

Università Politecnica delle Marche

Facoltà di Ingegneria

Corso di Laurea triennale in Ingegneria Elettronica

e delle tecnologie digitali

Implementazione di un DSP audio su piattaforma Raspberry Pi

Implementation of an audio DSP on a Raspberry Pi platform

Tesi di Laurea di:

Francesco Giannessi

Relatore:

Dott. Alessandro Terenzi

Correlatore:

Prof.ssa Stefania Cecchi

A.A. 2025-2026

Abstract

The study involves the design and implementation of the entire audio chain starting from the acquisition, and then developing the algorithm for processing and reproducing the signal from scratch with the related parameter management. The code that manages the real-time DSP logic uses a Raspberry Pi platform, using C++ in addition to threading to manage the workflow faster.

The setup used includes a sound card that converts the raw signal from analog to digital, the processed signal from digital to analog, and manages communication with the processor; the parameters can be changed in real time via DIP switches and potentiometers connected to the Raspberry's GPIO pins.

The algorithm involves the acquisition, processing and playback of a 16-bit stereo audio signal. Its main function is to process the signal via a series of independent parametric high-pass, low-pass, and band-pass biquad filters, with automatic generation of the coefficients via the RBJ (Robert Bristow-Johnson) method. The problem that is solved is that of being able to insert digital crossovers, thus eliminating analog filters, which are known to be very bulky, expensive and subject to parameter variations over time.

Sommario

Lo studio prevede la progettazione e l'implementazione dell'intera catena audio partendo dall'acquisizione, per poi sviluppare da zero l'algoritmo per l'elaborazione e la riproduzione del segnale con la relativa gestione dei parametri. Il codice che gestisce la logica del DSP real-time utilizza una piattaforma Raspberry Pi, utilizzando il C++ in aggiunta al threading per gestire più velocemente il flusso di lavoro.

Il setup utilizzato prevede una scheda audio che converte da analogico a digitale il segnale raw, da digitale ad analogico il segnale processato, e gestisce la comunicazione con il processore; i parametri sono modificabili in tempo reale tramite DIP-Switch e potenziometri connessi ai pin di GPIO della Raspberry.

L'algoritmo prevede l'acquisizione, l'elaborazione e la riproduzione di un segnale audio stereo 16-bit. La sua funzione principale è quella di processare il segnale tramite una serie di filtri biquad parametrici indipendenti passa alto, passa basso, e passa banda, con generazione automatica dei coefficienti tramite metodo RBJ (Robert Bristow-Johnson). Il problema che si va a risolvere è quello di poter inserire i crossover digitali eliminando così i filtri analogici, noti per essere molto ingombranti, costosi e soggetti a variazioni dei parametri nel tempo.

Indice

1 Introduzione **1**

2 Stato dell'arte **3**

2.1 Obiettivi 4

3 Teoria dei filtri **5**

3.1 Poli, zeri e stabilità del filtro 6

3.2 Struttura del biquad 8

3.3 Aggiornamento dei coefficienti con metodo RBJ 9

3.3.1 Low-pass filter 10

3.3.2 High-pass filter 10

3.3.3 Band-pass filter 10

4 Descrizione dell'Hardware **11**

4.1 Informazioni generali 11

4.2 Raspberry Pi-4 12

4.3 Behringer UCA-222 13

4.4 ADS1115 15

4.5 Schemi di collegamento 17

4.5.1 Audio 17

4.5.2 Analog input 19

4.5.3 Digital input 21

5 Architettura e funzionamento del DSP **23**

5.1 Struttura logica del sistema 23

5.2 Elaborazione audio a blocchi 24

5.3 Threading 25

6 Installazione e configurazione di Pi OS **27**

6.1 Avvio automatico 28

6.2 Comandi aggiuntivi 29

7 Codice **31**

7.1 Introduzione alle librerie 31

7.2 Definizione delle variabili 33

7.3 captureThread 36

7.4 processorThread 38

7.4.1 Il sincronismo 40

7.5 playbackThread 41

7.6 Main 42

7.7 Funzioni richiamate nel main 46

7.7.1 Configurazione scheda audio 46

7.7.2 Calcolo dei coefficienti 47

7.7.3 Inizializzazione I2C 50

7.7.4 Lettura dall'ADC 51

7.7.5 Mappature dei segnali 52

8 Risultati sperimentali **55**

8.1 Low-pass e High-pass 55

8.2 Band-pass 57

9 Conclusioni **59**

Bibliografia **61**

Capitolo 1

Introduzione

L'evoluzione della tecnologia ha portato il settore audio a un livello superiore, con l'intervento del digitale nei primi anni '60, la gestione della musica è cambiata sotto ogni punto di vista. In primis qualità sonora, per poi parlare dell'elaborazione (capacità di applicare algoritmi ed effetti complessi), la velocità di registrazione e riproduzione, lo storage di grandi quantità di dati in piccoli supporti molto più affidabili come memorie SSD.
Per entrare nel merito, l'ambito del DSP (digital signal processing) ha potuto prendere piede grazie allo sviluppo di piattaforme tecnologiche capaci di elaborare i dati in maniera veloce, affidabile, con basso costo e manutenzione.
Un DSP (Digital Signal Processor) dunque è un sistema di elaborazione del segnale che, in generale, può avvenire in real-time o offline (post-processing), e può modificare il segnale digitale in più modi come filtri, effetti, compressori, limitatori e molto altro.
Lo scopo di questo studio è quindi, oltre la ricerca del giusto schema hardware adatto allo scopo, l'implementazione di un algoritmo per DSP real-time con filtri parametrici su piattaforma Raspberry Pi, scelta per le grandi performance computazionali unite al basso costo, con relativa facilità di sfruttare sistemi operativi su cui eseguire codice. Il linguaggio di programmazione scelto è il C++, diffusissimo nell'ambito per capacità di controllo e velocità, fondamentali per un algoritmo real-time, così da lasciare un grande margine di sviluppo per aggiornamenti futuri e non rischiare di avere colli di bottiglia adottando linguaggi di più alto livello ma che richiedono sistemi più potenti oltre che più lenti.

Capitolo 2

Stato dell'arte

L'idea della tesi è nata dall'esigenza di implementare dei filtri crossover digitali per un impianto audio custom, dato che c'era il bisogno di mandare a ogni via il giusto range di frequenze. Le generalità dell'ambiente di sperimentazione e applicativo sono: si manda il segnale via Bluetooth dal telefono ad un ricevitore posto nel retro dell'auto, questo lo converte in analogico e lo manda all'ingresso della scheda audio, Raspberry lo processa, lo rimanda in output filtrato (attraverso la scheda audio) e da lì si smista il segnale ai diversi amplificatori così da non avere più bisogno di un filtraggio analogico e tutta la potenza può arrivare agli altoparlanti per un totale di 2 kWrms (4 kW di picco).

Per alimentare lo studio è stata fatta una ricerca preliminare sullo stato dell'arte trovando alcuni articoli affini al progetto. Il primo tra questi è il lavoro svolto da Pasolini, Bazzi e Zabini [1] i quali implementano su piattaforme a basso costo (quali le Raspberry) programmi dsp per software-defined radio (SDR) e signal processing. Il loro setup comprende una Raspberry Pi 2 Model B interfacciata con una scheda audio USB che come per questo studio lavora a 16-bit 48khz; con lo scopo di filtraggio digitale real-time, cancellazione attiva del rumore e sistemi di modulazione per telecomunicazioni.

Un secondo lavoro particolarmente attinente è quello di Ngoc Thy Pham [2], intitolato "Real time audio processing on a Raspberry Pi using partitioned convolution"; viene comunque utilizzata la piattaforma Raspberry Pi 2 Model B implementando un filtro FIR mediante una scheda audio Cirrus Logic [3] utilizzando diverse tecniche di convoluzione (diretta, a blocchi e partizionata).

Il terzo lavoro notato è quello di Noriega-Linares, Navarro Ruiz [4], in cui sviluppano un sistema per il monitoraggio ambientale su scheda a basso costo quale Raspberry Pi 2 Model B. Il dispositivo è in grado di acquisire audio elaborandolo on-board eliminando il problema

di mandare grosse quantità di dati ad un server cogliendo solo le informazioni necessarie dopo il processamento. Il seguente sistema è stato usato anche per una campagna di misurazione reale, dimostrando l'efficacia di questo tipo di approccio.

L'ultimo studio preso in considerazione è quello di Caldas-Morgan, Alvarez-Rosario, Padovese [5], che hanno sviluppato un registratore acustico Low-cost con Raspberry Pi. Acquisiscono il segnale da un idrofono effettuando la conversione tramite lo stesso chip che usa la Behringer UCA222, ovvero il PCM 2902 [6], avendo di conseguenza una performance di 16-bit a 24kHz.

2.1 Obiettivi

A partire dallo stato dell'arte sono stati definiti gli obiettivi del progetto che sono:

- La fattibilità di lavorare con un single-board computer low-cost come la Raspberry per un'elaborazione real-time dei segnali audio.
- L'utilizzo di una scheda audio esterna per semplicità, costo ridotto, e qualità elevata della conversione analogico digitale.
- L'elaborazione Real-time 16-bit a 48kHz, quindi l'acquisizione, l'elaborazione e la restituzione dei dati durante il funzionamento.
- Analizzare i risultati del sistema tramite FFT per stabilire e dimostrare la buona riuscita da un punto di vista oggettivo, e non solo soggettivo in maniera acustica.
- Configurabilità del sistema, quindi possibilità di cambiare i parametri e vedere subito un riscontro nella variazione dell'output.

Capitolo 3

Teoria dei filtri

Un filtro digitale è un algoritmo capace di modificare il contenuto spettrale di un segnale digitalizzato mediante operazioni matematiche.

Le tipologie di filtri digitali sono molteplici. Anche gli stessi filtri sono a loro volta una sottocategoria degli effetti che un codice può implementare per modificare una traccia audio o un segnale di telecomunicazioni. In questo progetto si tratteranno solamente filtri audio, quindi con un raggio d'azione che ricopre lo spettro udibile ovvero da 20 Hz a 20 kHz. In particolar modo il codice dà la possibilità di svolgere filtraggio Low-pass, Band-pass e High-pass con frequenza di taglio (f_c) e comportamento in frequenza (q) variabili [7] [8]. Dal punto di vista matematico un filtro è descritto come una relazione tra il segnale di ingresso $x[n]$ e di uscita $y[n]$, ne consegue che a seconda della relazione utilizzata il filtro avrà un comportamento diverso e uditivamente si percepirà un suono differente (dovuto alle armoniche che saranno attenuate in un determinato modo).

