AOP a 120 dB SPL ( $\sim 20\text{ Pa}$ ) indica il massimo senza distorsione significativa.

#### 4.1.2.2 Adafruit MAX98357A I<sup>2</sup>S Amplificatore Classe-D

Il breakout Adafruit basato sul chip **MAX98357A** è un amplificatore audio digitale in classe D, in grado di convertire direttamente un flusso I<sup>2</sup>S in segnale audio analogico amplificato per pilotare altoparlanti a bassa impedenza. La caratteristica principale è l'eliminazione della necessità di un DAC esterno: il MAX98357A integra al suo interno la conversione digitale-analogica, il filtraggio, e la sezione di potenza [31, 32].

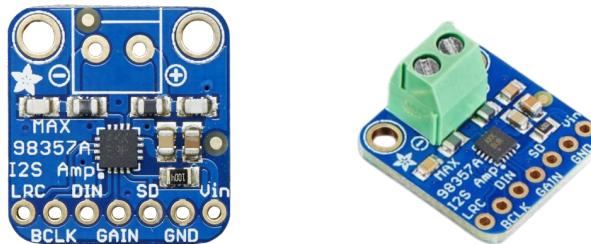


Figura 4.4: Amplificatore digitale Classe-D Adafruit MAX98357A.

#### Caratteristiche essenziali.

- **Alimentazione:** 2.5 V a 5.5 V, con tensione tipica 3.3 V o 5 V.
- **Potenza d'uscita:** fino a 3.2 W su  $4\Omega$  a 5 V, THD+N  $\approx 1\%$ .
- **Interfaccia:** I<sup>2</sup>S standard (BCLK, LRCLK, DIN); pin aggiuntivi per modalità (GAIN, SD, DMP).
- **Efficienza:** superiore all'85 % grazie all'architettura in classe D.
- **Gamma dinamica:**  $\approx 98$  dB, con rumore di fondo contenuto [31].

**Funzionamento.** Il MAX98357A riceve un flusso I<sup>2</sup>S in formato PCM (16, 24 o 32 bit, standard Philips). L'architettura interna comprende:

1. **Interfaccia I<sup>2</sup>S:** decodifica il flusso digitale e gestisce il framing (left/right).
2. **DAC sigma-delta:** converte i campioni PCM in una modulazione a densità di impulsi.
3. **Driver in classe D:** due MOSFET push-pull pilotano direttamente l'altoparlante in configurazione *bridge-tied load (BTL)*.
4. **Filtro LC virtuale:** la stessa induttanza del carico acustico e la risposta del driver fungono da filtro passa-basso, eliminando la componente PWM ad alta frequenza.

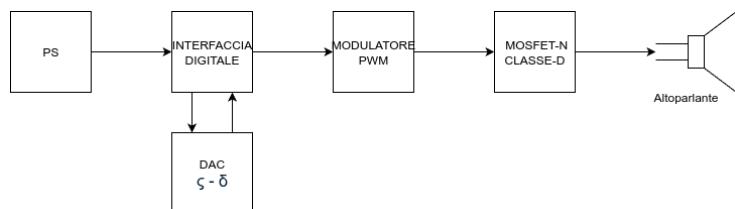


Figura 4.5: Funzionamento interno del MAX98357A.

**Motivazioni fisiche del funzionamento.** L'amplificatore in classe D si basa su modulazione a *duty cycle* dei MOSFET: il segnale analogico ricostruito corrisponde al valore medio del PWM. L'elevata efficienza deriva dal fatto che i transistor lavorano quasi sempre in saturazione o interdizione, riducendo le perdite per dissipazione. L'integrazione del DAC sigma-delta consente un noise-shaping che sposta il rumore di quantizzazione fuori banda audio, garantendo un'ottima qualità percepita [33].

## 4.2 Progettazione digitale

Per la progettazione digitale del sistema è stato utilizzato il software **Fritzing**, un tool open-source per la progettazione di circuiti elettronici, con la possibilità di creare moduli personalizzati.

Nel [Capitolo 3](#) è stato descritto il protocollo di comunicazione, facendo riferimento al **Livello Fisico**, come il livello più basso del modello. L'interazione I/O con il mondo circostante che avviene nel Livello Fisico, è realizzata con due componenti: una software, che tratteremo a breve nel [Capitolo 5](#), e una hardware, che vedremo in questo capitolo.

Il Livello Fisico è realizzato con due periferiche di Input/Output, il microfono MEMS digitale e l'amplificatore Classe-D, che comunicano con il microcontrollore ESP32 tramite il protocollo I<sup>2</sup>S.

Per la realizzazione del circuito elettronico sono stati utilizzati due breakout board Adafruit, che integrano i componenti necessari per il corretto funzionamento del microfono e dell'amplificatore.

L'utilizzo del microfono MEMS I2S digitale Adafruit SPH0645LM4H viene da una selezione accurata, basata su test di qualità, il progetto, che all'inizio prevedeva un microfono analogico, è stato modificato per utilizzare un microfono digitale, in quanto i test effettuati con il microfono analogico non hanno dato risultati soddisfacenti.

### 4.2.1 Collegamento del microfono digitale I<sup>2</sup>S

La [Figura 4.6](#) mostra lo schema di cablaggio realizzato in **Fritzing** tra la breakout Adafruit basata sul microfono MEMS digitale **SPH0645LM4H** e la **ESP32**, mediante il bus **I<sup>2</sup>S** (Inter-IC Sound). L'interfaccia prevede tre linee digitali di segnale (BCLK, LRCLK, DOUT) e due linee di alimentazione (3V3, GND), oltre al pin SEL per la selezione del canale logico (sinistro/destro).

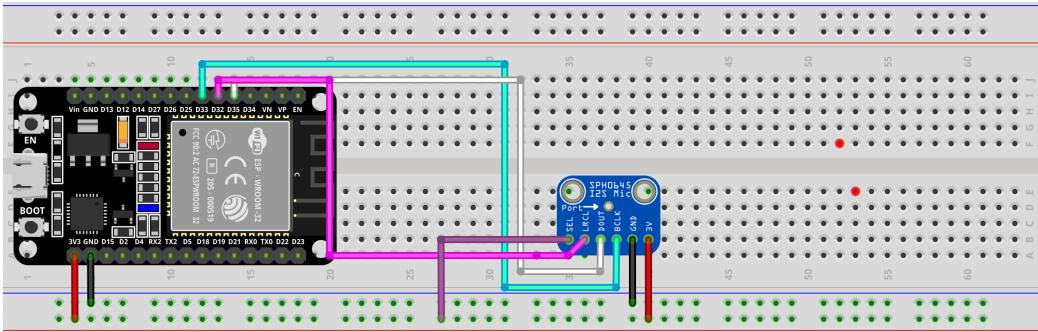


Figura 4.6: Collegamento fisico tra ESP32 e microfono I<sup>2</sup>S (Adafruit SPH0645LM4H) su breadboard.

