UNIVERSITÀ degli STUDI di CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

PROGETTO MULTIMEDIA E LABORATORIO

Davide Scalisi

AudioMonitor

PROGETTAZIONE DI UN SISTEMA HARDWARE E SOFTWARE PER L'AQUISIZIONE, REGISTRAZIONE ED INVIO DI DATI AUDIO DIGITALE TRAMITE INTERNET

Documentazione e progetto sono stati realizzati da Davide Scalisi (1000038316) per poter conseguire l'esame di Multimedia e laboratorio del corso di laurea magistrale in informatica dell'anno accademico 2021/2022 con i docenti Filippo Stanco e Dario Allegra.

ANNO ACCADEMICO 2021/2022

Sommario

Introduzione

Lo scopo di questo progetto è quello di ottenere un dispositivo hardware che sia in grado acquisire dati audio e di connettersi tramite la rete internet ad un back-end, il quale a sua volta avrà il compito di elaborare i dati ricevuti.

Più nello specifico, il dispositivo fisico è un dispositivo di digitalizzazione di segnali audio, o più comunemente un registratore audio, il quale ha il compito di acquisire, rielaborare e trasmettere dati audio provenienti da un circuito analogico connesso ad un microfono, al back-end, il quale eseguirà ulteriori operazioni di riproduzione real-time o di codifica e salvataggio.

Glossario

Termine	Significato
ADC	Analog to Digital Converter
SPI	Serial Peripheral Interface
MCU	Micro Controller Unit
ISR	Interrupt Service Routine
UDP	User Datagram Protocol
EDA	Electronic Design Automation
IDE	Integrated Development Environment

Repository GitHub del Progetto

Link al repository GitHub

Tecnologie utilizzate

Il progetto si sviluppa in diversi ambiti e per ogni ambito sono state utilizzate le seguenti tecnologie ed i seguenti software di sviluppo:

- **Hardware**
  - Simulatore di circuiti Falstad.
  - La suite EDA KiCad.
  - Amplificatore operazionale LM358.
  - Analog-to-Digital converter (ADC) MCP3202.
  - Diversi componenti hardware complementari.

- **Firmware**
  - Scheda di sviluppo ESP32 DEVKIT V1 – DOIT.
  - Linguaggio di programmazione C++.
  - Libreria MCP3202 per il controllo dell'ADC tramite interfaccia SPI.
  - Framework Arduino.
  - Sistema operativo real-time FreeRTOS.
  - IDE VS Code + PlatformIO.

- **Back-End**
  - Node.js.
  - UDP Socket.

Hardware

Il primo passo per lo sviluppo di questo progetto è stato sicuramente lo sviluppo dell'intera parte di acquisizione hardware del segnale.

Prima di poter codificare le informazioni audio in dati, è necessario eseguire l'acquisizione del segnale audio stesso sottoforma di differenza di potenziale e questo lo si ottiene elettronicamente tramite i seguenti tre blocchi funzionali:

- Circuito di polarizzazione del microfono
- Circuito di amplificazione
- Circuito di digitalizzazione

Schema funzionale del circuito

Il circuito si divide quindi nei seguenti blocchi funzionali:

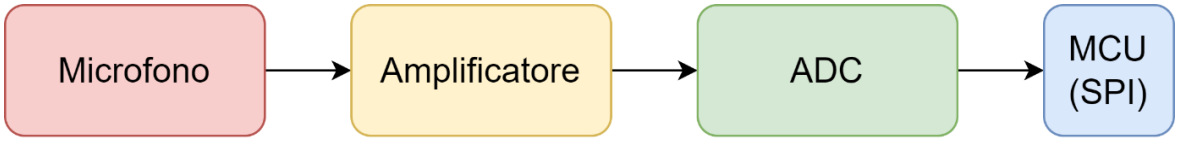

Figura 1: Blocchi funzionali in cui si divide logicamente il circuito hardware.

Circuito di polarizzazione del microfono

In questo stadio del circuito avviene la vera e propria conversione del suono da onda meccanica longitudinale a differenza di potenziale.

Il passaggio avviene utilizzando un microfono piezoelettrico:

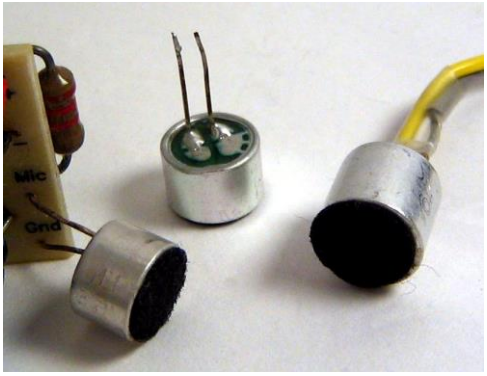

Figura 2: Capsule microfoniche contenenti microfoni piezoelettrici.

Questa tipologia di microfono converte il segnale audio in ingresso tramite una variazione di capacità in un condensatore interno composto dalla membrana stessa del microfono e da una placca fissa all'interno della capsula.

Questa variazione di capacità, tramite un opportuna polarizzazione del microfono è facilmente convertibile in una tensione elettrica.

Il circuito di acquisizione del segnale dal microfono è così composto:

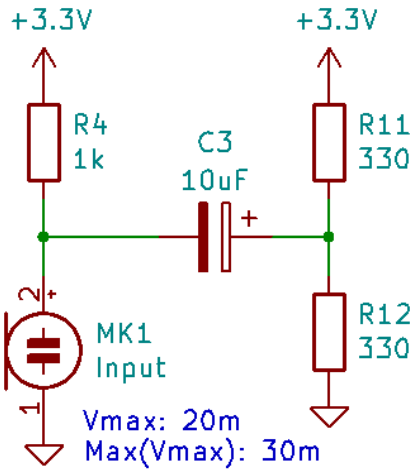

Figura 3: Circuito di acquisizione del segnale analogico proveniente dal microfono piezoelettrico.

La resistenza R4 è chiamata resistenza di polarizzazione del microfono e serve a garantire il corretto funzionamento del microfono stesso.

In questa immagine, il punto di mezzo a sinistra di C3 è dove si andrà a localizzare una certa tensione intermedia compresa tra 0 e 3.3V; questa tensione avrà principalmente la caratteristica di avere un valor medio DC non noto e variabile a cui non si è interessati (il circuito si comporta in maniera simile ad un partitore resistivo) ed un valore in AC, il quale rappresenta proprio la variazione del segnale audio rispetto al tempo.

Per poter eliminare la componente DC del segnale in modo tale da estrarre solamente la componente AC, si utilizza il disaccoppiamento capacitivo: tramite l'uso del condensatore C3 è possibile rimuovere il valor medio del segnale per poter ottenere in uscita solamente il segnale AC, il quale, come in figura, ha le caratteristiche di una tensione con picchi al più di $\pm$20-30 millivolt.
Ai fini dell'amplificazione introdotta nel blocco successivo, a destra del condensatore C3 è reintrodotto appositamente un altro offset DC pari a $3.3V/2 = 1.65V$.