I filtri digitali si dividono in due macro categorie:

- FIR: Finite impulse response, ovvero con risposta impulsiva finita, ciò vuol dire che mandando in ingresso un impulso (delta di Dirac digitalizzata) il sistema risponderà, nel tempo, con una successione finita di valori.

- IIR: Infinite impulse response, questo vuol dire che la sua risposta impulsiva teoricamente si estende indefinitamente nel tempo, pur tendendo a zero per un sistema stabile.

  NB: Risposta infinita non vuol dire coefficienti del filtro infiniti.

La differenza fondamentale tra un filtro FIR e un IIR è proprio nella presenza o meno nell'equazione alle differenze dei termini di retroazione dell'uscita: i coefficienti a_{k}.

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \quad \textit{oppure} \quad H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{M} b_k z^{-k}}{1 + \sum_{k=1}^{N} a_k z^{-k}} \tag{3.1}$$

3.1 Poli, zeri e stabilità del filtro

La funzione di trasferimento del filtro biquad (IIR del secondo ordine) può essere analizzata nel dominio della variabile complessa z se riscritta nella forma (3.2).

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}. \tag{3.2}$$

I coefficienti del numeratore determinano la posizione degli *zeri* della funzione di trasferimento, mentre quelli del denominatore determinano la posizione dei *poli*. Ponendo a zero il denominatore si ottiene l'equazione caratteristica del filtro:

$$1 + a_1 z^{-1} + a_2 z^{-2} = 0 \tag{3.3}$$

Per risolvere l'equazione si può moltiplicare per z^2 entrambi i membri

$$z^2 + a_1 z + a_2 = 0 \tag{3.4}$$

Le due soluzioni dell'equazione rappresentano i poli del filtro e possono essere espresse come

$$p_{1,2} = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2}}{2} \tag{3.5}$$

La posizione dei poli nel piano complesso è strettamente legata al comportamento dinamico del filtro. Per un sistema IIR, discreto e causale, la condizione di stabilità richiede che tutti i poli si trovino all'interno del cerchio di raggio unitario nel piano z.

Questa condizione garantisce che la risposta impulsiva del sistema decada nel tempo. Al contrario, la presenza di un polo sul cerchio unitario o all'esterno porterà rispettivamente ad una risposta non smorzata o divergente. Nel caso del filtro utilizzato nel progetto, i

coefficienti del denominatore vengono determinati a partire dai parametri di progetto, principalmente la frequenza caratteristica f_c e il fattore di qualità q.

La frequenza di campionamento determina inoltre la relazione tra la frequenza analogica desiderata e la frequenza angolare utilizzata nelle equazioni del filtro calcolata come:

$$\omega_0 = 2\pi \frac{f_c}{f_s} \tag{3.6}$$

Un ulteriore parametro fondamentale è il già nominato fattore di qualità q, che influenza la selettività della risposta in frequenza comparendo nel calcolo dei coefficienti RBJ attraverso α

$$\alpha = \frac{\sin(\omega_0)}{2q} \tag{3.7}$$

Una variazione di q modifica quindi i coefficienti del denominatore e, di conseguenza, la posizione dei poli. A parità di frequenza caratteristica, valori maggiori di q producono una risposta più selettiva e una maggiore concentrazione del comportamento del filtro attorno alla frequenza caratteristica; valori inferiori di q producono invece una risposta più smorzata e meno selettiva.

Tabella 3.1: Influenza qualitativa del fattore di qualità q.

q	Selettività	Comportamento della risposta
Basso	Ridotta	Maggiore smorzamento
$\frac{1}{\sqrt{2}}$	Intermedia	Oscillazione massimamente piatta
Alto	Elevata	Maggiore concentrazione attorno a f_c

Nel sistema sviluppato, la variazione dei parametri effettuata dall'utente non modifica quindi la struttura del filtro, ma determina un nuovo insieme di coefficienti. La stessa struttura biquad unita al metodo RBJ può pertanto essere utilizzata per realizzare differenti tipologie di filtraggio, modificando opportunamente numeratore e denominatore della funzione di trasferimento.

Questa caratteristica è particolarmente utile a livello implementativo poiché permette di aggiornare dinamicamente la risposta del filtro a partire dai valori di f_c e di q, mantenendo invariata la struttura dell'algoritmo. Il metodo RBJ costituisce in questo senso il

collegamento tra i parametri scelti dall'utente e i coefficienti utilizzati dall'algoritmo di filtraggio.

3.2 Struttura del biquad

Nell'implementazione del biquad si avranno 5 coefficienti (3 relativi all'ingresso e 2 alle uscite pregresse) normalizzati su a_0, implementati nella seguente formula:

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] - a_1y[n-1] - a_2y[n-2] \tag{3.8}$$

La sua funzione di trasferimento è così descritta dalla formula:

$$H(z) = \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{1 + a_1z^{-1} + a_2z^{-2}} \tag{3.9}$$

Il fatto di avere solamente 5 coefficienti non dice nulla sul comportamento in frequenza, ma il fatto che ce ne siano 2 sulla parte retroattiva bastano a dire che il filtro ha risposta impulsiva infinita, anche se tendente a 0.

La struttura biquad è particolarmente adatta in ambito audio poiché permette di avere un filtro del secondo ordine con un numero limitato di operazioni per campione. Altra caratteristica è che modificando i valori dei coefficienti è possibile avere un diverso comportamento in frequenza (banalmente variando la frequenza di taglio) ma anche del tipo di filtro (Low-pass, Band-pass e High-pass), senza modificare la struttura del filtro, proprietà importante per lo sviluppo di questo software.

La soluzione usata è la "$Transposed\ Direct\ form\ II$", per il motivo che il calcolo di ogni campione richiede la memorizzazione di sole due variabili di stato $s_1[\cdot]$, $s_2[\cdot]$ e i 5 coefficienti. La stessa funzione di trasferimento può essere implementata mediante diverse forme, come si evince in Orfanidis [9]. Le più comuni sono la "$Direct\ form\ I$", la "$Direct\ form\ II$" e le loro relative trasposte. La struttura è così descritta dalle seguenti equazioni:

$$y[n] = b_0x[n] + s_1[n-1] \tag{3.10}$$

$$s_1[n] = b_1 x[n] - a_1 y[n] + s_2[n-1] \tag{3.11}$$

$$s_2[n] = b_2 x[n] - a_2 y[n] \tag{3.12}$$

3.3 Aggiornamento dei coefficienti con metodo RBJ

Per il calcolo dei coefficienti del filtro, che, come visto, si modificano oltre che nella frequenza di taglio anche nel comportamento in frequenza, settato dall'utente, si è deciso di utilizzare il metodo di calcolo proposto da Robert Bristow-Johnson [10]. RBJ fornisce delle formule dirette e semplici così che possano essere usate anche più volte nel codice senza appesantire troppo il carico computazionale, partendo dai parametri principali del filtro quali f_s, f_c e q.

Per prima cosa si calcola la frequenza angolare digitale, ω_0, ovvero la posizione dove si colloca la f_c nello spettro digitale, che con $f_\mathrm{s} = 48kHz$ ha come limite superiore (secondo Nyquist) 24kHz.

$$\omega_0 = 2\pi \frac{f_c}{f_s} \tag{3.13}$$

Invece q entra in gioco nel calcolo di α, parametro che stabilisce la selettività del filtro e come effetto secondario può dare delle piccole oscillazioni nell'intorno della frequenza di taglio quando il filtro lavora con slope al limite. α è così definita:

$$\alpha = \frac{\sin(\omega_0)}{2q} \tag{3.14}$$

RBJ stabilisce questi coefficienti per le seguenti configurazioni:

3.3.1 Low-pass filter

$$b_0 = \frac{1-\cos(\omega_0)}{2(1+\alpha)}, \qquad b_1 = \frac{1-\cos(\omega_0)}{1+\alpha}, \qquad b_2 = \frac{1-\cos(\omega_0)}{2(1+\alpha)} \tag{3.15}$$

$$a_1 = \frac{-2\cos(\omega_0)}{1+\alpha}, \qquad a_2 = \frac{1-\alpha}{1+\alpha} \tag{3.16}$$

3.3.2 High-pass filter

$$b_0 = \frac{1+\cos(\omega_0)}{2(1+\alpha)} \quad b_1 = \frac{-(1+\cos(\omega_0))}{1+\alpha} \quad b_2 = \frac{1+\cos(\omega_0)}{2(1+\alpha)} \tag{3.17}$$

$$a_1 = \frac{-2\cos(\omega_0)}{1+\alpha} \quad a_2 = \frac{1-\alpha}{1+\alpha} \tag{3.18}$$

3.3.3 Band-pass filter

$$b_0 = \frac{\sin(\omega_0)}{2(1+\alpha)} \quad b_1 = 0 \quad b_2 = \frac{-\sin(\omega_0)}{2(1+\alpha)} \tag{3.19}$$

$$a_1 = \frac{-2\cos(\omega_0)}{1+\alpha} \quad a_2 = \frac{1-\alpha}{1+\alpha} \tag{3.20}$$

Capitolo 4

Descrizione dell'Hardware

4.1 Informazioni generali

Il primo passo è stato individuare il corretto sistema operativo da installare sulla Raspberry, inizialmente era stato utilizzato "ELK-audio OS", che però non dava accesso a schede audio non proprietarie. Per questo motivo, avendo come obiettivo la creazione di un sistema open source e totalmente manipolabile, si è optato per Pi OS, comunque un OS Linux-kernel che però, anche se meno ottimizzato per l'audio, consentiva ampio margine di manovra per lo scopo [11]. Lo scopo è stato raggiunto usando una scheda di sviluppo Raspberry Pi 4 [12], quad-core 64-bit 1.8 GHz, le caratteristiche del processore interessano maggiormente poiché è il cuore di un'elaborazione audio real-time. Si è scelto in primo luogo di utilizzare una scheda audio esterna, la Behringer U-CONTROL UCA-222 [13], che svolga il compito di ADC-DAC in modo da concentrarsi sullo sviluppo software, l'obiettivo futuro è progettare autonomamente la suddetta scheda per ottimizzare dei passaggi, adattando così in modo migliore le periferiche, migliorando notevolmente latenza, bit-rate e risoluzione. I segnali che trasportano le informazioni dei parametri sono conferiti alla Raspberry in due modi: Segnali analogici, modulo ADS1115 [14], che è un adc 4 canali che comunica tramite I2C; Segnali Digitali, dip-switch in pull-down in ingresso a pin digitali GPIO.