**Pinout del modulo microfonico.** Sulla scheda Adafruit, da sinistra a destra (come serigrafato) sono presenti i pin: SEL, LRCL (o LRCLK/WS), DOUT (o SD), BCLK (o SCK), GND, 3V. DOUT è un’uscita a logica 3.3 V; LRCLK e BCLK sono ingressi di clock; SEL è un ingresso statico per selezionare su quale slot (sinistro o destro) il microfono presenta i campioni sul bus I<sup>2</sup>S.

**Mappatura dei segnali sull’ESP32.** La sezione 4.2.1 riassume la corrispondenza dei pin usati, come nella figura. Le scelte rispettano i vincoli dell’ESP32 (evitando pin con funzioni di strapping all’avvio come GPIO15-12 o limitazioni di sola-lettura).

Pin modulo	Segnale	Pin ESP32 (sigla serigrafata)
3V	Alimentazione 3.3 V	3V3
GND	Riferimento di massa	GND
BCLK	Bit Clock I <sup>2</sup> S	GPIO14 (D14)
LRCL	Word-Select / LRCLK	GPIO25 (D25)
DOUT	Serial Data (uscita microfono)	GPIO32 (D32)
SEL	Selezione canale	GND → canale sinistro

Tabella 4.1: Mappatura hardware dei pin I<sup>2</sup>S tra ESP32 e microfono SPH0645LM4H.

Il segnale SEL determina in quale slot vengono emessi i campioni audio: se portato a massa (SEL = GND) i dati escono sul canale sinistro, mentre a livello alto (SEL = 3V3) sul destro.

Nel nostro schema è stato fissato a 3V3, quindi l’uscita è forzata a destra. Questo a seguito di alcuni test dove si è verificato che per qualche difformità nel IC il canale

destro garantiva maggiore precisione.

**Dettagli elettrici e motivazioni delle scelte.** Il microfono Adafruit SPH0645LM4H funziona solo a 3.3 V: non sono previsti level shifter e un'alimentazione a 5 V lo danneggierebbe [29]. L'assorbimento è dell'ordine dei mA, quindi l'uscita 3V3 dell'ESP32 è più che sufficiente [29].

Per l'ESP32 si è scelto **GPIO14** come **BCLK**, **GPIO25** come **LRCLK** e **GPIO32** per il dato in ingresso.

Infine, il microfono trasmette campioni a 24 bit allineati su frame da 32 come discusso precedentemente nella [sottosezione 4.1.2.1](#).

#### 4.2.2 Collegamento dell'amplificatore digitale I<sup>2</sup>S

La [Figura 4.7](#) mostra lo schema di cablaggio realizzato in **Fritzing** tra la breakout Adafruit basata sul chip **MAX98357A** (amplificatore in classe D con interfaccia I<sup>2</sup>S) e la **ESP32**. L'interfaccia prevede tre linee digitali di segnale (BCLK, LRCLK, DIN) e due linee di alimentazione (3V3, GND). Il modulo espone inoltre i pin di configurazione **GAIN** e **SD** (shutdown), che in questo caso non vengono utilizzati e restano flottanti.

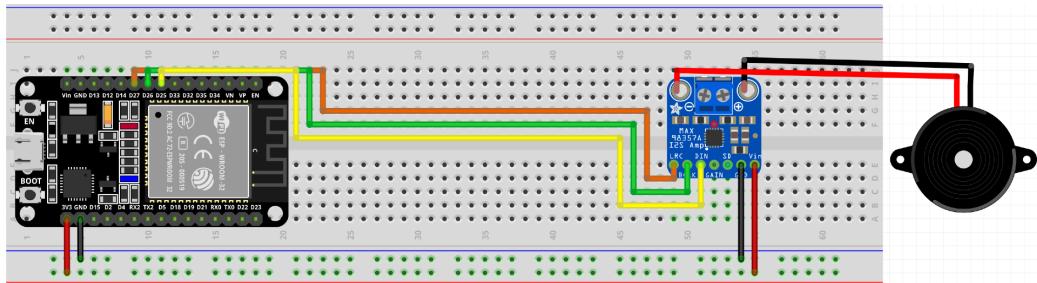


Figura 4.7: Collegamento fisico tra ESP32 e amplificatore I<sup>2</sup>S (Adafruit MAX98357A) su breadboard.

**Pinout del modulo amplificatore.** Sulla scheda Adafruit, da sinistra a destra (come serigrafato) sono presenti i pin: GAIN, SD, DIN, BCLK, LRCLK, GND, Vin. Il pin **DIN** è l'ingresso dati audio seriali (logica 3.3 V); **BCLK** e **LRCLK** sono segnali di clock forniti dall'ESP32; **Vin** riceve l'alimentazione (3–5 V, compatibile col regolatore onboard). I pin **GAIN** e **SD** sono opzionali: **GAIN** permette di configurare il guadagno interno tramite resistenze, mentre **SD** forza il chip in modalità risparmio energetico.

**Mappatura dei segnali sull'ESP32.** La sezione 4.2.2 riassume la corrispondenza dei pin usati, come nello schema. Anche in questo caso la scelta dei GPIO tiene conto delle raccomandazioni di Espressif, evitando pin con funzioni critiche all'avvio.

Pin modulo	Segnale	Pin ESP32 (sigla serigrafata)
Vin	Alimentazione 3.3 V / 5 V	3V3
GND	Riferimento di massa	GND
BCLK	Bit Clock I <sup>2</sup> S	GPIO14 (D14)
LRCLK	Word-Select / LRCLK	GPIO25 (D25)
DIN	Serial Data (ingresso audio)	GPIO26 (D26)
SD	Shutdown (non connesso)	–
GAIN	Configurazione guadagno (non connesso)	–

Tabella 4.2: Mappatura hardware dei pin I<sup>2</sup>S tra ESP32 e amplificatore MAX98357A.

Il collegamento all'altoparlante avviene direttamente dai terminali di uscita del modulo (OUT+, OUT-), come mostrato in figura. Essendo un amplificatore in classe D capace di pilotare piccoli altoparlanti (4–8 Ω, fino a circa 3 W a 5 V), non è necessaria ulteriore elettronica per il collegamento all'altoparlante.

**Dettagli elettrici e motivazioni delle scelte.** Il MAX98357A accetta alimentazione sia a 3.3 V che a 5 V [31]: in questo schema si è scelto di usare il **3V3 dell'ESP32** per semplicità, tenendo conto che ciò limita la potenza massima erogabile all'altoparlante e considerando che in una reale applicazione a scopo industriale si utilizzerebbero amplificatori da 230 V. Il consumo è nell'ordine di poche decine di mA a volume medio, quindi l'uscita 3V3 dell'ESP32 è sufficiente in prototipazione.

### 4.2.3 Sistema di segnalazione LED

La Figura 4.8 mostra lo schema di cablaggio realizzato in **Fritzing** tra la scheda **ESP32** e tre LED collegati su breadboard. Il sistema fornisce un feedback visivo sullo stato di decodifica del segnale audio, a supporto del protocollo discusso nella ??.