Essendo infatti che l'intero circuito di amplificazione è alimentato da zero a 5V (non 3.3 per motivi legati agli amplificatori operazionali non ideali) e che il segnale AC proveniente dal microfono è un segnale avente anche picchi negativi, allora uno dei modi con cui è possibile lavorare con questo genere di segnali nel blocco di amplificazione è quello di rimuovere inizialmente l'offset DC che il segnale presenta (conseguenza dell'acquisizione con questa tipologia di microfono) per poi andare ad introdurre volutamente un nuovo offset DC noto a $\frac{V_{cc}}{2}$.

Questo offset alla metà della tensione di alimentazione si ottiene elettricamente tramite un partitore resistivo con due resistenze di valore uguale (in questo caso, R11 ed R12 a destra di C3).

In questa sezione del circuito il segnale è stato in parte preparato per l'acquisizione, ma i picchi che rappresentano il vero e proprio segnale audio sono troppo bassi per poter essere apprezzati significativamente dal circuito di acquisizione: è quindi necessaria l'amplificazione attiva del segnale tramite il blocco funzionale successivo.

Circuito di amplificazione

L'obbiettivo di questa parte del circuito è quello di amplificare il segnale in modo tale da portare i suoi picchi da $1.65 \pm 0.03V$ a $1.65 \pm \sim 1.50V$, in modo tale che l'intero segnale vari appunto da circa poco più di zero a circa poco meno di 3.3V.

Questo tipo di amplificazione, considerando fattori come l'adattamento di impedenza e la banda passante desiderata, è facilmente ottenibile tramite amplificatori operazionali:

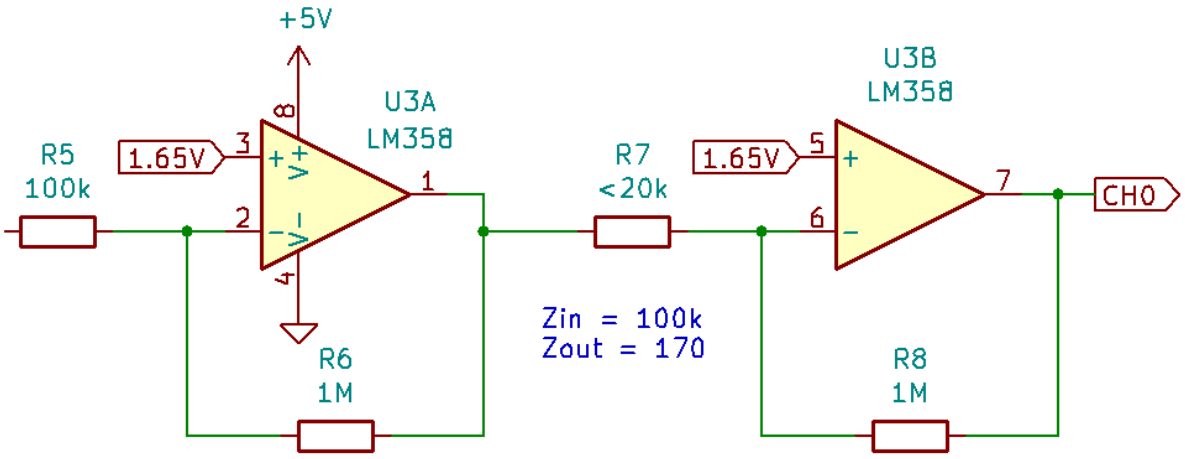

Figura 4: Circuito amplificatore analogico composto da due amplificatori operazionali in configurazione invertente.

Amplificatore invertente

L'amplificatore operazionale è un componente elettronico configurabile in vari modi di funzionamento differenti; in questo circuito, la configurazione utilizzata è a doppio amplificatore invertente con massa virtuale ad 1.65V.

Il guadagno totale è quindi pari al prodotto del guadagno dei due singoli stati e, per le formule sul guadagno dell'amplificatore invertente, si ha:

$$\begin{cases} G_1 = -\dfrac{R6}{R5} \\ G_2 = -\dfrac{R8}{R7} \\ G = G_1 G_2 \end{cases} \rightarrow G = \frac{R6 \cdot R8}{R5 \cdot R7}$$

A questo punto, considerando ad esempio un segnale di uscita massimo voluto di $1.50V$ e considerando che il guadagno è anche definito come rapporto $G = \frac{V_{out}}{V_{in}}$ dell'amplificatore stesso, si ha:

$$\frac{1.50V}{0.03V} = \frac{R6 \cdot R8}{R5 \cdot R7} \rightarrow 1.50V = \frac{R6 \cdot R8}{R5 \cdot R7} 0.03V$$

Risolvendo quindi questa equazione andando ad imporre appositamente determinati valori su alcuni di questi resistori, si ottengono più o meno i valori riportati in figura.

NB: la resistenza R7 in figura potrebbe anche essere un potenziometro da $20k\Omega$, in modo tale da poter imporre perfettamente il guadagno voluto direttamente nella fase di test dell'intero circuito.

L'amplificatore operazionale utilizzato in questo progetto è stato il classico LM358.

Circuito di digitalizzazione

Alla fine della catena di amplificazione, si arriva finalmente al blocco di digitalizzazione:

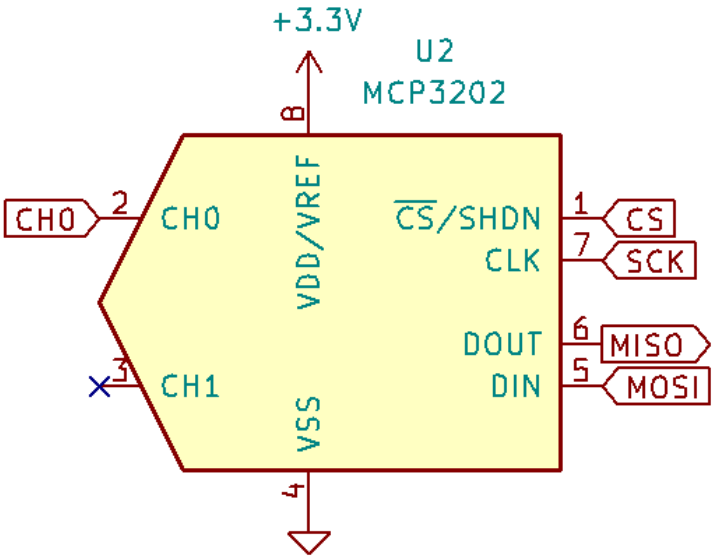

Figura 5: Circuito digitalizzazione composto da un ADC.

Questo blocco è semplicemente composto da un ADC, l'MCP3202, il quale permette effettivamente la digitalizzazione del segnale analogico direttamente proveniente dal blocco amplificatore.

L'ADC è interfacciato con il microcontrollore tramite l'interfaccia SPI (collegamenti a destra nell'immagine) ed il segnale viene dato in input tramite il CH0 (Channel 0).

Interfaccia SPI

L'interfaccia SPI è una tipologia di interfaccia Master-Slave orientata al byte molto popolare ed utilizzata in ambito microcontrollori; essa non ha limiti teorici di velocità massima e limiti pratici molto superiori alle specifiche richieste da questo progetto.