4.2 Raspberry Pi-4

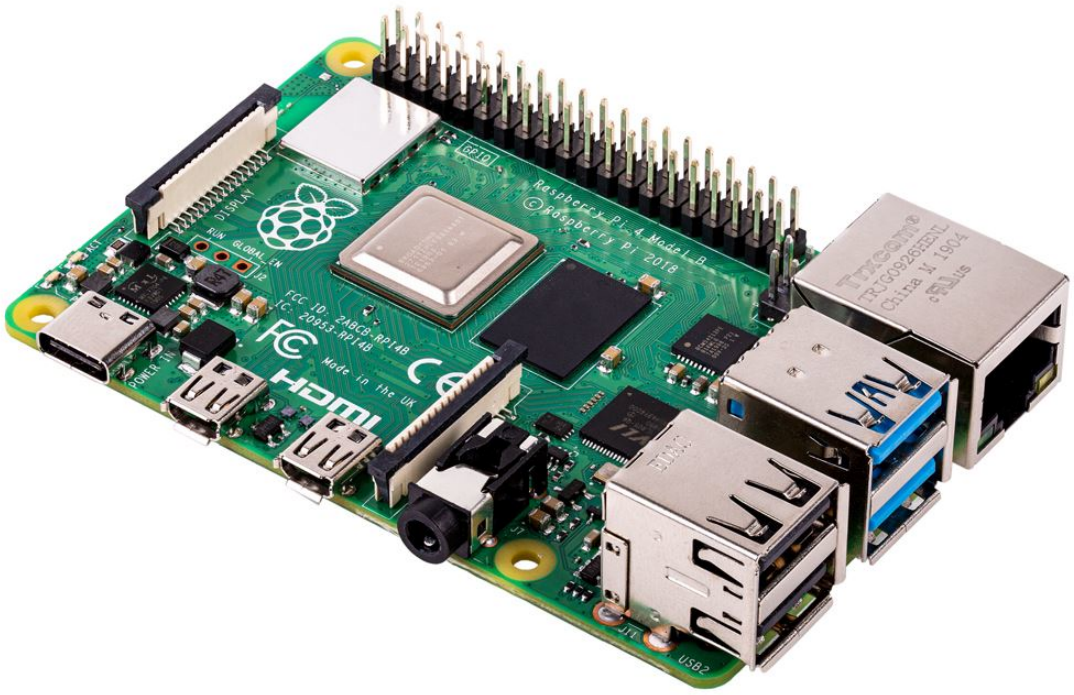


Figura 4.1: Raspberry Pi 4

La scheda Raspberry Pi 4 Model B mostrata in figura 4.1 è un computer a scheda singola con processore ARM Cortex-A72 quad-core 64-bit, equipaggiato in questo caso con 2GB di memoria RAM LPDDR4. Per l'interfacciamento dispone di una piedinatura 40 pin in grado di consentire comunicazioni I2C, SPI, UART oltre che GPIO per controllo digitale. Per l'elaborazione audio si usano anche le porte USB dove è connessa la scheda audio, quindi il segnale in ingresso e in uscita transitano attraverso le porte USB-A 3.0 disponibili, mentre per la gestione ci si affida a I2C e GPIO. Il basso costo della scheda e la grandissima capacità di manipolazione dei dati ad alta efficienza mediante OS Linux-kernel rendono questa piattaforma perfetta per l'applicazione, con alte capacità di scalabilità, controllo a basso livello e prestazioni.

4.3 Behringer UCA-222



Figura 4.2: Behringer UCA222

La scheda audio Behringer U-CONTROL UCA-222 mostrata in figura 4.2 è un dispositivo di interfacciamento audio USB utilizzato per l'acquisizione e la riproduzione di audio digitale. Il dispositivo integra un'interfaccia USB con due canali di ingresso e due canali di uscita, consentendo di effettuare la conversione tra i domini analogico e digitale senza richiedere un'interfaccia audio dedicata sulla Raspberry. Consente l'acquisizione

e la riproduzione di segnali audio stereo, con risoluzione di 16-bit e una frequenza di campionamento fino a 48 kHz. Il cavo USB-A consente sia l'alimentazione sia l'interfaccia con il processore utilizzato, comunicando con le librerie ALSA [15] (Advanced Linux Sound Architecture) per trasmettere i segnali digitali. Nel sistema realizzato, il segnale acquisito dalla scheda viene quindi reso disponibile al programma attraverso il dispositivo ALSA, dove i campioni vengono elaborati dall'algoritmo DSP prima di essere inviati nuovamente alla periferica per la riproduzione. Nel progetto in questione svolge il ruolo di interfaccia tra l'ingresso audio RCA analogico e l'uscita RCA analogica, permettendo l'elaborazione in digitale da parte dell'algoritmo. Tra i parametri tecnici considerati nella progettazione rientrano l'assorbimento massimo di 100 mA gestibile dalla Raspberry, valore di picco in ingresso e uscita pari a 2 dBV e una risposta in frequenza piatta con un'incertezza di ± 1 dB su tutto lo spettro udibile. Il tutto è riportato nel datasheet del dispositivo [13].

4.4 ADS1115

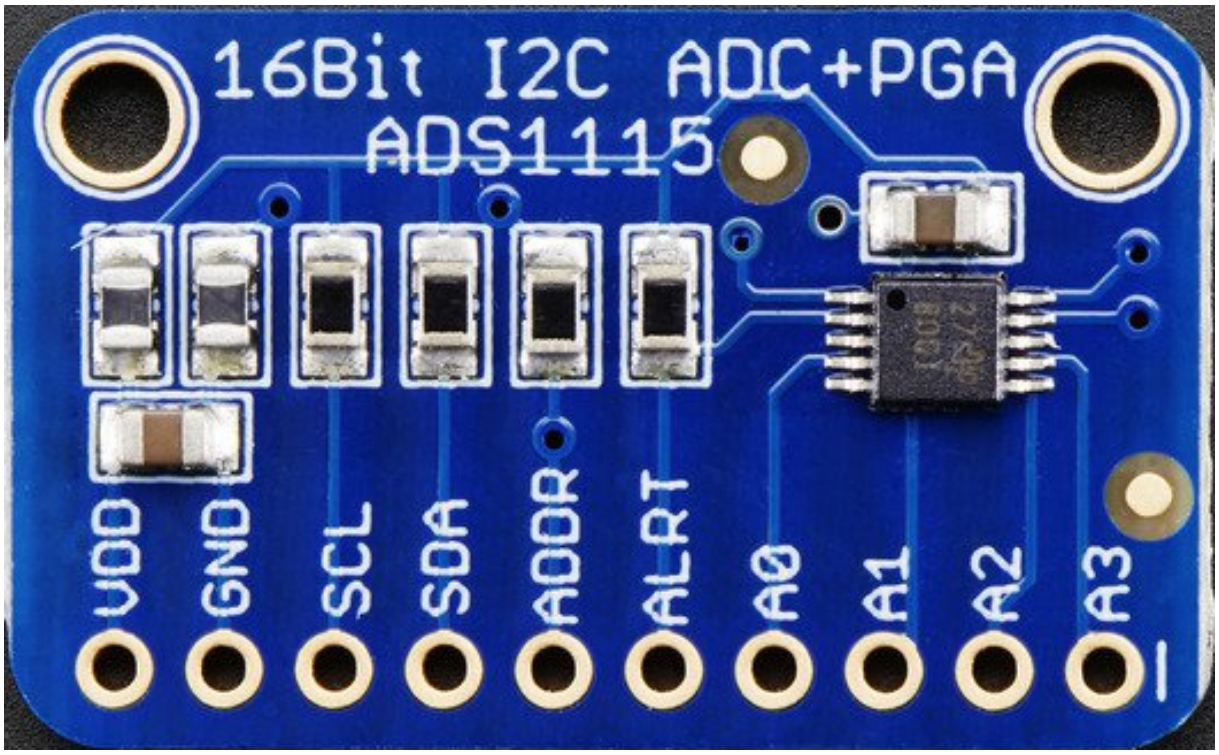


Figura 4.3: ADS1115

L'ADS1115 [14] mostrato in figura 4.3 è un chip prodotto dalla Texas Instruments che svolge il ruolo di ADC+PGA comunicando in I2C con la Raspberry. Nel progetto si utilizza una scheda che aggiunge al chip dei resistori di pull-up per il bus e un condensatore di livellamento per l'alimentazione. L'ADC è un 16-bit a 4 ingressi con architettura Sigma-Delta, affiancata a un PGA, ovvero un amplificatore interno che permette di gestire il range di misura. Le funzionalità utilizzabili sono:

- Single ended (4 canali indipendenti) o in differential mode (per i 4 segnali in ingresso i 2 valori corrispondenti alla differenza tra coppie)
- Il single shot (si invia una richiesta di conversione unica) oppure continuous mode (l'ADC converte sempre e viene letto il buffer)
- le velocità di conversione vanno ponderate a seconda del rapporto affidabilità/velocità e sono disponibili valori da 8SPS a 860SPS (Sample-per-second)

I range di misura disponibili sono riportati nella Tabella 4.1.

Tabella 4.1: Intervalli di misura selezionabili tramite il PGA dell'ADS1115.

PGA2:0	**Intervallo di misura**
000	±6.144 V
001	±4.096 V
010	±2.048 V
011	±1.024 V
100	±0.512 V
101	±0.256 V

Tabella 4.2: Velocità di conversione configurabile tramite i bit DR dell'ADS1115.

DR2:0	**Data rate**
000	8 SPS
001	16 SPS
010	32 SPS
011	64 SPS
100	128 SPS
101	250 SPS
110	475 SPS
111	860 SPS

Per la programmazione della velocità di conversione si fa riferimento alla Tabella 4.2.

Il setup usato per il progetto è single ended, single shot, 128 SPS, ±4.096 V.
Avendo alimentato la scheda direttamente dalla Raspberry, la massima tensione applicabile agli ingressi analogici sarà limitata a 3.3 V, ciò comporta che il range selezionato avrà una zona scoperta che limita l'ENOB dell'ADC, dovendo quindi correggere tramite una mappatura lineare la lettura del valore per rispettare l'informazione del sensore di posizione.