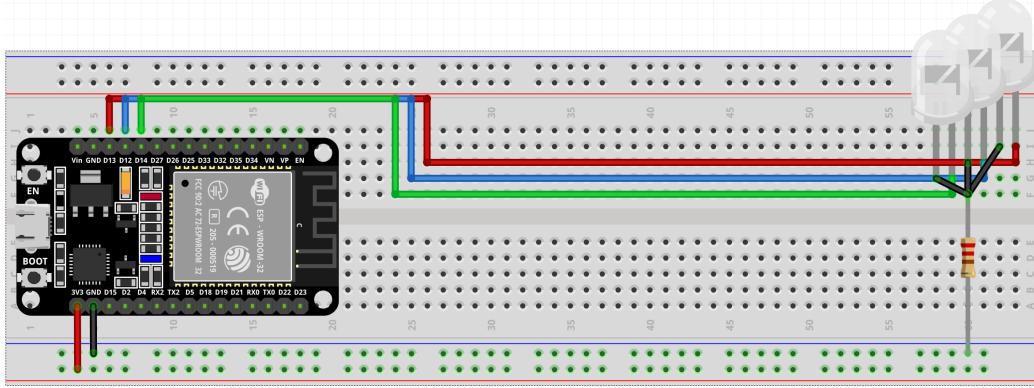


Figura 4.8: Collegamento fisico tra ESP32 e LED di segnalazione su breadboard.

**Funzionamento del sistema.** Sono stati utilizzati tre LED di colore diverso (verde, blu e rosso) per rappresentare in modo immediato lo stato di elaborazione:

- LED verde: il segnale è stato decodificato correttamente;
- LED blu: il segnale è in fase di decodifica;
- LED rosso: il segnale non ha superato il controllo dell'interpolazione parabolica e non è stato decodificato correttamente.

**Mappatura dei segnali sull'ESP32.** La [sezione 4.2.3](#) riassume la corrispondenza dei pin usati per il pilotaggio dei tre LED, come nello schema in figura. Ogni LED è collegato in serie a una resistenza di limitazione, necessaria a ridurre la corrente entro i limiti di sicurezza per i GPIO dell'ESP32 (massimo 12 mA consigliati).

LED	Funzione	Pin ESP32 (sigla serigrafata)
Verde	Decodifica corretta	GPIO14 (D14)
Blu	Decodifica in corso	GPIO27 (D27)
Rosso	Errore di decodifica	GPIO26 (D26)

Tabella 4.3: Mappatura hardware dei pin di uscita dell'ESP32 verso i LED di stato.

**Dettagli elettrici e motivazioni delle scelte.** Ciascun LED è collegato al rispettivo GPIO tramite una resistenza da  $220\ \Omega$ , sufficiente a limitare la corrente senza ridurre eccessivamente la luminosità. I pin **GPIO14**, **GPIO27** e **GPIO26** sono stati scelti in quanto liberi da vincoli di boot e comunemente usati per uscite digitali.

#### 4.2.4 Bottone di Hotspot

La [Figura 4.9](#) mostra il collegamento su breadboard di un pulsante e due LED all'ESP32. Il pulsante è collegato al pin **GPIO34 (D34)** e consente di abilitare la modalità *hotspot* prevista dal protocollo ([sezione 3.4](#)).

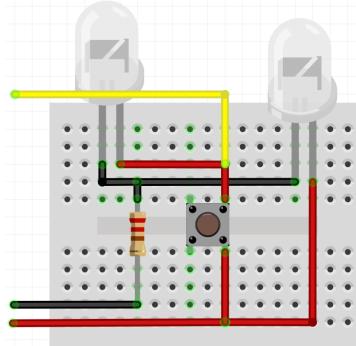


Figura 4.9: Schema di collegamento del bottone e dei LED di stato per la modalità hotspot.

Il LED giallo, a sinistra, indica l'attivazione dell'hotspot e si accende solo alla pressione del pulsante. Il LED a destra segnala invece lo stato di alimentazione della scheda, acceso quando l'ESP32 è in funzione e spento in assenza di alimentazione.

#### 4.2.5 Circuito completo

La [Figura 4.10](#) riassume l'intero sistema realizzato su breadboard, integrando in un unico schema tutte le funzionalità descritte nelle sezioni precedenti. L'**ESP32**, alimentato da un pacco batterie AAA, costituisce il cuore del circuito e gestisce sia la parte di acquisizione sia quella di elaborazione e segnalazione.

Il microfono **I<sup>2</sup>S** raccoglie il segnale audio e lo invia al microcontrollore, che a sua volta lo riproduce attraverso l'amplificatore digitale **MAX98357A** collegato a un piccolo altoparlante.

In parallelo, il sistema integra un'interfaccia visiva con tre LED che indicano lo stato della decodifica, insieme a un pulsante dedicato all'attivazione della modalità *hotspot*, il cui stato è reso evidente da un ulteriore LED giallo.

Tutti i collegamenti sono realizzati su breadboard, con le linee di alimentazione distribuite ai vari moduli e i segnali instradati verso i pin dell'ESP32, tra cui il **GPIO34**, correttamente utilizzato come ingresso per il pulsante. In questo modo il circuito offre un esempio completo di integrazione tra acquisizione, elaborazione,

riproduzione e interfaccia utente, consentendo il passaggio fluido dall’ascolto alla decodifica, fino alla segnalazione visiva e al controllo della connessione wireless.

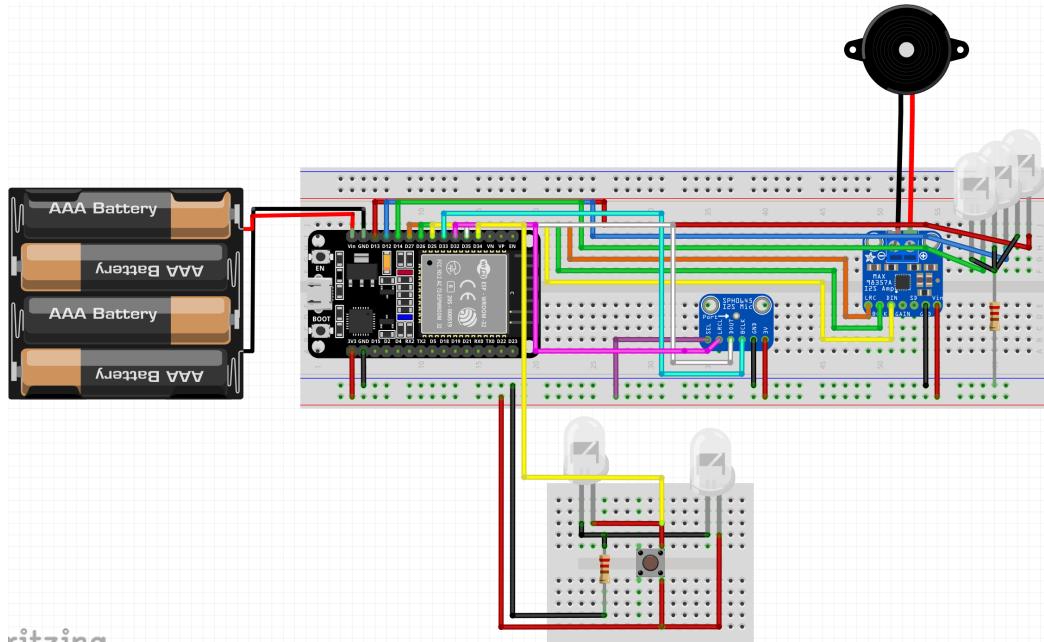


Figura 4.10: Schema del circuito completo con ESP32, microfono I<sup>2</sup>S, amplificatore MAX98357A, LED di stato e pulsante di hotspot.