In questa applicazione, la banda passante del bus SPI deve essere almeno di:

$$2byte \cdot 16kHz = 32kbps$$

Il bus SPI è fisicamente composto da quattro collegamenti elettrici tra i dispositivi:

- **MISO** (Master In - Slave Out)
- **MOSI** (Master Out -Slave In)
- **SCK** (Serial Clock)
- **CS** (Chip Select)

Circuito finale

L'intero circuito progettato ed utilizzato è quindi il seguente:

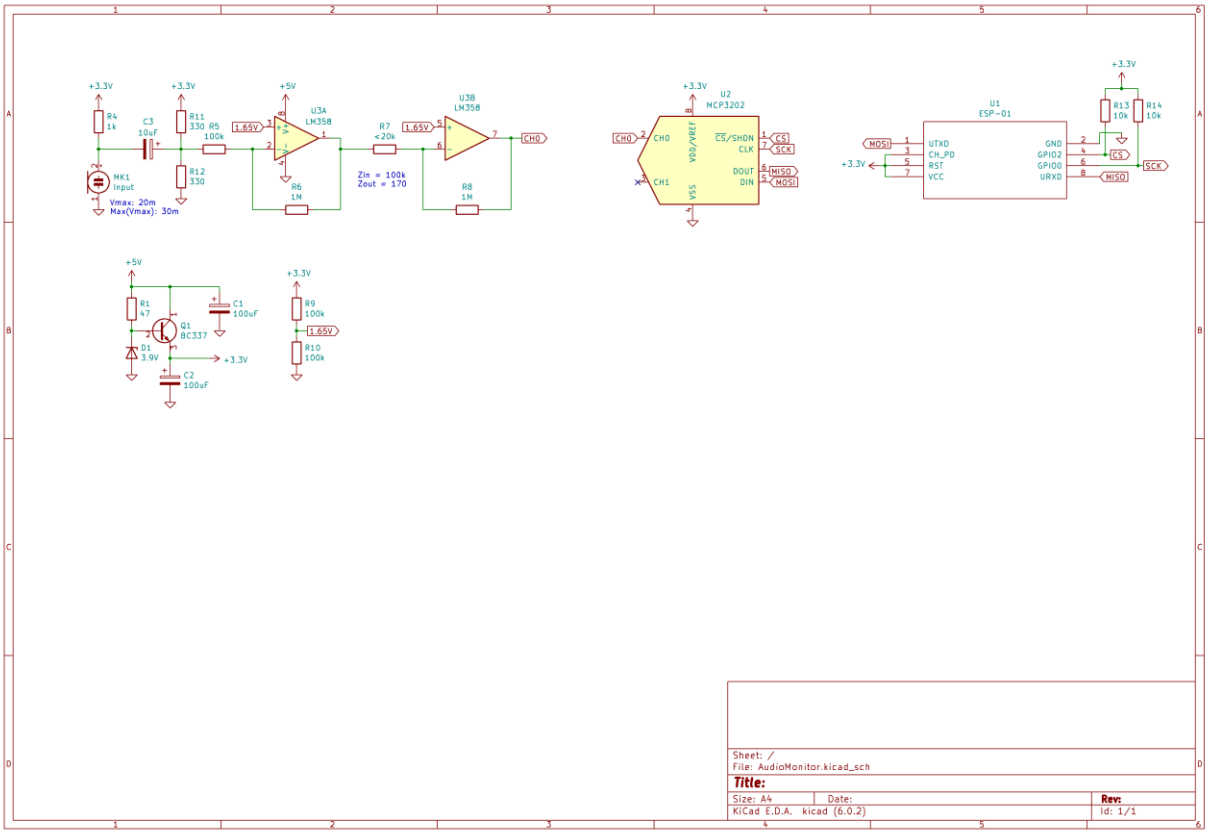

Figura 6: Schema finale del circuito elettrico.

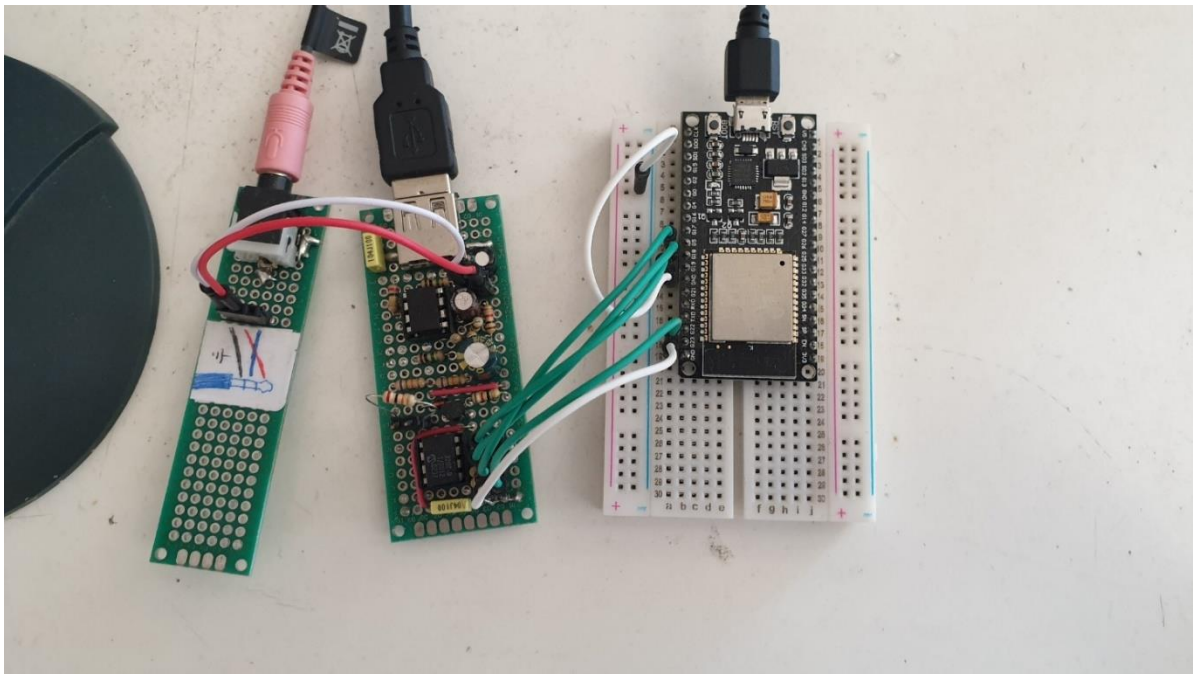

Figura 7: Prototipo del dispositivo hardware su due basette millefori (a sinistra) e su una breadboard (a destra).

Firmware

Una volta che all'ADC viene inviato ad un certo istante il comando di campionamento, questo esegue l'operazione di quantizzazione inviando come risposta al sistema microcontrollore (in questo caso, un ESP32), i dati appositamente digitalizzati.

Tipologia di ADC

L'MCP3202 è un ADC ad approssimazioni successive della Microchip a doppio canale configurabile in modalità single-ended o in modalità pseudo-differenziale.

La risoluzione di questo ADC è pari a 12 bit per campione con un tasso di campionamento massimo di 100ks/s.