Nel progetto si è usato anche il pin ALERT che consente di avere una condizione del segnale che varia quando la conversione è terminata, evitando di implementare delay fissi riducendo l'efficienza generale del codice (questa parte verrà poi approfondita nel capitolo 7). Per leggere questo segnale Raspberry utilizza uno dei suoi bit GPIO, riempiendo la memoria locale con il valore del buffer dell'adc solo quando il segnale va basso (condizione di fine conversione). Gli approfondimenti sull'utilizzo nel codice sono rimandati all'apposita sezione 7.7.4, i dati sono stati presi dal datasheet [14]

4.5 Schemi di collegamento

4.5.1 Audio

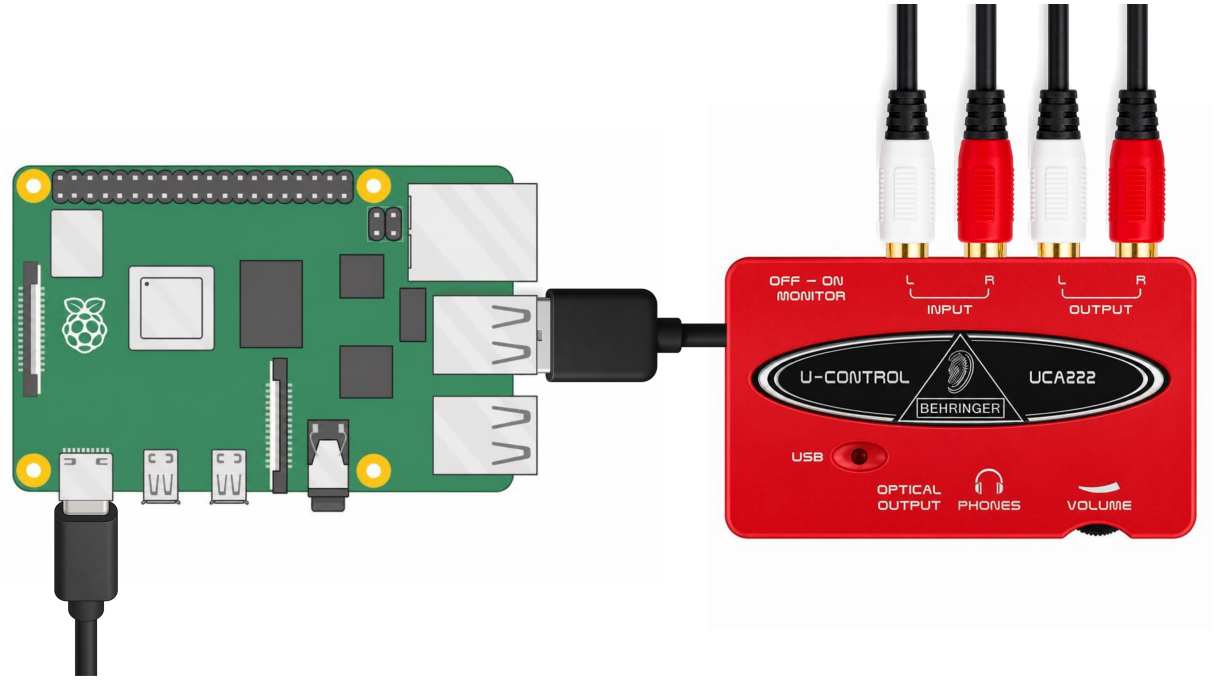


Figura 4.4: Schema di collegamento delle tracce audio

La figura 4.4 mostra lo schema di collegamento audio, partendo dalle alimentazioni si può notare che la scheda audio sarà alimentata tramite la porta USB-A direttamente dalla Raspberry, che a sua volta sarà alimentata mediante un cavo USB-C dedicato solo a questo scopo. L'alimentatore che consiglia e fornisce casa madre ha come specifiche 5 V a 5 A per un totale di 25 W di potenza che è il picco di consumo che viene dichiarato per la scheda Raspberry Pi 4, per il progetto non disponendo di questo componente è stato utilizzato un caricatore per telefono che garantisce 6 A a 5 V quindi comunque sopra soglia. Per l'utilità finale del DSP, trovandosi in auto, si è predisposto un DC-DC converter step-down che viene alimentato dalla batteria dell'auto ed è in grado di erogare una corrente di 10 A. Detto ciò si passa ad analizzare il percorso del segnale partendo dalla sorgente, si suppone di avere una sorgente di audio analogico (che viaggiando su cavo RCA risulta sbilanciato) che rispetti i parametri di ingresso della scheda, nel caso considerato l'unico limite è la tensione di 2 dBV ($\approx$ 1.26 V) considerabile valore di fondo scala "0 dBFS", mentre la scheda offre un'impedenza di ingresso pari a 27 kΩ. Per impianti

car audio le tensioni sui cavi RCA possono salire fino a 2.5 Vrms per ridurre il rumore di fondo causato dall'ambiente elettromagneticamente rumoroso dell'impianto. Il segnale viene così campionato, quantizzato e codificato dalla Behringer che lo trasmette in formato digitale tramite un CODEC PCM [6] (Pulse Code Modulation), una modulazione lossless senza compressione. Questa codifica è la più semplice che si possa utilizzare, un grande pregio di cui ci si può servire lavorando così a basso livello con i driver ALSA di linux. Il tutto avviene a 48 kHz 16-bit tramite lo stesso cavo USB-A usato per l'alimentazione. Dopo l'invio, entro 5.33 ms (ovvero la durata di un PERIOD del buffer implementato) il segnale processato è già tornato alla scheda tramite PCM, dove stavolta verrà convertito in analogico sullo stesso range di ingresso. Ovviamente tramite codice è possibile riconoscere se il valore del frame in un istante supera il fondo scala ($\pm$32768 ovvero i 16-bit signed) di conseguenza ci si accorge del clipping e tramite un LED lo si segnala in output ogni qualvolta si verifichi questa condizione di distorsione. Parametro importante da tenere sotto controllo per i sistemi digitali è la latenza, che in questo caso dovendo solo ascoltare un segnale musicale non è così rilevante. Nonostante ciò, il sistema ha una latenza inferiore ai 20 ms che è il valore limite considerato in letteratura oltre il quale si inizia a percepire la latenza.

4.5.2 Analog input

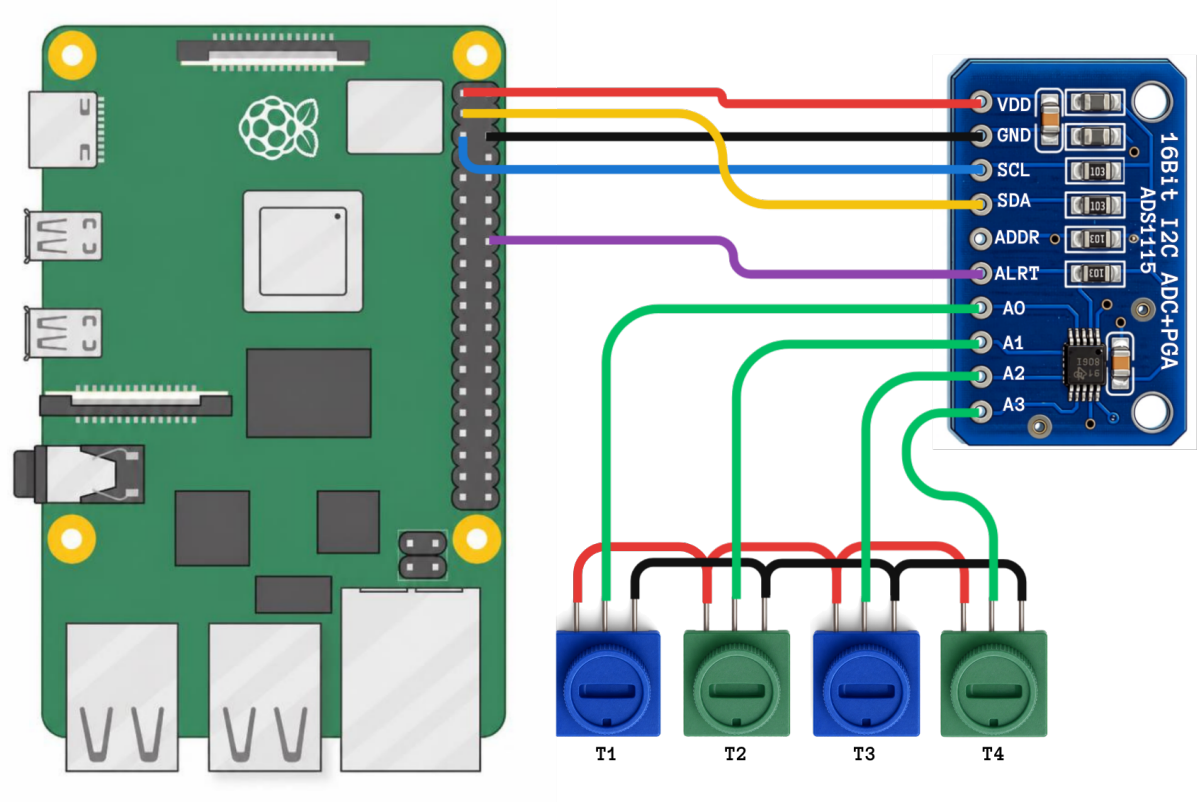


Figura 4.5: Schema di collegamento degli ingressi analogici

In questa sottosezione viene spiegato lo schema 4.5 di collegamento degli ingressi analogici, l'utente con questo sistema può cambiare i parametri più importanti per l'ascolto audio (quali volume e frequenza di taglio) con continuità attraverso dei trimmer (anche detti trasduttori di posizione) che mediante l'azione meccanica di rotazione producono una variazione della resistenza, in questo modo attraverso un dedicato circuito di condizionamento restituiscono un valore in tensione proporzionale alla posizione angolare stabilita dall'utente. Per portare questa informazione all'algoritmo filtrante è necessario costruire una serie di passaggi per fare in modo che tutti i blocchi riescano a capirsi tra loro; in primo luogo il circuito di condizionamento che consiste nel collegare VCC (3.3 V) e GND sui reofori esterni così che il pin centrale restituisca invece che una variazione di resistenza, una variazione di tensione, che è possibile leggere con il modulo ADS1115. Il chip comunica con la Raspberry tramite protocollo I2C, noto per la sua semplicità e robustezza oltre che alla grande diffusione per applicazioni del genere in questo settore.