#### 4.2.6 Realizzazione del PCB

Dopo aver completato lo schema su breadboard, il progetto è stato trasferito in **vista PCB** all’interno di Fritzing, con le dovute modifiche, ottenendo così una scheda stampata ordinata e compatta, come mostrato nella [Figura 4.11](#).

In questa configurazione l’**ESP32** rimane al centro della scheda, con le piste di segnale distribuite verso i moduli periferici: i LED di stato con le relative resistenze sono disposti lungo il bordo destro per facilitare la visibilità [LED-1-2-3], mentre il pulsante di hotspot è collocato nella parte inferiore [S2] insieme al LED [LED4] che ne segnala l’attivazione.

Il layout delle piste è stato ottimizzato per ridurre le interferenze e mantenere separati i percorsi di potenza da quelli di segnale.

Inoltre il PCB è stampato su un solo strato, questo ha richiesto un’attenta disposizione dei componenti e delle piste per evitare incroci, ma ha diversi vantaggi come la riduzione dei costi di produzione e la semplificazione del processo di fabbricazione.

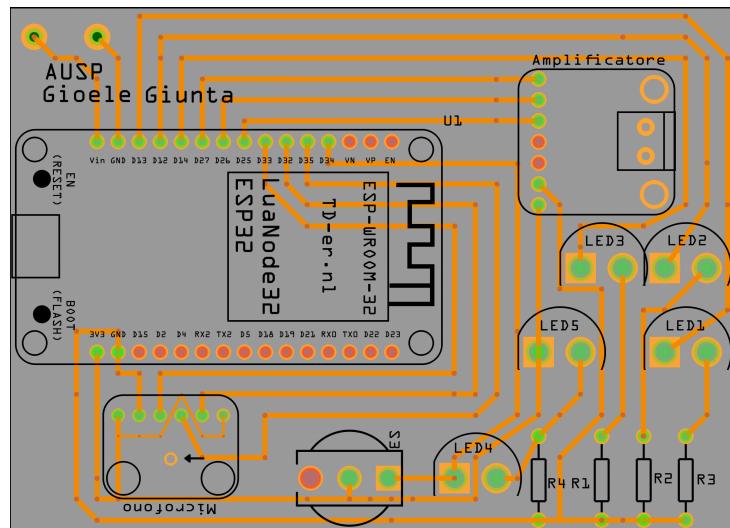


Figura 4.11: Layout del circuito stampato (PCB), con ESP32, LED di stato, pulsante di hotspot e connettori di alimentazione.

# Capitolo 5

## Implementazione Software: Design del Protocollo

### 5.1 Architettura generale del sistema

Il software è stato sviluppato in linguaggio C. Questa scelta permette di avere un controllo diretto sull'hardware dell'ESP32, ottenendo esecuzioni prevedibili ed un uso minuzioso delle risorse.

L'utilizzo del C ha inoltre garantito maggiore leggerezza rispetto al C++ e una buona portabilità, grazie alla disponibilità di librerie già consolidate in ambito firmware.

Alcune librerie Arduino originariamente scritte in C++ sono state adattate in C questo per permettere la compilazione del codice.

Allo stesso tempo, l'uso del framework Arduino su ESP32 ha offerto vantaggi importanti tra cui: documentazione estesa, supporto diretto alle periferiche del microcontrollore e una community ampia, che ha velocizzato la fase di sviluppo.

Per lo sviluppo è stato utilizzato *Visual Studio Code*, scelto per l'ampia disponibilità di estensioni. In particolare, l'integrazione con **PlatformIO** ha reso più semplice la gestione del progetto: compilazione, caricamento e debugging sono stati unificati in un unico ambiente, con gestione automatica delle librerie e delle dipendenze, ha permesso uno sviluppo più rapido ed organizzato.

L'architettura complessiva si basa su moduli separati e specializzati, ognuno responsabile di una parte precisa del protocollo. Questo approccio modulare ha garantito una migliore manutenibilità del codice e ha semplificato l'integrazione tra i diversi livelli funzionali descritti in [Capitolo 3](#).

## 5.2 Emissione dei toni

L'uscita audio discussa nella [sottosezione 3.2.2](#) è gestita dal modulo **audio\_driver**, responsabile dell'inizializzazione del bus I<sup>2</sup>S e della generazione in tempo reale delle sinusoidi che rappresentano i bit.

La funzione **play\_two\_tones** produce due toni simultanei, richiedendo come parametri le due frequenze delle quali dovrà generare le sinusoidi. La generazione del segnale avviene con persistenza di fase, evitando discontinuità tra un'emissione e la successiva.

```
1 void play_two_tones(int freq1, int freq2) {
2     if(freq1 == 0 && freq2 == 0){
3         delay(80);
4     } else {
5         const float tone_duration = 0.024f;
6         const int tone_samples = (int)(G_SAMPLE_RATE *
7             ↪ tone_duration);
8         const int tone_buffer_size = tone_samples * 2;
9         int16_t tone_buffer[tone_buffer_size];
```

Le variabili sotto garantiscono la continuità di fase tra chiamate successive della funzione:

```
1     static float phase1 = 0.0f;
2     static float phase2 = 0.0f;
3     const float inc1 = 2.0f * PI * freq1 / G_SAMPLE_RATE;
4     const float inc2 = 2.0f * PI * freq2 / G_SAMPLE_RATE;
```

Questo frammento di codice mostra la generazione del buffer di campioni, che viene poi scritto sul bus I<sup>2</sup>S:

```
1     for (int i = 0; i < tone_samples; i++) {
2         float mixed = sinf(phase1) + sinf(phase2);
3         int16_t sample = (int16_t)(3000 * (mixed /
4             ↪ 2.0f));
5         tone_buffer[2 * i] = sample;
6         tone_buffer[2 * i + 1] = sample;
7         phase1 += inc1;
8         if (phase1 >= 2.0f * PI) phase1 -= 2.0f * PI;
9         phase2 += inc2;
10        if (phase2 >= 2.0f * PI) phase2 -= 2.0f * PI;
11    }
```

```

11     size_t bytes_written = 0;
12     i2s_write(I2S_NUM, tone_buffer, sizeof(tone_buffer),
13                 → &bytes_written, portMAX_DELAY);
14 }

```

L'ampiezza massima del segnale è limitata a un valore inferiore a 32767 per evitare saturazioni sul DAC, questa conclusione deriva da test che hanno mostrato come il fattore di scaling a **3000** garantisce **headroom sufficiente**. Quando entrambe le frequenze sono nulle, la routine introduce una pausa di **80 ms**, che equivale a una “inattività” progettata per separare gruppi di simboli o pacchetti.

### 5.3 Campionamento e bufferizzazione

Il percorso inverso, ovvero l'acquisizione dei toni ricevuti, discusso nella [sottosezione 3.2.1](#) come l'ingresso del Livello Fisico è affidato al **reader**.