Interfaccia tra ADC ed MCU

L'ADC comunica con il sistema MCU tramite l'interfaccia SPI (Serial Peripheral Interface), la quale a sua volta ha una velocità massima teorica illimitata e pratica di circa 50Mhz; è molto importante sottolineare che è necessaria l'analisi dalla velocità massima per ogni bus di passaggio per cui transita l'informazione del segnale audio: parlando infatti di un applicazione real-time, nessun blocco deve introdurre colli di bottiglia.

Il bus di comunicazione a livello fisico è connesso in maniera standard tra i pin dedicati SPI compatibili dell'ADC ed i GPIO dedicati alla periferica SPI hardware presente all'interno del microcontrollore.

Libreria MCP3202

Per quanto riguarda l'implementazione del protocollo di comando dell'ADC tramite il bus SPI, è stata da me implementata la libreria MCP3202, la quale espone un'API semplificata per poter controllare ad un livello più alto l'ADC stesso; questo controllo avviene inviando direttamente il comando di campionamento e restituendo quindi il conseguente dato acquisito.

NB: la libreria funziona solamente in modalità single-ended in entrambi i canali dell'ADC.

Modello a thread del firmware

Prima di implementare il firmware utilizzando il modello a thread, si è provato ad utilizzare il classico modello ad event loop (versione 1.0.0), ma per l'utilizzo in real-time il modello si è dimostrato non sufficientemente flessibile ed è quindi nata spontaneamente l'esigenza di cambiare la struttura di base del firmware da un modello event loop ad un modello basato su thread (versione 1.1.0).

Piattaforma ESP32

Il sistema MCU fisico dove il firmware è eseguito è un ESP32, una piattaforma open source progettata da Espressif e compatibile anche con il framework Arduino; questa piattaforma è munita di molteplici periferiche integrate tra cui WiFi 802.11 b/g/n teoricamente fino a 150Mbps.

L'ESP32 è munito di un processore dual core a 32 bit operante fino ad una frequenza di 240Mhz; entrambi i processori sono gestiti internamente dal sistema operativo real-time FreeRTOS ed è possibile scegliere in fase di creazione di un thread, in quale core questo dovrà essere schedulato ed eseguito.

NB: esistono anche dei modelli single-core di ESP32, ma in questo progetto è stato utilizzato il classico modello dual-core.

Cenni a FreeRTOS

Il firmware per l'ESP32 è stato sviluppato e compilato tramite il plugin PlatformIO per VS Code utilizzando come base il framework Arduino, il quale a sua volta integra di default una versione modificata per ESP32 di FreeRTOS, chiamata ESP-IDF FreeRTOS; questo SO è quindi a tutti gli effetti un SO minimale per microcontrollori basato sul modello a thread.

Timer hardware

Un altro aspetto molto importante di ESP32 è la presenza di quattro timer hardware: una specifica periferica che ha anche lo scopo di generare interrupt software periodici con dei timing molto precisi.

Una volta che l'interrupt viene invocato, a questo viene associata una routine di servizio che sarà incaricata di segnalare al sistema che è il momento di campionare.

Interfaccia con il back-end

L'interfacciamento con il back-end è implementato utilizzando delle socket UDP, ma inizialmente si è tentato anche l'utilizzo di WebSocket come protocollo di trasporto dati.

WebSocket si è però dimostrato inadeguato per lo streaming di audio non compresso in real-time e per questo motivo si è deciso di passare ad UDP per poter ottenere delle performance migliori.

Struttura del firmware

Come spiegato precedentemente, il firmware è sviluppato utilizzando il multithreading offerto da FreeRTOS:

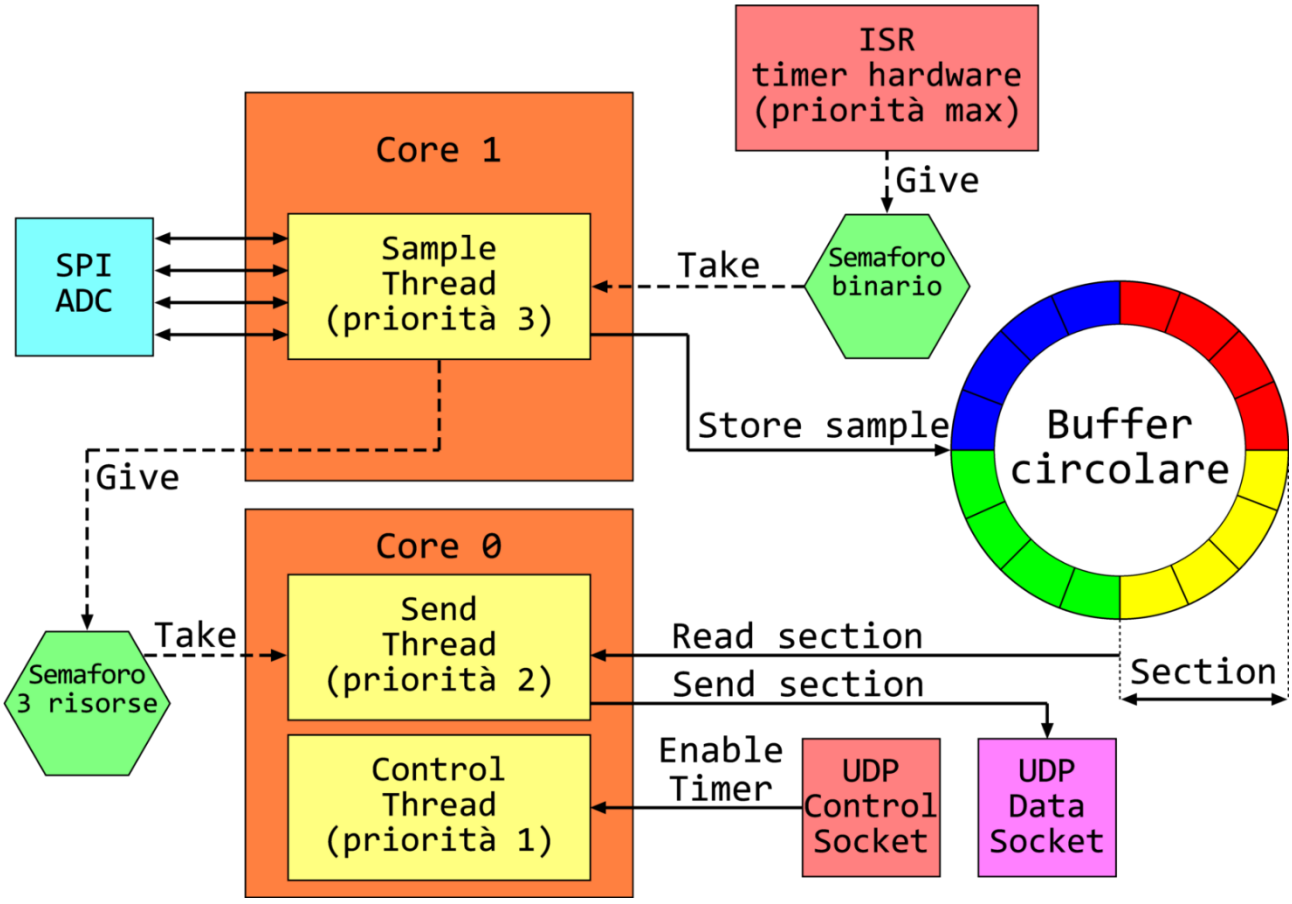

Figura 8: Schema funzionale del firmware progettato per il dispositivo fisico.