Le particolarità di questa comunicazione nonché il suo più grande pregio è l'utilizzo di un bus con soli 2 fili (data e clock) questo consente adattabilità e facilità nella maggior parte dei contesti, aiutato dal fatto che si possono usare molti slave mediante l'addressing software pilotato dal master I2C. La comunicazione è sincrona half-duplex ovvero il clock è scandito dal master e i dati possono viaggiare in entrambi i versi ma solo uno alla volta. Per assicurarsi che il byte sia stato ricevuto il chip di destinazione manda sempre un bit basso (acknowledge) che viene letto dalla scheda mittente per controllare appunto che sia avvenuta la lettura, si potrebbero dire molte altre cose su questo protocollo ma esulerebbe dallo scopo del progetto. Nell'attuale configurazione impostata da codice, l'ADC manda sul bus il valore della conversione dei 4 ingressi analogici indipendenti, con una velocità di 128 SPS e un valore di fondo scala pari a 4.096 V, tutto questo ogni volta che l'algoritmo effettua la chiamata per conoscere il valore dell'ingresso specificato. I pin sono riportati nella seguente Tabella 4.3

Tabella 4.3: Collegamento dei pin utilizzati

Pin Raspberry Pi	**Segnale**	**Funzione**
1	VCC	Alimentazione 3.3 V
3	SDA	Linea dati I^2C
5	SCL	Clock I^2C
6	GND	Massa
16	GPIO23	ALERT/RDY

In conclusione si anticipa che una problematica riscontrata è quella di un rumore che produce delle oscillazioni sul valore letto dai trimmer, si è risolto adottando un filtraggio software che verrà approfondito nella parte di codice sezione 7.7.5.

4.5.3 Digital input

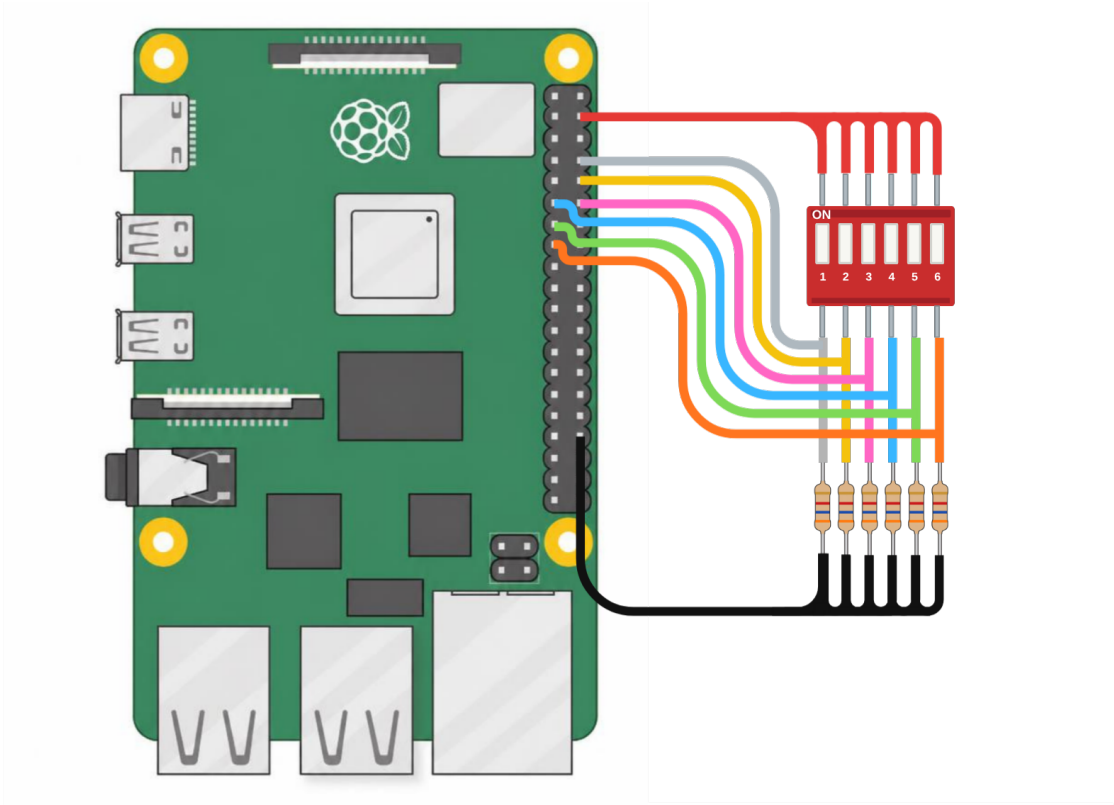


Figura 4.6: Schema di collegamento degli ingressi digitali

Considerando lo schema 4.6 rappresentante gli ingressi digitali, che pilotano i parametri quali tipo del filtro e comportamento in frequenza, vengono implementati mediante un DIP-Switch modulo 6 configurato in pull-down, ovvero alimentato nella parte superiore a VCC e nella parte inferiore portato a massa da una resistenza per ogni interruttore. Il principio di funzionamento di questa configurazione è quello di prelevare il segnale come tensione ai capi della resistenza, in questo modo quando l'interruttore è OFF si noterà che la R è flottante e collegata a massa quindi la differenza di potenziale ai suoi capi è nulla, mentre chiudendo lo switch nel circuito scorrerà corrente generando una caduta di tensione sul resistore di shunt leggendo così il valore HIGH. I pin sono elencati nella Tabella 4.4.

Tabella 4.4: Collegamento dei pin digitali utilizzati

Pin Raspberry	**GPIO**	**Funzione**
8	GPIO14	$q = 0.1$
10	GPIO15	$q \approx 0.707$
11	GPIO17	$q = 0.99$
12	GPIO18	$type_f = LOW$
13	GPIO27	$type_f = BAND$
15	GPIO22	$type_f = HIGH$

Capitolo 5

Architettura e funzionamento del DSP

5.1 Struttura logica del sistema

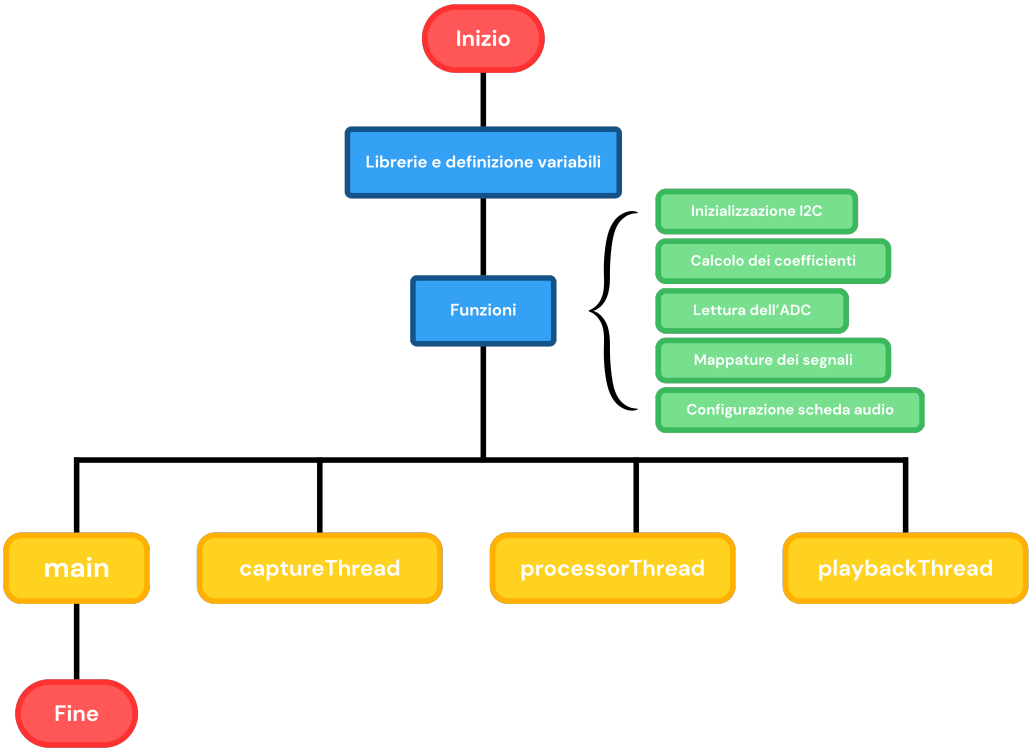


Figura 5.1: Struttura logica del sistema

La figura 5.1 rappresenta il flowchart che esegue la logica di questo algoritmo DSP, dopo la parte introduttiva delle librerie, si inizializzano le variabili e le costanti globali, usate tra le funzioni durante tutta l'esecuzione del programma. Seguono così le funzioni che sono accessorie al main e che verranno richiamate in vari punti del codice per una migliore organizzazione logica.

5.2 Elaborazione audio a blocchi

La catena audio prevede una frequenza di campionamento di 48 kHz, questo vuol dire che un frame (array contenente sx e dx del segnale stereo) arriva ogni 20.8 us, per elaborarlo digitalmente va acquisito, processato e riprodotto; per aumentare l'efficienza del sistema evitando di eseguire queste 3 operazioni per ogni singolo frame si ricorre alla suddivisione in blocchi. Viene raggruppato così un input che è composto da n frame (nel caso 256) che diventerà poi il blocco processato, infine il blocco di output; si gestisce così in maniera strutturata il segnale audio riducendo di molto il carico computazionale. Questo inoltre è particolarmente efficace con lo scheduling dei thread, argomento che si tratterà nella sezione 5.3. Avendo un blocco (PERIOD) di 256 campioni la latenza in uscita non sarà inferiore ai 5.33 ms poiché

$$T_{\text{lat}} = \frac{period_size}{f_s} = \frac{256}{48000} \approx 5.33\,\text{ms} \tag{5.1}$$

Per assicurarsi che il sistema abbia ridondanza si conservano temporaneamente i PERIOD in un BUFFER tale che buffer_size sia multiplo intero di period_size (nel caso 4) quindi:

$$T_{\text{buffer}} = \frac{buffer_size}{f_s} = \frac{1024}{48000} \approx 21.33\,\text{ms} \tag{5.2}$$

Nel sistema si possono distinguere due buffer, buffer_raw e buffer_proc della stessa grandezza, che conterranno campioni a livelli differenti di elaborazione. Per spiegare brevemente come si dimensionano le dimensioni di BUFFER e PERIOD in modo da gestire la latenza il metodo è quello di fare un classico trade-off: un PERIOD piccolo comporta sì minore latenza ma anche maggiore overhead (ovvero la frequenza di gestione delle operazioni), un PERIOD grande al contrario aumenta la latenza ma la gestione delle risorse e il carico computazionale sono molto più basse ed efficienti.