Il modulo crea due buffer circolari tra cui effettua un meccanismo di **swapping sincronizzato**: mentre uno viene riempito dal task di I<sup>2</sup>S, l'altro è disponibile per l'elaborazione FFT, questo avviene attraverso un insieme di variabili rese **extern** e mediante l'uso di un **puntatore** con attributo **extern** che punta sempre al buffer pronto, così da renderlo disponibile alla funzione richiedente senza che questa entri nella logica del reader.

```

1 extern volatile int data_ready;
2 extern volatile int status_flag;
3 extern complex_g3_t *array_ready;

```

La funzione **reader\_task** legge i campioni dal bus I<sup>2</sup>S, li converte in numeri complessi e li memorizza nel buffer corrente.

```

1 static void reader_task(void *param) {
2     size_t bytes_read;
3     int32_t dma_buffer[DMA_BUFFER_SIZE / 4];
4     static float dc_mean = 0.0f;
5     const float alpha = 1.0f / 1024.0f;
6     while (1) {
7         i2s_read(I2S_PORT, (void*)dma_buffer,
8                  → DMA_BUFFER_SIZE, &bytes_read, portMAX_DELAY);
9         int samples = bytes_read / 4;

```

```

9      for (int i = 0; i < samples; i++) {
10         int32_t s32 = dma_buffer[i] >> 8;
11         float x = (float)(s32 >> 12);
12         dc_mean += alpha * (x - dc_mean);
13         float val = x - dc_mean;
14         current_data[counter].re = (double)val;
15         current_data[counter].im = 0.0;
16         counter++;
17         if (counter >= ARRAY_ELEMENTS) {
18             data_ready = 1;
19             swap_array();
20             counter = 0;
21         }
22     }
23 }
24 }
```

L'eliminazione della componente DC avviene con una media mobile parametrizzata da **alpha**,

il quale viene; i campioni centrati vengono convertiti in numeri complessi con parte immaginaria nulla per l'elaborazione spettrale.

L'uso di **swap\_array** consente di non perdere alcun campione e di garantire un'elaborazione a flusso continuo.

```

1 static void swap_array(void) {
2     array_ready = current_data;
3     current_data = (current_data == main_array) ?
4         secondary_array : main_array;
5 }
```

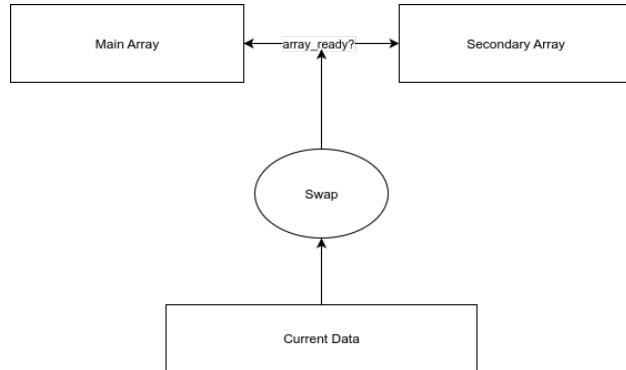


Figura 5.1: Swapping degli array di campioni

## 5.4 Trasformata veloce di Fourier

Altro aspetto critico del Livello Fisico è la trasformata rapida di Fourier (FFT), che converte i campioni temporali in rappresentazioni spettrali. Così come discusso nella [sezione 3.2.1](#), l'implementazione adottata è una versione personalizzata dell'algoritmo di Cooley–Tukey, ottimizzata per il contesto real-time e per la risoluzione richiesta.

L'analisi frequenziale viene eseguita nel modulo **fft.c**, questa include due funzioni principali **FFT\_get\_twiddle\_factors** e **FFT\_calculate**.

```

1 void FFT_calculate (complex_g3_t *x, long N, complex_g3_t *X,
2                      complex_g3_t *scratch, complex_g3_t
2                      ↵      *twiddles){
3     int k, m, n;
4     int skip;
5     boolean evenIteration = N & 0x55555555;
6     complex_g3_t* E;
7     complex_g3_t* Xp, *Xp2, *Xstart;
8     if (N == 1){
9         X[0] = x[0];
10        return;
11    }
12    E = x;
13    for (n = 1; n < N; n = n * 2){
14        Xstart = evenIteration ? scratch : X;
15        skip = N / (2 * n);
16        Xp = Xstart;

```

```

17     Xp2 = Xstart + N / 2;
18     for (k = 0; k < n; k++){
19         double tim = twiddles[k * skip].im;
20         double tre = twiddles[k * skip].re;
21         for (m = 0; m < skip; ++m){
22             complex_g3_t* D = E + skip;
23             double dre = D->re * tre - D->im * tim;
24             double dim = D->re * tim + D->im * tre;
25             Xp->re = E->re + dre;
26             Xp->im = E->im + dim;
27             Xp2->re = E->re - dre;
28             Xp2->im = E->im - dim;
29             ++Xp; ++Xp2; ++E;
30         }
31         E += skip;
32     }
33     E = Xstart;
34     evenIteration = !evenIteration;
35 }
36 }
```

Lo schema fa uso di array globali per l'uscita e per lo spazio temporaneo, così da evitare allocazioni dinamiche che in un contesto real-time sarebbero troppo costose.

Il flag **evenIteration** serve a gestire il ping-pong tra l'array risultato e quello di appoggio: in questo modo si alternano i ruoli dei due buffer a ogni passo, senza dover copiare dati inutilmente.

## 5.5 Decodifica spettrale

Le frequenze individuate dalla trasformata passano al modulo **decoder**, che ha il compito di capire se un picco spettrale corrisponde davvero a un tono del protocollo oppure se si tratta di rumore. Questa logica è implementata nella funzione **check\_active\_frequencies**, che lavora in tre fasi: calcolo della soglia dinamica, controllo dei massimi locali e interpolazione del picco confermato.

```

1 struct_interpolated_frequency check_active_frequencies(
2     complex_g3_t *data, int bin_1, int bin_2,
3     int id, double noise_floor){
4     int i, j;
```

```

5     struct_interpolated_frequency detected_freq;
6     detected_freq.work = 0;
7     detected_freq.frequency = -1.0;
8     detected_freq.estimated_amplitude = -1.0;
9     detected_freq.dynamic_amplitude_threshold = -1.0;

```

La funzione crea una struttura **detected\_freq** che conterrà il risultato. Tutti i campi sono inizializzati a valori “non validi” (ad esempio -1.0) così che, se nessuna frequenza viene trovata, il chiamante può accorgersene immediatamente.

```

14    for (j = bin_1; j <= bin_2; j++){
15        double freq = (double)(FS * j) / NN;
16        double amp = complex_magnitude(data[j]);

```

Il ciclo scorre i bin tra **bin\_1** e **bin\_2** che, come discusso nel [sottosottosezione 3.2.1.1](#) i bin sono la rappresentazione discreta della frequenza nello spettro ottenuto dalla FFT.

$$bin[i] = (FS * i) / NN = (48000 * i) / 512 = 93.75 * i. \quad (5.1)$$

La formula calcola la frequenza corrispondente a ogni bin, dove **FS** è la frequenza di campionamento e **NN** il numero di punti della FFT.