Questo schema rappresenta la struttura logica del firmware ed il percorso che i dati seguono prima di essere trasmessi tramite UDP al back-end.

Abilitazione del dispositivo

Prima di poter iniziare l'acquisizione vera e propria dei dati provenienti dall'ADC, deve essere presente un server web che invii un certo comando di abilitazione tramite una socket UDP all'indirizzo del microcontrollore ed alla porta 5000.

Una volta ricevuto il comando di abilitazione esterno tramite la socket, il dispositivo salva l'IP e la porta del mittente tramite le seguenti due istruzioni:

```
remoteIP = sock.remoteIP();
remotePort = sock.remotePort();
```

Questi indirizzi serviranno successivamente per l'invio dei dati audio in PCM attraverso la socket dati.

Control Thread

Tutta questa fase di controllo di abilitazione è gestita da Control Thread.

Una volta ricevuto il pacchetto UDP di abilitazione, Control Thread abilita il Timer0 preconfigurato dell'ESP32 il quale, come detto precedentemente, deve avere una frequenza di generazione dell'interrupt associato (di chiamata all'ISR) pari a 16kHz, cioè alla frequenza a cui si vuole fare campionare l'ADC.

ISR timer hardware

La routine software associata al Timer0 ha il compito specifico di segnalare a Sample Thread che è il momento giusto per campionare; questa segnalazione avviene tramite un semaforo binario condiviso tra l'ISR e Sample Thread.

Si potrebbe anche pensare di campionare direttamente all'interno dell'ISR, ma in questo caso l'operazione prende troppo tempo ed il sistema non riesce a gestire l'evento in maniera abbastanza rapida (il watchdog genera un eccezione che porta al riavvio del firmware).

Sample Thread

Per ovviare al problema di non poter campionare all'interno dell'ISR, è stato introdotto Sample Thread, il quale ha proprio il compito di effettuare le richieste di campionamento all'ADC tramite l'interfaccia SPI utilizzando la libreria MCP3202:

```
xSemaphoreTake(sem_ready_for_sampling, portMAX_DELAY);
buf[cur] = adc.read();
```

Deferred interrupt

In questa parte di firmware è implementato un meccanismo detto di deferred interrupt, cioè un meccanismo necessario in tutti quei casi in cui si deve comunicare allo scheduler di controllare se qualche thread di priorità più alta del thread interrotto dall'interrupt ed in attesa di qualche semaforo, si è sbloccato in seguito all'interrupt stesso; per spiegare questo meccanismo, si consideri:

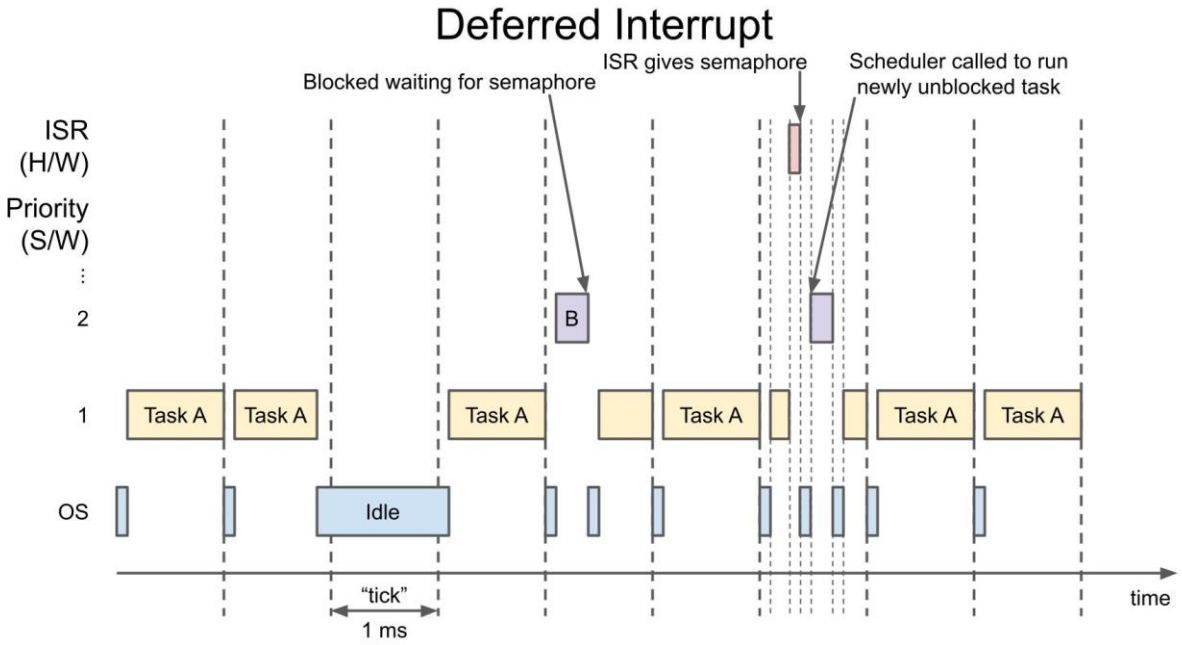

Figura 9: Esempio di situazione in cui è necessario applicare il meccanismo del Deferred Interrupt.

Supponiamo per semplicità di essere in un sistema mono core, allora, se si pone B pari a Sample Thread ed A pari ad un qualsiasi altro thread con priorità strettamente minore di B, si ha:

1) Ad un certo istante di tempo, B si blocca in attesa del semaforo binario.
2) Vengono eseguiti altri task fino a quando tocca ad A.
3) Durante l'esecuzione di A, scatta l'interrupt di campionamento e quindi viene invocata l'ISR, la quale non essendo un thread ma essendo invece un interrupt hardware, ha priorità superiore a qualsiasi thread software.

4) Durante l'ISR, viene sbloccato il semaforo binario che B stava attendendo.
5) Essendo B con priorità maggiore di A, allora all'interno della ISR verrà chiamata un API di FreeRTOS che indicherà allo scheduler di effettuare immediatamente il context switch a B al ritorno dalla ISR.

6) Finita l'ISR, di default A torna il controllo allo scheduler e quest'ultimo darà immediatamente priorità a B, in quanto avente priorità maggiore di A.

Implementazione del deferred interrupt nella ISR

Il meccanismo del deferred interrupt è implementato nell'ISR come segue:

```
void IRAM_ATTR onTimerISR(){
  BaseType_t task_woken = pdFALSE;

  //Sblocca il semaforo per campionare.
  if(!xSemaphoreGiveFromISR(sem_ready_for_sampling, &task_woken))
    Serial.println("errQUEUE_FULL in xSemaphoreGiveFromISR.");

  //API di FreeRTOS per gestire il deferred interrupt.
  if(task_woken)
    portYIELD_FROM_ISR();
}
```

Se il parametro `task_woken` dopo la chiamata a `xSemaphoreGiveFromISR` vale `pdTRUE`, allora esiste un thread di priorità più alta di quello interrotto che si è sbloccato a seguito della chiamata stessa.