5.3 Threading

La parte più corposa di codice riguarda le funzioni audio denominate captureThread, processorThread e playbackThread. Queste funzioni devono essere eseguite ininterrottamente (poiché responsabili del comportamento real-time [16]), senza essere mai disturbate, nel minor tempo possibile; in questo caso la teoria dello scheduling dei thread è un'alleata importante. Si vuole così parallelizzare, ovvero sovrapporre temporalmente più esecuzioni, il comportamento del codice eseguendo più funzioni contemporaneamente (in questo caso acquisizione, processamento e riproduzione) oltre al main che gestisce i parametri, al quale però già è riservato un thread all'avvio senza necessità di scheduling [17]. In questo modo si sfruttano tutti e 4 i core della CPU di Raspberry migliorando l'efficienza complessiva del sistema (che ingegneristicamente parlando è un grande risultato) al solo costo della maggiore complessità nella gestione. Suddividere il flusso in 3 attività implica che a ogni thread è assegnata un'attività specifica, il carico computazionale che prima era assegnato ad una sola esecuzione ora è ripartito su più thread permettendo al processore di sfruttare in modo migliore le risorse disponibili, lasciando maggiore margine per gestire eventuali variazioni dei tempi di esecuzione. Questo è evidente se si verifica un rallentamento in una fase, gli altri blocchi possono continuare a lavorare senza interrompere il flusso. Il guadagno in velocità si nota anche guardando al tempo di esecuzione di un blocco, che nel caso sequenziale impiegherebbe tutte e 3 le funzioni, mentre con lo scheduling dura una funzione sola dato che le altre due vengono svolte in parallelo (Figura 5.2).

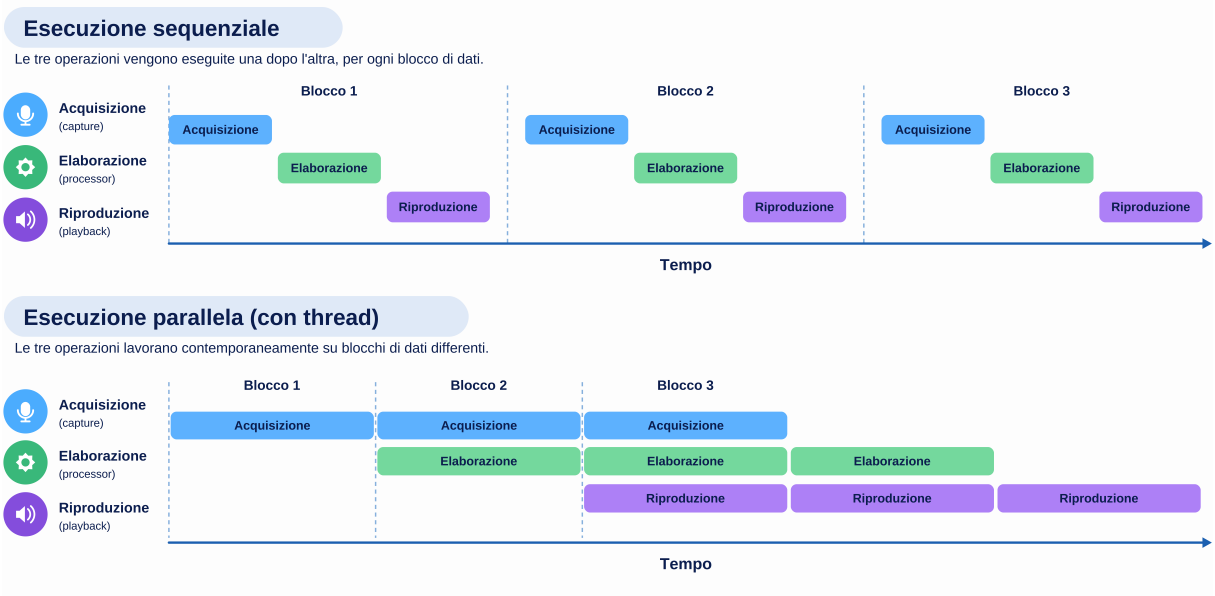


Figura 5.2: Differenza tra processi in serie e in parallelo

Capitolo 6

Installazione e configurazione di Pi OS

In primo luogo bisogna installare il sistema operativo sulla scheda micro-SD destinata alla Raspberry, tramite un altro computer si deve installare il Raspberry Pi Imager (la schermata principale è raffigurata nella Figura 6.1), qui è possibile attraverso le varie pagine configurare al meglio secondo le specifiche volute il proprio OS.

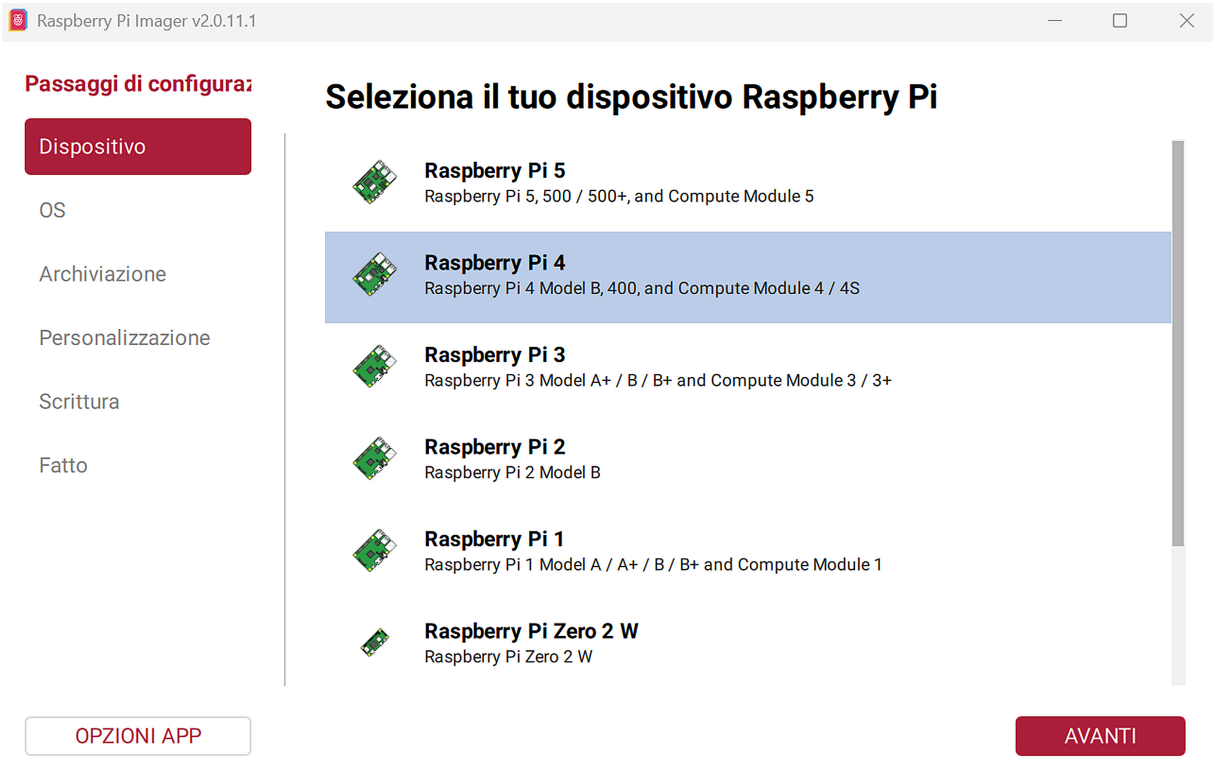


Figura 6.1: Ambiente di installazione del Sistema Operativo

Per il DSP si adotterà Pi OS Lite, che è la versione più leggera senza GUI (Graphical User Interface) dato che alla fine il programma verrà eseguito automaticamente e gli interventi si possono fare tranquillamente da terminale conoscendo i comandi di un qualsiasi OS

Linux-kernel [11]. Inoltre si può connettere preventivamente a una rete Wi-Fi in modo che non ci sia il bisogno di collegare il cavo HDMI per la configurazione, basta il collegamento tramite SSH da un altro PC.

Una volta connessi tramite SSH si può cominciare a configurare il dispositivo per accogliere il DSP. I primi comandi che si eseguono sono:

```
sudo apt update
sudo apt upgrade
sudo apt install libasound2-dev
```

Queste righe aggiornano l'elenco delle versioni dei pacchetti disponibili, aggiornano all'ultima versione i pacchetti, e installano il pacchetto per poter utilizzare correttamente le librerie ALSA. Ora, siccome per lo sviluppo si utilizza Visual Studio Code per permettere la compilazione includendo la libreria bisogna andare nella cartella ".vscode" e nel file tasks.json inserire come args "-lasound". In più per funzionare il codice ha bisogno dell'accesso al bus I2C che va abilitato (per la comunicazione con l'ADC), con i seguenti comandi:

```
raspi-config
interface options
i2c
yes
```

6.1 Avvio automatico

L'utilità finale del codice prevede l'avvio automatico quando la scheda viene alimentata, in modo da essere autonoma e cominciare a processare semplicemente accendendola. Questo si ottiene creando un file in system con privilegi di amministratore (sudo) contenente un service, ovvero il servizio che eseguirà il codice:

```
/etc/systemd/system
sudo nano dsp.service
[Service]
```

```
4 ExecStart=/home/tesi/main
5 [Install]
6 WantedBy=multi-user.target
7 sudo systemctl daemon-reload
8 sudo systemctl enable dsp
```

Le ultime due righe di codice dicono a *systemd* di ricontrollare i *.service* in modo da attivare subito il servizio (enable).

6.2 Comandi aggiuntivi

Verrà elencata una serie di comandi utili alla gestione del sistema, anche come debug, in modo da gestire sempre da terminale quello che sta succedendo ed agire di conseguenza.

- `htop` visualizza in tempo reale l'utilizzo della CPU mostrando anche i vari processi attivi e quale processo sta gravando sul singolo core, funzione molto utile per questo scopo. È possibile visualizzare anche la RAM utilizzata e la distribuzione di carico computazionale.

- `watch vcgencmd measure_temp` è utile per tenere sotto controllo la temperatura, in caso di un alto carico computazionale si può vedere lo storico delle temperature e magari associarlo a dei glitch causati dal thermal throttling, che è un meccanismo per cui la CPU in caso di surriscaldamento abbassa la sua frequenza di clock e nel caso di un DSP real-time genera glitch molto fastidiosi.

- `aplay -l` elenca le schede audio e i dispositivi visibili ad ALSA, informa anche il numero della porta, fondamentale da inserire nel codice nella sezione 7.7.1

- `sudo journalctl -u dsp -f`, nel codice sono stati inseriti degli output sul terminale, in caso di errori o semplicemente ci si possono plottare informazioni come le frequenze di taglio attuali, con questo comando si ha accesso all'elenco (anche allo storico) degli output del codice dall'inizio dell'esecuzione.