Il BIN è quindi un indice che rappresenta una specifica frequenza nell’array risultante dalla FFT.

La funzione **complex\_magnitude** calcola il modulo del numero complesso associato al bin corrente, che rappresenta l’ampiezza del segnale a quella frequenza.

```

18    if(G_LINEAR_REGRESSION_MODE == 0){
19        double dynamic_amplitude_threshold = noise_floor
20        ↪ * 8.0;
21        if (amp > dynamic_amplitude_threshold){
22            for(i = j-6; i < j + 6 &&
23                ↪ complex_magnitude(data[i]) <= amp; i++)
24                ↪ {};
25            if (i == j+6){
26                detected_freq =
27                    interpolate_peak_frequency(data, j,
28                    ↪ FS, NN);
29
30                ↪ detected_freq.dynamic_amplitude_threshold
31                ↪ = dynamic_amplitude_threshold;

```

```

25           detected_freq.work = 1;
26           return detected_freq;
27       }
28   }
```

La soglia dinamica (dynamic\_amplitude\_threshold) è otto volte il **noise\_floor**. Una volta superata la soglia, viene controllata una finestra di 12 bin, come rappresentato in [Figura 3.5](#) (sei a sinistra e sei a destra) per assicurarsi che il valore corrente sia davvero un massimo locale.

Solo in questo caso si invoca **interpolate\_peak\_frequency**, le quali motivazioni e funzionamento sono spiegate nel [sezione 3.2.1.1](#), che con una formula di interpolazione parabolica calcola la frequenza “vera”, per sopperire alla bassa risoluzione spettrale, questa tecnica, quindi fornisce una precisione sub-bin, cioè di una frequenza intermedia tra due bin adiacenti.

Infine, se nessuna frequenza valida è trovata, la funzione ritorna la struttura con i valori -1.0.

## 5.6 Traduzione frequenza-bit

Una volta isolati i picchi, i moduli riguardanti il Livello Fisico lasciano spazio ai **moduli del Livello Bit** il primo passo è tradurli in **Signal Codes** come nella seguente tabella. Questa operazione è svolta dal modulo **bit\_freq\_codec**, e in particolare dalla funzione **interpret\_bits**.

Frequenza	Signal Code (0)	Signal Code (1)
1400	(0)	
1800	(1)	
2200	(2)	
2600	(3)	
3000	(4)	
3400	(5)	
3800	(6)	
4200	(7)	
4600	(8)	
5000		(9)
5400		(10)
5800		(11)
6200		(12)
6600		(13)
7000		(14)
7400		(15)
7800		(16)
8200		(17)

Tabella 5.1: Mappatura frequenze con divisione dei Signal Code in 0 e 1

```

1 int interpret_bits(int freqs[3]){
2     int count = 0;
3     int active[3] = {0};
4     for (int i = 0; i < 3; ++i) {
5         if (freqs[i] > 0) {
6             active[i] = 1;
7             count++;
8         }
9     }

```

La funzione riceve in ingresso un array di tre frequenze, dove la prima posizione è riservata alla **frequenza dello zero**, la seconda alla **frequenza della portante** questa serve ad identificare il canale tra (master, slave ed config) più che alla creazione del Signal Code, e la terza alla **frequenza dell'uno**. Il contatore **count** tiene traccia del numero di toni attivi: se è diverso da due, oppure i toni ricevuti sono quelli esterni, quindi senza tono che indica il canale, allora il simbolo non è valido.

```

10     if (count == 2) {
11         if (active[0] && active[1] && !active[2]) {

```

```

12     if(freqs[0] < MASTER_BASE + (TONE_STEP*19) &&
13         freqs[1] == MASTER_BASE){
14         return (freqs[0]-MASTER_BASE)/(TONE_STEP);
15     }
16     if(freqs[0] < SLAVE_BASE + (TONE_STEP*19) &&
17         freqs[1] == SLAVE_CARRIER){
18         return (freqs[0]-SLAVE_BASE)/(TONE_STEP);
19     }
20     if(freqs[0] < CONFIG_BASE + (TONE_STEP*19) &&
21         freqs[1] == CONFIG_CARRIER){
22         return (freqs[0]-CONFIG_BASE)/(TONE_STEP);
23     }

```

Se ci sono due frequenze attive, si controlla quale delle tre posizioni è spenta e si determina il ruolo: **master**, **slave** o **config**.

Il calcolo restituisce un il **Signal Code** che rappresenta **quanti zeri o uno consecutivi** sono codificati in quel simbolo come è stato illustrato nella [Tabella 3.3](#).

```

23     if (!active[0] && active[1] && active[2]){
24         if(freqs[2] < MASTER_BASE + (TONE_STEP*19) &&
25             freqs[1] == MASTER_BASE){
26             return (freqs[2]-MASTER_BASE)/(TONE_STEP);
27         }
28         if(freqs[2] < SLAVE_BASE + (TONE_STEP*19) &&
29             freqs[1] == SLAVE_CARRIER){
30             return (freqs[2]-SLAVE_BASE)/(TONE_STEP);
31         }
32         if(freqs[2] < CONFIG_BASE + (TONE_STEP*19) &&
33             freqs[1] == CONFIG_CARRIER){
34             return (freqs[2]-CONFIG_BASE)/(TONE_STEP);
35         }
36         return -2;
37     } else if (count == 3) {
38         return -2;
39     } else {
40         return -3;
41     }

```

I valori negativi gestiscono gli errori: **-2** significa che la combinazione di frequenze non è valida, mentre **-3** indica che non ci sono abbastanza toni per interpretare un simbolo.

Infine, la conversione inversa è svolta da **frequency\_coder**, che genera le coppie di frequenze a partire da un **Signal Code** e dal ruolo del canale.

## 5.7 Ricevimento e trasmissione dei bit (Packers)

### 5.7.1 Ricezione dei bit

Terminata la decodifica, i **Signal Code** vengono accumulati in buffer a sette bit dal modulo **bit\_input\_packer**.

Un aspetto sofisticato di questo componente è la gestione del flush: questo aspetta l'arrivo di un Signal Code 8 che rappresenta 21 zero consecutivi, ma è condizionato da un timeout, utile quando i toni cessano di arrivare ma il Code 8 non è arrivato, a causa di qualche interferenza.

L'algoritmo controlla costantemente l'istante dell'ultimo bit per ogni canale, e se per più di un secondo non sopraggiungono nuovi simboli effettua la conversione in ASCII. Questa ne è l'implementazione:

```

1  static bool timeout_flush_if_needed(BitPacker* packer,
2                                     const char* label,
3                                     bool* timeout_armed,
4                                     uint64_t last_bit_ms,
5                                     bool
6                                     ↵ no_new_bit_this_tick){
7     if (!*timeout_armed) return false;
8     if (!no_new_bit_this_tick) return false;
9     uint64_t tnow = now_ms();
10    if ((tnow - last_bit_ms) < TIMEOUT_MS) return false;
11    bool ok = flush_and_convert_to_ascii(packer, label);
12    *timeout_armed = false;
13    return ok;
14 }
```

Quando il flush viene eseguito, la routine **flush\_and\_convert\_to\_ascii** trasferisce il contenuto della struttura in array di caratteri, inoltre ne verifica la qualità del pacchetto accertandosi che non ci siano caratteri non stampabili.