A questo punto, deve essere comunicato allo scheduler di dare priorità a quello specifico thread (nel caso specifico, a Sample Thread); questo lo si fa tramite la chiamata a `portYIELD_FROM_ISR()`.

Buffer circolare

L'ADC, come detto precedentemente, ha una risoluzione di 12 bit per campione ed ogni campione deve essere salvato in una qualche struttura dati per poter poi essere inviato: a questo scopo viene introdotto un buffer circolare.

Dimensioni del buffer

La dimensione del buffer è strettamente legata alla larghezza di banda messa a disposizione dal protocollo UDP e, soprattutto, dalla qualità chip WiFi fisico utilizzato.

Il problema deve essere analizzato avendo chiaro che l'obbiettivo è quello di trovare un valore quasi ottimo di campioni per pacchetto, tenendo presente che si è vincolati ad una certa quantità massima di pacchetti inviabili (vincolo imposto dalla scheda di rete del microcontrollore).

Sperimentalmente si è determinato che con un payload di $w = 1000$ byte, il massimo throughput ottenibile è di $p = 50$ pacchetti al secondo.

Sapendo quindi che:

1) Ogni campione pesa $w = 2$ byte.
2) La frequenza di campionamento vale $f_s = 16.000Hz$.
3) Il numero di pacchetti al secondo vale $p = 50$.

Allora il numero di campioni per pacchetto è dato da:

$$n = \frac{w \cdot f_s}{p} = \frac{2 \cdot 16.000}{50} = 640 \text{ byte} = 320 \text{ campioni per pacchetto}$$

Il valore di 640 byte per pacchetto è inferiore ai 1000 byte massimi supponendo che si inviino 50 pacchetti al secondo e quindi, il valore di 320 campioni per pacchetto rappresenta un buon compromesso.

Avendo calcolato n, il buffer circolare che ha dato origine al problema avrà, per sicurezza, un valore m pari a:

$$m = s \cdot n = 1280$$

Dove $s = 4$ è il numero di sezioni in cui il buffer circolare viene logicamente suddiviso; ogni sezione è rappresentata in figura tramite i quattro colori delle caselle del buffer circolare stesso.

NB: è importante che valga $s \geq 2$: deve infatti essere presente una sezione attuale di scrittura dei campioni ed una sezione attuale di lettura dei campioni; per evitare che avvengano troppo spesso sovrapposizioni, si è scelto il valore $s = 4$.

Sovrapposizioni nel buffer circolare

Essendo il buffer una risorsa condivisa tra Sample Thread (scrittore) e Send Thread (lettore), l'accesso a questa struttura dati non può essere indiscriminato, ma deve essere orchestrato da un semaforo contatore, il quale valore massimo deve essere sempre $s - 1$.

Questa gestione della struttura dati condivisa risolve implicitamente le possibili sovrapposizioni di lettura e scrittura descritte precedentemente.

Esempio di risoluzione delle sovrapposizioni

Supponiamo di avere Sample Thread che opera con una frequenza di campionamento maggiore della capacità di invio dati del sistema: in questo scenario si genererebbe velocemente un buffer saturo e si avrebbe la sovrapposizione dei due buffer di lettura e scrittura dei due thread (Sample Thread e Send Thread).

Utilizzando il semaforo contatore sopra descritto, se Sample Thread producesse dati più velocemente di quanto Send Thread possa consumarli, allora il semaforo rimarrà al valore massimo

e di conseguenza, Sample Thread si sospenderebbe in attesa che Send Thread decrementi il semaforo.

Se invece si considera il caso opposto dove Send Thread consuma troppo velocemente i dati, allora il semaforo rimarrà a zero sospendendo conseguentemente Send Thread fino a quando Sample Thread non avrà riempito interamente la prossima sezione di dati ed avrà quindi incrementato nuovamente il semaforo.

Problema del produttore-consumatore

Questo problema è noto in letteratura come problema del produttore-consumatore e la classica soluzione che i sistemi operativi moderni adottano è proprio quella di utilizzare semafori (binari o contatori) per poter sincronizzare dei vari processi che interagiscono con una certa struttura dati condivisa.

Send Thread

Quest'ultimo thread ha il compito di inviare dati al back-end sfruttando la UDP Data Socket; i campioni vengono quindi depositati singolarmente da Sample Thread nel buffer circolare, mentre vengono letti e spediti sezione per sezione da Send Thread:

```
xSemaphoreTake(sem_ready_for_sending, portMAX_DELAY);
xSemaphoreTake(mutex_remote_ip_port, portMAX_DELAY);

sock.beginPacket(remoteIP, remotePort);
sock.write(
  (uint8_t*) &buf[section * UDP_PAYLOAD_SIZE],
  UDP_PAYLOAD_SIZE * sizeof(uint16_t)
);
sock.endPacket();

xSemaphoreGive(mutex_remote_ip_port);
```

Il primo parametro della `sock.write` rappresenta l'offset al primo elemento della sezione corrente ed il secondo rappresenta invece il numero byte che devono essere scritti nella socket.

Sezioni critiche

Sia in Sample Thread che in Send Thread, l'accesso al codice è inizialmente bloccato dai rispettivi semafori `sem_ready_for_sampling` e `sem_ready_for_sending`.

Questi due semafori hanno il compito di gestire la sincronizzazione tra i diversi thread e di garantire la coerenza delle varie informazioni generate dal sistema.

Ha un ruolo logico totalmente diverso il mutex `mutex_remote_ip_port` presente all'interno di Send Thread: un mutex è infatti un particolare tipo di semaforo binario che ha il compito di implementare la mutua esclusione tra diversi thread, cioè l'accesso simultaneo di più di un thread in determinate sezioni di codice, denominate sezioni critiche.

Un mutex non serve quindi a gestire la sincronizzazione, bensì serve per gestire la mutua esclusione e lo scopo finale di `mutex_remote_ip_port` è quindi quello di proteggere le sezioni critiche di codice dove vengono modificate le due variabili `remoteIP` e `remotePort`.

Codice di Control Thread

Se infatti si riporta parte del codice di Control Thread:

```
xSemaphoreTake(mutex_remote_ip_port, portMAX_DELAY);
remoteIP = sock.remoteIP();
remotePort = sock.remotePort();
xSemaphoreGive(mutex_remote_ip_port);
```

Si nota che anche in questo codice sono modificate le due variabili `remoteIP` e `remotePort`; ci si rende subito conto che è quindi necessario un mutex per poter evitare sezioni critiche a runtime tra Control Thread e Send Thread.

Spinlock e Mutex in ESP-IDF FreeRTOS

Sia i mutex che gli spinlock sono strumenti atti a garantire la mutua esclusione, ma nei comuni sistemi operativi la differenza è trascurabile.