- `g++ main.cpp -o main -lasound`, in caso di veloci modifiche al codice, si può ricompilare senza interfaccia VS Code tramite questo comando

Capitolo 7

Codice

7.1 Introduzione alle librerie

```
1  // Audio-ALSA
2  #include <alsa/asoundlib.h>
3  #include <stdint.h>
4
5  //Threading e sincronizzazione
6  #include <thread>
7  #include <atomic>
8
9  //Elaborazione numerica
10 #include <algorithm>
11 #include <cmath>
12
13 //Gestione GPIO e plot su terminale
14 #include <iostream>
15 #include <fstream>
16
17 //I2C e ADC
18 #include <fcntl.h>
19 #include <unistd.h>
20 #include <sys/ioctl.h>
21 #include <linux/i2c-dev.h>
22 #include <stdio.h>
```

Breve descrizione delle librerie utilizzate:

- <asoundlib.h> per gestire l'acquisizione e riproduzione dei campioni con ALSA
- <stdint.h> definizione dei tipi di formato con cui ALSA manda i campioni
- <thread> creazione e gestione dei thread
- <atomic> permette di generare un tipo di variabile che può essere condivisa tra i thread senza errori logici
- <algorithm> funzione clamp per gestire gli overflow
- <cmath> funzioni matematiche usate nel filtro e nella mappatura dei valori dell'ADC
- <iostream> visualizzazione sul terminale di messaggi quali errori, informazioni sullo stato del codice e ADC
- <fstream> lettura stato GPIO dalla cartella di sistema
- <fcntl.h> utilizzato per aprire il dispositivo I2C con open()
- <unistd.h> comunicazione con l'ADC e gestione dei ritardi con usleep()
- <ioctl.h> configurazione dell'indirizzo dello slave I2C
- <i2c-dev.h> accesso all'interfaccia I2C-Linux e definizione parametri I2C
- <stdio.h> segnalazione di errori all'apertura della comunicazione I2C

7.2 Definizione delle variabili

```
constexpr double PI = 3.14159265358979323846;
// ADC
#define ADS1115_ADDR 0x48
#define REG_CONVERT 0x00
#define REG_CONFIG  0x01
int fd;
int adc0;
int adc1;
int adc2;
int adc3;
int last_adc0;
int last_adc1;
int last_adc2;
int last_adc3;

// Parametri audio
#define CH_IN 2
#define CH_OUT 2
#define PERIOD 256
#define BUFFER_SIZE 1024
#define SAMPLE_RATE 48000

//GPIO INTERFACE
int GPIO23; //interrupt conv
int GPIO24; //Clipping LED
int GPIO14; //low-pass
int GPIO15; //high-pass
int GPIO18; //band-pass
int GPIO17; // q low
int GPIO27; // q mid
int GPIO22; // q high

```

```
33 // Gain
34 double gain_dx = 1.0;
35 double gain_sx = 1.0;
36
37 //Filtro
38 char type_f;
39 int fc_dx;
40 int fc_sx;
41 double q = 0.7071067811865476;
42
43 //Strutture
44 struct Frame {
45     int16_t L;
46     int16_t R;
47 };
48 struct State {
49     double L;
50     double R;
51 };
52 struct biquad {
53     double b0;
54     double b1;
55     double b2;
56     double a1;
57     double a2;
58 };
59 biquad coeff_sx;
60 biquad coeff_dx;
61
62 //Stati del filtro biquad
63 State Y1;
64 State Y2;
65
66 //Variabili atomiche
67 std::atomic<bool> coeff_upt_flag = false;
```

```
68 std::atomic<Frame> bufferRaw[BUFFER_SIZE];
69 std::atomic<Frame> bufferProc[BUFFER_SIZE];
70 std::atomic<short> i = 0;
71 std::atomic<short> h = 0;
72 std::atomic<int> posRaw = 0;
73 std::atomic<int> posProc = 0;
```

- L'ADC ha come indirizzo per il protocollo I2C 0x48, è composto da 4 registri differenti e indipendenti, da cui si prenderanno i valori della conversione.

- Parametri per la scheda audio, quali numero di canali in ingresso e in uscita, grandezza del buffer, il sample rate e in quanti periodi è suddiviso (sempre un sottomultiplo).

- I GPIO servono sia per il controllo di fine conversione dell'ADC sia come ingresso per l'informazione delle forme dei filtri.

- I gain dei due canali sono double per avere una minore approssimazione durante i calcoli, e sono controllabili indipendentemente dai trimmer.

- I tipi di filtro selezionabili dagli switch sono low-pass, high-pass e band-pass, con frequenze di taglio variabili in real-time.

- Le strutture servono per contenere i sample in diversi punti del codice accorpandoli in un unico tipo di dato.

- Le variabili atomiche vengono utilizzate quando più thread devono accedere contemporaneamente alla stessa variabile. L'utilizzo di questo tipo di variabile è necessario quando si implementa il threading, queste infatti garantiscono l'impossibilità che si verifichi una race condition. La race condition, in generale, è una situazione nella quale il risultato di un' operazione dipende dal timing con il quale il thread svolge l'operazione, ad esempio nel caso in cui si voglia incrementare una variabile, entrambi i thread al momento antecedente l'operazione leggono lo stesso valore, scrivendo quello successivo, quando in realtà ci si aspetterebbe alla fine di entrambi i processi l'incremento di due unità invece che una.

7.3 captureThread

```
1  void captureThread(snd_pcm_t* capture) {
2      usleep(100000);
3      Frame input[PERIOD];
4      while (true) {
5
6          for( i ; i < BUFFER_SIZE/PERIOD; i.store(i=i+1)){
7              int frames = snd_pcm_readi(capture, input, PERIOD);
8              // Controllo errori di lettura
9              if (frames ==-EPIPE) {
10                 std::cout << "Error: buffer overrun\n";
11                 snd_pcm_prepare(capture);
12                 continue;
13                 } //EPIPE vale 32
14
15             else if (frames != PERIOD) {
16                 std::cout << "short read error\n";
17                 snd_pcm_prepare(capture);
18                 continue;
19             }
20             // Acquisizione dei campioni e salvataggio nel bufferRaw
21             for ( int k = 0; k < PERIOD; k++) {
22                 bufferRaw[i*PERIOD+k].store(input[k]);
23                 posRaw.store(i*PERIOD+k);
24             }
25         }
26         i.store(0);
27     }
28 }
```

La funzione captureThread associata al thread t1 (vedi sezione 5.3) svolge la fase di acquisizione della catena audio, viene richiamata passando il parametro dell'handler stabilito nella sezione 7.7.1. Si chiama per prima cosa un ritardo di 100 ms, questo viene

svolto progressivamente in cascata su ogni thread per permettere al main di avviarsi e fare le prime letture in modo che i primi campioni letti seguano i parametri correttamente processati nelle prime fasi. Inizializzo una variabile di tipo frame per contenere i valori locali. Inizia così il loop infinito che eseguirà il thread, il for serve per scandire l'andamento dei PERIOD nel frame buffer (in questo caso 4 iterazioni visto che il period è lungo 256 frame e il buffer 1024). Il primo comando è il più importante, `snd_pcm_readi()` è una delle principali funzioni di ALSA per leggere campioni da una scheda audio, ha bisogno di 3 parametri: l'handler che è una variabile che gestisce tutti i parametri e le specifiche di come si vuole che avvenga l'acquisizione, come ad esempio il fatto che bisogna avere i campioni in ordine interleaved (i canali entrano alternati nel buffer). `input` è il frame da riempire, mentre PERIOD è la grandezza del buffer da riempire. Questa funzione ritorna un valore pari al numero dei frame effettivamente letti, se meno di period è una short read, se vale "-EPIPE" vuol dire overrun (il programma non legge i dati abbastanza velocemente e si passa alla riacquisizione mediante *continue*). Terminati questi check ed eventualmente visualizzare gli errori, si entra nel for per spostare i frame nel buffer atomico condiviso con gli altri thread aggiornando la variabile posRaw comunicando a che punto passa l'acquisizione del frame e il timing del thread [17]. Si ricorda inoltre che i e *posRaw* sono variabili atomiche, questo è fondamentale ai fini della sincronizzazione tra i thread, i rappresenta quale PERIOD è stato appena acquisito (da 0 a 3) mentre *posRaw* rappresenta la posizione nel BUFFER (quindi a quale frame passa, da 0 a 1023).

7.4 processorThread

```
1  void processorThread(){
2      usleep(120000);
3      Frame proc[PERIOD];
4      State X;
5      State Y;
6      while (true) {
7
8          for( h ; h < BUFFER_SIZE/PERIOD; h.store(h=h+1)){
9              while(i.load() == h){std::this_thread::yield();}
10             for ( int k = 0; k < PERIOD; k++) {
11                 proc[k] = bufferRaw[h*PERIOD+k].load();
12             }
13
14             ///////////////    ALGORITMO DSP    ///////////////
15
16             while(coeff_upt_flag.load()){std::this_thread::yield();}
17
18             for( int k = 0; k < PERIOD; k++) {
19             X.L = static_cast<double>(proc[k].L);
20             X.R = static_cast<double>(proc[k].R);
21
22             Y.L = coeff_sx.b0 * X.L + Y1.L;
23             Y1.L = coeff_sx.b1 * X.L - coeff_sx.a1 * Y.L + Y2.L;
24             Y2.L = coeff_sx.b2 * X.L - coeff_sx.a2 * Y.L;
25
26             if(Y.L * gain_sx > 32767.0 || Y.L * gain_sx < -32768.0){
27                     std::cout << "Clipping detected on left channel"
28                     std::ofstream gpio24("/sys/class/gpio/gpio536/value");
29                     gpio24 << 1;
30             }
31             proc[k].L = (std::clamp(Y.L * gain_sx, -32768.0, 32767.0));
32
```

```
33              Y.R = coeff_dx.b0 * X.R + Y1.R;
34              Y1.R = coeff_dx.b1 * X.R - coeff_dx.a1 * Y.R + Y2.R;
35              Y2.R = coeff_dx.b2 * X.R - coeff_dx.a2 * Y.R;
36
37              if(Y.R * gain_dx > 32767.0 || Y.R * gain_dx < -32768.0){
38                      std::cout << "Clipping detected on right channel"
39                      std::ofstream gpio24("/sys/class/gpio/gpio536/value");
40                      gpio24 << 1;
41              }
42              proc[k].R = (std::clamp(Y.R * gain_dx, -32768.0, 32767.0));
43              }
44
45              ///////////////   FINE ALGORITMO DSP    ///////////////
46
47              for( int k = 0; k < PERIOD; k++) {
48              bufferProc[h*PERIOD+k].store(proc[k]);
49              posProc.store(h*PERIOD+k);
50              }
51          }
52          h.store(0);
53          std::ofstream gpio24("/sys/class/gpio/gpio536/value");
54          gpio24 << 0;
55      }
56  }
```

La funzione processorThread associata al thread t2 (vedi sezione 5.3) svolge la fase di processamento della catena audio. Si chiama il ritardo di 120 ms (per eseguire la funzione nel modo corretto avendo già il buffer pieno dalla prima iterazione) e si inizializza la variabile di tipo frame per contenere i valori locali, in questo caso il buffer e gli stati del filtro; inizia così il loop infinito che eseguirà il thread. In primo luogo però bisogna prima sincronizzare il sistema per fare in modo che il dialogo tra i buffer sia il più efficiente possibile senza aspettare troppo tempo ma nemmeno rileggendo vecchi valori. Questo viene attuato con una strutta a blocchi tipica dell'attuale implementazione, ovvero riprendendo prima il buffer nelle sue sezioni PERIOD con indice h facendo scorrere al loro interno i frame con indice i. Così il while aspetta che gli indici siano uguali (questo vuol dire che il

blocco che si vorrebbe processare è appena stato acquisito correttamente), nel mentre la funzione *yield* rilascia il carico computazionale, che nella precedente implementazione era occupato dal polling, ad un altro thread, finché la condizione del while diventa falsa.