La funzione ritorna **true** se il pacchetto è valido, **false** altrimenti.

### 5.7.2 Trasmissione dei bit

L'inversa della fase di ricevimento dei bit, è la fase di trasmissione dei bit, ovvero l'impacchettamento dei bit in uscita in **Signal Code**, è svolto dal modulo **bit\_output\_packer**, che comprime i bit in **Signal Code** applicando una semplice forma di **run-length encoding** e replicando ogni codice più volte per incrementare la robustezza in presenza di rumore, così che venga trasmesso dal livello fisico per 3 volte consecutivamente, come illustrato in figura [Figura 3.7](#) ogni codice verrà trasmesso per un tempo di 24ms, quindi ritrasmettendo per 3 volte ogni codice, si avrà una finestra di ascolto di 72ms per ogni codice.

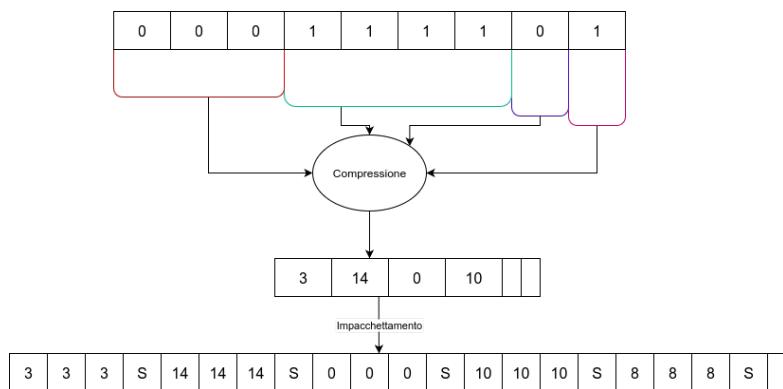


Figura 5.2: Compressione ed impacchettamento mediante Algoritmo Run-Length  
S= Silenzio

Nel seguente frammento è mostrata la parte conclusiva di **bit\_output\_packer\_convert** che aggiunge il Code terminatore 8 (21 zeri) ed il silenzio finale che verrà interpretato come un silenzio di 80ms dal modulo **emit\_tones**:

```

1 for(int b = 6; b >= 0; --b){
2     printf(" 8 ");
3     packer->pairs[packer->pair_count++] = frequency_coder(8,
4         → role);
5 }
5 packer->pairs[packer->pair_count++] = silent;
  
```

## 5.8 Instradamento e protocollo

Il livello superiore, il Livello Link, del protocollo è rappresentato da **protocol.c**, dove vengono gestite l'assegnazione degli identificativi, l'emissione di comandi e

la ricezione delle risposte.

I messaggi, incapsulati con la sintassi **ID:{dest}{OP}k{src}**, come discusso nella [sezione 3.4](#) ed nella [sottosezione 3.4.1](#) vengono convogliati all'uscita del canale appropriato tramite **send\_master**, **send\_slave** o **send\_config**.

Ogni funzione di trasmissione sfrutta **bit\_output\_packer** ed **emit\_tones**, ma si differenzia per la portante adottata. Il frammento seguente illustra la procedura di invio sul canale master:

```
1 static unsigned long send_master(const char *msg){  
2     log_send(msg);  
3     BitOutputPacker packer;  
4     bit_output_packer_init(&packer);  
5     unsigned long duration = 0;  
6     if(bit_output_packer_compress(&packer, msg)){  
7         if(bit_output_packer_convert(&packer, 0)){  
8             duration = estimate_send_duration(&packer);  
9             emit_tones(packer.pairs, packer.pair_count);  
10        }  
11    }  
12    bit_output_packer_free(&packer);  
13    return duration;  
14 }
```

La funzione calcola anche il tempo necessario alla trasmissione attraverso **estimate\_send\_duration**, valore impiegato per ritardare eventuali retransmission in caso di mancato ACK. All'interno della routine **protocol\_handle\_message** sono implementate le sequenze di registrazione, il meccanismo del token di trasmissione e la mappatura dei comandi provenienti dall'interfaccia esterna.

Quando un nodo privo di identificativo invia una richiesta **{REQ}l{xxx}**, l'hotspot genera un ID univoco e risponde con il messaggio **ID:idAssegnato{SET}k{0000}**; il nodo memorizza l'ID non volatile e conferma con **{OK}**. Le stesse routine gestiscono anche le operazioni **ABORT** e **MOVEMENT**, delegando al modulo **movement\_sensor** l'effettiva lettura del sensore PIR.

## 5.9 Sensore di movimento

Infine per mostrare come il protocollo possa essere esteso con funzionalità pratiche, è stato sviluppato il modulo **movement\_sensor**, che non appartiene al protocollo in senso stretto, ma è la prima applicazione del **Livello Applicativo**. La routine **movement\_sensor\_detect** attiva il sensore PIR per un intervallo prefissato e restituisce **true** o **false** a seconda della rilevazione. Alla ricezione del

comando **MOVEMENT:ON\_xxx**, il protocollo esegue la chiamata e risponde con **MOVEMENT:YES** o **MOVEMENT:NO**, questo per offrire all'utilizzatore del sistema un'astrazione sulla quale sfruttando le potezialità di un browser web, si può costruire un'interfaccia utente.

# Elenco delle figure

1.1	Corner Loss ~30 dB in un condotto con angolo di 90° per radio frequenze HF/SHF [2].....	4
2.1	Mappa Tunnel U.S. National Institute for Occupational Safety and Health's [2]. .....	7
3.1	Livelli del Protocollo.....	10
3.2	Verifica della cattura del tono a 9kHz con campionamento a 48kHz nel periodo 0-0.010 secondi, utilizzando finestra di Hann su 512 elementi.....	11
3.3	Grafico contenente lo spettro di frequenza calcolato tramite FFT ..	12
3.4	Grafico contenente lo spettro di frequenza di tre frequenze (bassa, media, alta), utilizzato per identificare l'attenuazione in frequenza del microfono. ....	14
3.5	Picco Locale nell'intorno dei sei Bins .....	15
3.6	Interpolazione Parabolica [22].....	16
3.7	Finestra d'ascolto della FFT sul segnale emesso .....	17
3.8	Schema del flusso dei pacchetti tra FIFO Input, Livello Link e FIFO Output .....	22
3.9	Schema del flusso dei Token tra hotspot e nodi non-hotspot .....	23
3.10	Utilizzo del Livello Applicazione tramite l'interfaccia Blynk.io .....	25
4.1	Componenti hardware principali di un microcontrollore ESP32.....	27
4.2	Microfono MEMS I2S digitale Adafruit SPH0645LM4H-B.....	28
4.3	Sezione verticale del microfono Adafruit MEMS. .....	30
4.4	Amplificatore digitale Classe-D Adafruit MAX98357A. ....	30
4.5	Funzionamento interno del MAX98357A.....	31
4.6	Collegamento fisico tra ESP32 e microfono I <sup>2</sup> S (Adafruit SPH0645LM4H) su breadboard. ....	33
4.7	Collegamento fisico tra ESP32 e amplificatore I <sup>2</sup> S (Adafruit MAX98357A) su breadboard. ....	34
4.8	Collegamento fisico tra ESP32 e LED di segnalazione su breadboard.	36