È doveroso premettere che su ESP32 in particolare, è utilizzato ad alto livello il framework Arduino, il quale a sua volta si appoggia ad una versione modificata di FreeRTOS chiamata ESP-IDF FreeRTOS; questo sistema operativo non permette nativamente di gestire il multi-core e, per questa ragione, gli sviluppatori della piattaforma ESP32 sono stati obbligati a modificare ed adattare FreeRTOS per poter ottimizzare al meglio l'utilizzo dell'hardware.

Quando quindi si parla di sistemi operativi embedded come FreeRTOS, è necessario specificare che sussiste una grande differenza tra un mutex ed uno spinlock:

- **Mutex**
  Un classico mutex software che si appoggia alle istruzioni dedicate dell'architettura e che garantisce la mutua esclusione.

  In caso tentativo di entrata di un thread in una sezione critica già occupata da un altro thread in ready su uno dei due core o in running sull'altro core, il thread si mette semplicemente in stop in attesa che la sezione critica si liberi.

Il problema dell'utilizzo di questo tipo di mutex sta nel fatto che ci si trova in un sistema dove possono occorrere interrupt hardware che fanno riferimento ad ISR e, siccome questa non è considerabile come un thread, non è possibile metterla in stop.

Esistono infatti delle specifiche funzioni di lock ed unlock non bloccanti fatte apposta per essere chiamate all'interno di ISR; se la chiamata a questa lock dovesse fallire, non sarebbe infatti possibile bloccare un interrupt ed il codice dovrebbe logicamente andare in errore.

- **Spinlock**
  In FreeRTOS classico, uno spinlock è un mutex molto restrittivo: tramite la chiamata alle sue lock ed unlock, vengono rispettivamente disattivate e riattivate tutte le interrupt hardware (quindi anche lo scheduler), eliminando il problema che si veniva a creare con i classici mutex.

  In più, ESP-IDF FreeRTOS aggiunge agli spinlock un altro livello di protezione da cui questi prendono il nome: se infatti si dovesse ripresentare lo stesso tentativo di accesso ad una sezione critica già occupata da un altro thread in running nell'altro core in quel preciso istante, il thread non andrebbe in stop, bensì aspetterà in spinlock fintantoché l'altro thread in esecuzione nell'altro core non sbloccherà quella determinata sezione critica.

  Questa protezione aggiuntiva non ha senso in sistemi mono-core, ma in sistemi a più core è necessario includere un tale meccanismo per prevenire situazioni inattese con le varie sezioni critiche e per massimizzare il throughput: mettere infatti in pausa un processo potrebbe a volte costare di più rispetto a fargli fare busy waiting.

Nella parte di codice per difendere da race conditions le variabili `remoteIP` e `remotePort`, basta semplicemente utilizzare un mutex invece di uno spinlock, in quanto la sezione critica non è interna ad alcuna ISR e riguarda solamente due thread; utilizzare uno spinlock sarebbe decisamente un errore dato che non ci si può permettere di perdere interrupt hardware o di fermare lo scheduler: si sta eseguendo contemporaneamente un campionamento real-time da un ADC.

In conclusione, in queste tipologie di sistemi integrati real-time, la best practice è quella di tenere le sezioni critiche in generale più corte possibile onde evitare di perdere interrupt o di fermare troppo a lungo lo scheduler.

Problema dell'inversione di priorità

Sia i mutex che gli spinlock risolvono il bug dell'inversione di priorità rispettivamente tramite l'algoritmo della Priority Inheritance e, banalmente, disabilitando qualunque altro processo che tenti l'accesso alla sezione critica in questione.

L'implementazione di un semaforo binario differisce in generale da quella di un mutex proprio perché un mutex implementa spesso al suo interno l'algoritmo per evitare problemi come l'inversione di priorità.

Back-end

L'abilitazione del dispositivo è effettuata dal client (back-end), il quale ha l'iniziale compito di inviare al dispositivo MCU uno specifico pacchetto chiamato pacchetto di abilitazione.

Pacchetto di abilitazione

Il pacchetto di abilitazione può essere inviato da un qualsiasi client che sia in grado di aprire una connessione UDP verso un altro host; questo pacchetto deve contenere un payload pari a 0x01, comando che comunica all'MCU di iniziare ad acquisire ed inviare i dati audio in PCM.

I dati audio verranno inviati al client stesso che ha effettuato la richiesta di abilitazione e, tramite questo meccanismo, è quindi possibile evitare di specificare il client nel codice o in un possibile file di configurazione, in quanto un client diventa un qualsiasi applicativo in grado di inviare il pacchetto di abilitazione via UDP.

La porta predefinita presso la quale l'MCU si mette in ascolto in attesa di pacchetti di abilitazione è specificata nel codice all'interno del file *const.h*:

```
#define UDP_LOCAL_PORT 5000
```

Pacchetto di disabilitazione

Se, analogamente, si desidera interrompere l'acquisizione dei dati, il client dovrà inviare l'analogo pacchetto di disabilitazione contenente un payload pari a 0x00, il quale indicherà al dispositivo MCU che non è più necessario acquisire ed inviare i dati audio; in questo modo, il dispositivo MCU torna in stato di ready, cioè in attesa di un altro pacchetto di abilitazione, il quale potrebbe anche provenire da un host diverso dal precedente.

Dettagli del canale di comunicazione

Come discusso precedentemente, il canale di comunicazione è instaurato tramite socket; dal punto di vista del dispositivo MCU, esistono due socket per la comunicazione ed entrambe utilizzano il protocollo di trasporto UDP:

- **Socket di controllo**
  Questa è la socket in ascolto sulla porta `UDP_LOCAL_PORT` tramite la quale vengono ricevuti ed interpretati i vari pacchetti di controllo (in questo caso, solamente i pacchetti di abilitazione e di disabilitazione).

  La socket è quindi logicamente di tipo simplex, in quanto viene utilizzata solamente per la ricezione di messaggi di controllo da parte di client esterni.

- **Socket dati**
  La socket dati è sempre una socket simplex e viene utilizzata per inviare al client che ha effettuato la richiesta di abilitazione, i dati audio acquisiti codificati in PCM.

Dettagli del client

Come tipologia di client, è possibile utilizzare un qualsiasi tipo di client UDP; per il debug iniziale del firmware è stato inizialmente utilizzato l'applicativo Packet Sender.

Il client definitivo è invece stato scritto utilizzando Node.js tramite l'utilizzo dei seguenti strumenti:

- Primitive di rete per poter gestire in maniera diretta le socket.
- Primitive per poter accedere all'output audio del sistema.
- L'encoder MP3 Lame per poter eseguire la codifica dei dati PCM in dati conformi allo standard MP3.

- Primitive per poter accedere al filesystem locale, in modo tale da poter salvare i dati codificati in MP3 sotto forma di file MP3.

Successivamente è stata presa la decisione di creare due client diversi a seconda della funzionalità desiderata dall'utente finale, uno chiamato `speaker.js` e l'altro chiamato `recorder.js`.