4.9	Schema di collegamento del bottone e dei LED di stato per la modalità hotspot .....	37
4.10	Schema del circuito completo con ESP32, microfono I <sup>2</sup> S, amplificatore MAX98357A, LED di stato e pulsante di hotspot .....	38
4.11	Layout del circuito stampato (PCB), con ESP32, LED di stato, pulsante di hotspot e connettori di alimentazione.....	39
5.1	Swapping degli array di campioni .....	44
5.2	Compressione ed impacchettamento mediante Algoritmo Run-Length S= Silenzio .....	51

## Elenco delle tabelle

3.1	Frequenze e tipi di segnale.....	13
3.2	Struttura di comunicazione tra Livello Bit e Livello Fisico .....	18
3.3	Mappatura frequenze, codici e portanti .....	19
3.4	Indirizzi di rete del Livello Link .....	24
3.5	Mappatura comandi Blynk, operazioni interne e azioni eseguite dai nodi .....	24
4.1	Mappatura hardware dei pin I <sup>2</sup> S tra ESP32 e microfono SPH0645LM4H. ....	33
4.2	Mappatura hardware dei pin I <sup>2</sup> S tra ESP32 e amplificatore MAX98357A. ....	35
4.3	Mappatura hardware dei pin di uscita dell'ESP32 verso i LED di stato. ....	36
5.1	Mappatura frequenze con divisione dei Signal Code in 0 e 1 .....	48

# Bibliografia

- [1] I. F. Akyildiz and E. P. Stuntebeck, “Wireless underground sensor networks: Research challenges,” *Ad Hoc Networks*, vol. 4, no. 6, pp. 669–686, 2006.
- [2] R. Jacksha and C. Zhou, “Measurement of rf propagation around corners in underground mines and tunnels,” *Transactions of the Society for Mining, Metallurgy, and Exploration*, vol. 340, no. 1, pp. 30–37, 2016.
- [3] L. E. Kinsler, A. R. Frey, A. B. Coppens, and J. V. Sanders, *Fundamentals of Acoustics*. Wiley, 4th ed., 2000.
- [4] P. M. Morse and K. U. Ingard, *Theoretical Acoustics*. Princeton University Press, 1986.
- [5] A. D. Pierce, *Acoustics: An Introduction to Its Physical Principles and Applications*. Acoustical Society of America, 1989.
- [6] A. Heifetz, S. Bakhtiari, X. Huang, R. Ponciroli, and R. B. Vilim, “First annual progress report on transmission of information by acoustic communication along metal pathways in nuclear facilities,” anl/ne-17/30 technical report, Argonne National Laboratory, Nuclear Engineering Division, Argonne, Illinois, Sept. 30 2017. Prepared under contract DE-AC02-06CH11357, UChicago Argonne, LLC.
- [7] M. A. Al Moshi *et al.*, “Wireless underground sensor communication using acoustic technology: A review,” *Sensors*, 2024. Open access review on acoustic WUSNs.
- [8] A. Salam *et al.*, “Taking wireless underground: A comprehensive summary,” *ACM Computing Surveys*, 2023.
- [9] M. Fishta, G. Franceschinis, A. Braglia, Alj, S. Bagheri, and C. Seceleanu, “From radio to in-pipe acoustic communication for smart water networks,” *Information*, vol. 14, no. 8, p. 435, 2023.

- [10] A. Heifetz and et al., “Transmission of information by acoustic communication along metal pathways,” Tech. Rep. ANL-17/13, Argonne National Laboratory, 2017.
- [11] O. O. Farai, J. M. Muggleton, E. Rustighi, and M. J. Brennan, “Analysis of acoustic signal propagation for reliable digital communication along mdpe pipes,” *Applied Sciences*, vol. 13, no. 15, p. 8886, 2023.
- [12] A. Heifetz and et al., “Transmission of images with ultrasonic elastic shear waves on pipes,” Tech. Rep. ANL-20/35, Argonne National Laboratory, 2020.
- [13] S. Yang, A. Kalantari, S. Markham, J. Saniie, and F. Kong, “Development of an underground through-soil wireless acoustic communication system.” Technical Report, University of Illinois Chicago, 2020.
- [14] U. Raza, A. Salam, T. Y. Al-Naffouri, and M.-S. Alouini, “A survey on wireless underground communication with acoustic waves,” *Signals*, vol. 1, no. 1, pp. 21–52, 2020.
- [15] A. Salam, U. Raza, T. Y. Al-Naffouri, and M.-S. Alouini, “Wireless underground sensor networks: A comprehensive survey and tutorial,” *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.
- [16] O. O. Farai, *Acoustic Communication in Polymeric Pipes*. PhD thesis, University of Birmingham, 2021.
- [17] Z. Zheng, J. He, Y. Li, X. Liu, and B. Han, “Acoustic wave data transmission while drilling using drilling fluid channel,” *IET Communications*, vol. 17, no. 5, pp. 593–602, 2023.
- [18] C. E. Shannon, “Communication in the presence of noise,” *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.
- [19] E. Zwicker and H. Fastl, *Psychoacoustics: Facts and Models*. Springer, 2nd ed., 1999.
- [20] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.
- [21] Espressif Systems, *ESP32 Technical Reference Manual*, 2020.
- [22] J. Smith and X. Serra, “Quadratic interpolation of spectral peaks.” Online, CCRMA, Stanford University, n.d. Available at [https://ccrma.stanford.edu/~jos/sasp/Quadratic\\_Interpolation\\_Spectral\\_Peaks.html](https://ccrma.stanford.edu/~jos/sasp/Quadratic_Interpolation_Spectral_Peaks.html).

- [23] E. Systems, *ESP32 Series Datasheet*, 2024. Disponibile su: <https://www.espressif.com/en/products/socs/esp32>.
- [24] D. Patti, *Introduction to Microcontrollers - Communication Interfaces*. Catania, Italia: Self-published, 2024.
- [25] Adafruit Industries, *Adafruit Learning System: I2S MEMS Microphone Breakout SPH0645*, 2020.
- [26] “Adafruit i2s mems microphone breakout - sph0645lm4h,” 2020.
- [27] “Q&a: Using sph0645lm4h with nordic socs (i2s timing and osr),” 2019.
- [28] Knowles, *Knowles SPH0645LM4H-B MEMS Microphone Datasheet*, 2016.
- [29] “Sph0645lm4h-b knowles datasheet and distributor page,” 2020.
- [30] D. Liu *et al.*, *Acoustic MEMS: Microphones and Micromachined Loudspeakers*. Springer, 2014.
- [31] M. Integrated, *MAX98357A: DirectDrive, Mono, Class D Amplifier with I2S Interface*, 2015. Datasheet ufficiale.
- [32] Adafruit, “Adafruit max98357 i2s class-d mono amp - guide,” 2023.
- [33] T. Instruments, *Class-D Audio Amplifier Basics*, 2016. Application Report.