I pacchetti utilizzati in entrambi gli applicativi che permettono l'utilizzo delle primitive sopra citate sono:

```
const dgram = require('dgram');          //UDP Socket.
const stream = require('stream');        //File as stream.
const readline = require('readline');    //Read from stdin.
const Speaker = require('speaker');      //Write into the speaker buffer.

const lame = require('@suldashi/lame'); //MP3 data encoding.
const fs = require('fs');               //File system.
```

Operazioni preliminari

Prima di poter utilizzare uno dei due client, è necessario installare tutte le dipendenze necessarie descritte sopra; per far questo, bisogna aver in primo luogo installato localmente Node.js.

A questo punto, basta posizionarsi con una qualsiasi shell all'interno della cartella `AudioMonitor/ESP32/AudioMonitor-Server` per poi lanciare il comando:

```
npm install
```

A questo punto, tutte le dipendenze necessarie per l'utilizzo dei due applicativi dovrebbero essere installate e presenti all'interno della cartella `node-modules`.

Speaker.js

Il client `speaker.js` è un applicativo che permette di effettuare direttamente la riproduzione dei dati audio ricevuti tramite la socket.

Il codice si basa principalmente sul concetto di stream:

```
//Stream di passaggio dove verranno immessi i campioni.
const readStream = new stream.Readable();
```

Tramite l'oggetto `readStream` avviene quindi il vero e proprio passaggio dei dati da un elemento del codice ad un altro.

Il funzionamento generale del codice è quindi descrivibile nel seguente modo:

1) Alla ricezione di un nuovo campione `sample`, esegui `readStream.push(sample)`, aggiungendo così il nuovo campione in coda nello stream.

2) Concatena il contenuto presente in `readStream` all'oggetto `speaker`, il quale ha il compito di riprodurre i campioni tramite l'output audio di sistema:

```
readStream.pipe(speaker);
```

Oggetto speaker

Il modo in cui viene definito l'oggetto `speaker` è il seguente:

```
//Parametri dei dati PCM in input.
const speaker = new Speaker({
  channels:   1,
  bitDepth:   16,
  sampleRate: 16000
});
```

In questo modo, si comunica all'API di interfacciamento con l'audio di sistema che i campioni sono stati acquisiti secondo le caratteristiche specificate, le quali devono ovviamente essere le medesime di quelle impostate nel firmware dell'MCU.

NB: la `bitDepth` fisica dei campioni è in realtà di 12 bit, ma questo aspetto non crea problemi in quanto basta rimappare linearmente i numeri ricevuti in modo tale che il range dinamico del segnale vari in un range effettivo di 16 bit; viene in pratica effettuata un'operazione di upscaling campione per campione del segnale ricevuto.

Recorder.js

Questo eseguibile ha invece il seguente funzionamento generale:

1) Alla ricezione di un nuovo campione `sample`, esegui `readStream.push(sample)`, aggiungendo così il nuovo campione in coda nello stream.

2) `readStream.pipe(encoder);`
3) `readStream.pipe(file);`

Analogamente al caso precedente, i dati originali passano attraverso `readStream` subendo varie trasformazioni: in primo luogo, i dati in PCM vengono convertiti in dati MP3 tramite l'oggetto `encoder`, per poter essere successivamente salvati in uno specifico file tramite l'uso dell'oggetto `file`.

Oggetto encoder

L'oggetto `encoder` è definito come segue:

```
//Encoder MP3.
const encoder = new lame.Encoder({
  //Input
  channels:   1,
  bitDepth:   16,
  sampleRate: 16000,

  //Output
  bitRate:        128,
  outSampleRate:  16000,
  mode:           lame.MONO
});
```

Come si può osservare, l'oggetto `encoder` funziona allo stesso modo dell'oggetto `speaker` nel codice di `speaker.js`, cioè come uno stream.

A questo punto, gli unici parametri che si devono specificare per la corretta istanziazione dell'encoder sono i soliti parametri con i quali sono stati acquisiti i suddetti campioni in PCM più le caratteristiche dei dati MP3 in output dall'encoder stesso.

Nella configurazione utilizzata, l'istanza dell'encoder Lame è programmata in modo tale da produrre un flusso di dati compresso in output con un bit rate pari a 128kbps ed un sample rate di 16kHz in modalità mono.

Oggetto sock

In entrambi gli applicativi, la comunicazione tramite socket UDP è realizzata tramite l'oggetto sock:

```
const sock = dgram.createSocket("udp4");
```

Questo oggetto rappresenta quindi una vera e propria socket e, come tale, la prima operazione da effettuare è un binding alla porta specifica relativa all'applicativo:

```
sock.bind(PORT, "0.0.0.0");
```

Come definito lato MCU, la porta di ricezione predefinita per l'applicativo è la 5000:

```
const PORT = 5000;
```

Da questa porzione di codice in poi, verrà assegnata la seguente funzione di callback per gestire i pacchetti in arrivo sulla porta 5000:

```
sock.on("message", (message, remote) => {
  const buf = new Int16Array(message.length/2);

  for(let i=0; i<message.length/2; i++){
    //Upscaling da 12 a 14 bit (aumento volume).
    buf[i] = map(message.readInt16LE(i * 2), 0, 4095, -16384, 16383);
  }

  readStream.push(Buffer.from(buf.buffer));
});
```

Come accennato precedentemente, l'upscaling avviene campione per campione direttamente all'interno della callback di ricezione dati.

Un'altra operazione che deve essere presa in considerazione è l'invio del pacchetto di abilitazione; questo avviene tramite il seguente codice:

```
//Comando di inizio campionamento.
const message = Buffer.alloc(1);
message.writeUInt8(1, 0);

sock.send(message, 0, message.byteLength, PORT, HOST);
```

Infine, tramite la pressione di un qualsiasi tasto durante l'esecuzione di uno dei due client, è possibile terminare correttamente il client stesso andando a trasmettere il pacchetto di disabilitazione all'MCU (codice da recorder.js):

```
//Premere un tasto per uscire.
readline.emitKeypressEvents(process.stdin);

if(process.stdin.isTTY)
  process.stdin.setRawMode(true);

process.stdin.on("keypress", (chunk, key) => {
  console.log("Salvo e disconnetto...");

  //Comando di fine campionamento.
  const message = Buffer.alloc(1);
  message.writeUInt8(0, 0);

  sock.send(message, 0, message.byteLength, PORT, HOST, () => {
    sock.unref();
    sock.close();

    file.end(() => process.exit());
  });
});
```

Utilizzo dei due applicativi

Per l'utilizzo dei due applicativi da shell, la sintassi richiesta è la seguente:

- recorder.js: `node recorder IP_MCU nomeFile`
- speaker.js: `node speaker IP_MCU`

Conclusioni

L’intero progetto è stato infine un successo nonostante le molteplici difficoltà incontrate nel corso dello sviluppo delle tre fasi ed il prossimo passo sarà sicuramente quello di rendere il tutto facilmente trasportabile tramite il design di un apposito circuito stampato atto a contenere tutti i componenti hardware descritti nell’apposita sezione.

Un'altra miglioria sarà sicuramente quella di scrivere un apposito client con interfaccia grafica più raffinato dei due client in linea di comando utilizzati.

Il risultato finale è quindi quello desiderato inizialmente: un dispositivo di monitoring dell’audio espandibile sotto tutti i punti di vista e che funzioni attraverso la rete internet.