Si entra poi nel cuore del programma, la sezione dell'algoritmo che implementa il filtro, con architettura Transposed direct form II. Per prima cosa in caso ci sia un aggiornamento in corso dei coefficienti il codice si ferma nel while per evitare distorsioni, quando il flag viene rilasciato si può cominciare l'algoritmo. Si lavorano i frame a blocchi di PERIOD, in modo tale da avere sempre qualcosa da mandare in uscita e al contempo non sovraccaricare il processore. Vengono importati i campioni negli stati locali double, per fare in modo da avere una miglior risoluzione nei calcoli per poi fare un casting della variabile in unsigned 16-bit, si lavora prima con il canale sinistro, implementando le operazioni descritte dall'architettura, per poi controllare che la variabile non saturi a seguito di un gain troppo elevato; in questo caso si assegna il massimo valore del range (positivo o negativo) e si accende il LED designato a questo scopo, stessa cosa per il canale destro. Al termine dell'algoritmo viene caricato il secondo buffer per permettere poi all'altro thread la riproduzione in output, infine si spegne il clipping-LED e si azzera l'indice del PERIOD.

7.4.1 Il sincronismo

La sincronizzazione è stata probabilmente il problema più importante riscontrato durante questo studio, non tanto per errori di compilazione o difficoltà logica di implementazione, quanto per un disturbo sonoro che faticava ad andare via, guardando il codice sembrava riportare tutto alla perfezione. Dopo svariati tentativi e ottimizzazioni, con la dovuta raffinatezza ricercata in funzionalità ignote prima dello studio, si è trovato il problema nel sincronismo tra i thread e il problema è scomparso riuscendo a non avere più glitch tra un PERIOD e l'altro, questo rumore infatti si manifestava quando veniva settata sul filtro low-pass una f_c molto bassa. Facendo momentaneamente un passo indietro si ricorda che un tono puro a 20 Hz ha un periodo che dura 50 ms, un tono a 200 Hz 5 ms, questo vuol dire che con le attuali impostazioni del buffer l'ultimo tono il cui periodo entra completamente in un buffer è 200 Hz. La conclusione è che probabilmente era presente una correlazione con il sincronismo tra i vari blocchi, quando la frequenza riprodotta doveva coprire più PERIOD si presentava il glitch che si generava tra un passaggio e l'altro per busy-wait (polling mal gestito), distorcendo l'intero segnale con un'oscillazione su tutto lo

spettro.

7.5 playbackThread

```
1  void playbackThread(snd_pcm_t* playback) {
2      usleep(140000);
3      Frame output[PERIOD];
4      while (true) {
5
6          for( int j = 0; j < BUFFER_SIZE/PERIOD; j++){
7              while(h.load() == j){std::this_thread::yield();}
8
9              for( int k = 0; k < PERIOD; k++) {
10                 output[k] = bufferProc[j*PERIOD+k].load();
11             }
12             int frames = snd_pcm_writei(playback, output, PERIOD);
13             if(frames ==-EPIPE) {
14                 std::cout << "Error: buffer underrun\n";
15                 snd_pcm_prepare(playback);
16             }
17         }
18     }
19 }
```

La funzione playbackThread associata al thread t3 (vedi sezione 5.3) svolge la fase di riproduzione della catena audio. Una volta chiamato il ritardo di 140ms per aspettare il `processorThread` si inizializzano le variabili frame per contenere i samples locali. Adottando lo stesso metodo di sincronizzazione sopra descritto faccio un passaggio dal buffer di processamento a quello locale così da poterlo mandare in uscita con la funzione ALSA `snd_pcm_writei()`. Il valore restituito indica il numero di frame che sono stati effettivamente scritti e mandati alla scheda audio, in caso si trovi il buffer vuoto, parzialmente o totalmente, (underrun) si deve rieffettuare la lettura dal `bufferProc` così da ripetere l'output con il PERIOD successivo di campioni corretto.

7.6 Main

```
int main(){

    // INPUT STEREO
    snd_pcm_t* capture = openPCM("plughw:1,0", SND_PCM_STREAM_CAPTURE, CH_IN);

    // OUTPUT STEREO
    snd_pcm_t* playback = openPCM("plughw:1,0", SND_PCM_STREAM_PLAYBACK, CH_OUT);

    std::thread t1(captureThread, capture);
    std::thread t2(processorThread);
    std::thread t3(playbackThread, playback);

    init_i2c();

    system("echo 535 > /sys/class/gpio/export");
    system("echo 536 > /sys/class/gpio/export");
    system("echo 526 > /sys/class/gpio/export");
    system("echo 527 > /sys/class/gpio/export");
    system("echo 530 > /sys/class/gpio/export");
    system("echo 529 > /sys/class/gpio/export");
    system("echo 539 > /sys/class/gpio/export");
    system("echo 534 > /sys/class/gpio/export");

    system("echo in > /sys/class/gpio/gpio535/direction");
    system("echo out > /sys/class/gpio/gpio536/direction");
    system("echo in > /sys/class/gpio/gpio526/direction");
    system("echo in > /sys/class/gpio/gpio527/direction");
    system("echo in > /sys/class/gpio/gpio530/direction");
    system("echo in > /sys/class/gpio/gpio529/direction");
    system("echo in > /sys/class/gpio/gpio539/direction");
    system("echo in > /sys/class/gpio/gpio534/direction");

```

```
33      std::cout << "RUNNING\n";
34  while(true){
35      adc0 = read_ADC(fd, 0xC3);
36      usleep(10000);
37      adc0 = read_ADC(fd, 0xC3);
38      usleep(10000);
39      adc1 = read_ADC(fd, 0xD3);
40      usleep(10000);
41      adc1 = read_ADC(fd, 0xD3);
42      usleep(10000);
43      adc2 = read_ADC(fd, 0xE3);
44      usleep(10000);
45      adc2 = read_ADC(fd, 0xE3);
46      usleep(10000);
47      adc3 = read_ADC(fd, 0xF3);
48      usleep(10000);
49      adc3 = read_ADC(fd, 0xF3);
50      usleep(10000);
51      fc_sx = mapADCtoFC(adc0);
52      fc_dx = mapADCtoFC(adc2);
53      gain_sx = mapADCtoGain(adc1);
54      gain_dx = mapADCtoGain(adc3);
55
56      std::ifstream gpio526("/sys/class/gpio/gpio526/value");
57      gpio526 >> GPIO14;
58      std::ifstream gpio527("/sys/class/gpio/gpio527/value");
59      gpio527 >> GPIO15;
60      std::ifstream gpio530("/sys/class/gpio/gpio530/value");
61      gpio530 >> GPIO18;
62      std::ifstream gpio529("/sys/class/gpio/gpio529/value");
63      gpio529 >> GPIO17;
64      std::ifstream gpio539("/sys/class/gpio/gpio539/value");
65      gpio539 >> GPIO27;
66      std::ifstream gpio534("/sys/class/gpio/gpio534/value");
67      gpio534 >> GPIO22;
```

```
68
69      if(GPIO15 == 1) type_f    = 'H';
70      else if(GPIO18 == 1) type_f = 'B';
71      else type_f = 'L';
72
73      if (GPIO17 == 1) q = 0.1;
74      else if (GPIO22 == 1) q = 0.99;
75      else q = 0.7071067811865476;
76
77     std::cout << fc_sx << " " << fc_dx << std::endl;
78
79      coeff_upt_flag.store(true);
80      coeff_sx = updateFilter(fc_sx, type_f, q);
81      coeff_dx = updateFilter(fc_dx, type_f, q);
82      coeff_upt_flag.store(false);
83
84      }
85      return 0;
86  }
```

Per prima cosa nel main vengono creati gli handler, ovvero puntatori che contengono le informazioni relative alla comunicazione audio con la scheda Behringer, che saranno utilizzate dalla libreria ALSA per l'interfacciamento e la comunicazione. `plughw:1,0` significa letteralmente "plugin hardware" e descrive il numero della porta che la Raspberry associa alla scheda audio. In seguito vengono associate le funzioni relative al DSP al proprio thread, viene inizializzata la comunicazione I2C (che verrà discussa nella sezione 7.7.3) e la generazione delle cartelle con conseguente direzione delle porte GPIO.

Terminata questa prima fase di setup del sistema viene mandato in output il messaggio "RUNNING" per comunicare all'utente che non si sono verificati errori e che le funzioni del DSP sono in esecuzione. Comincia la parte iterativa del gestore, la lettura degli ingressi analogici, relativa alle frequenze di taglio e ai guadagni dei canali, la lettura viene svolta richiamando la funzione `read_ADC()` (sezione 7.7.4) con duplice richiesta. L'ADS1115 [14] è composto come precedentemente spiegato da 4 ingressi analogici indipendenti, per la selezione dell'ingresso desiderato viene fornito dal chip un multiplexer che, a seconda di

come lo si programma, seleziona uno dei 4 ADC. Quando questo valore viene modificato la conversione immediatamente successiva può non rappresentare il giusto ingresso, quindi è necessario consentire al convertitore di eseguire una nuova conversione con il nuovo canale per garantire la corretta riuscita a discapito di un ritardo trascurabile.

Segue la mappatura dei valori appena acquisiti, in modalità differente a seconda dell'informazione che porta l'i-esimo buffer; per quanto riguarda `adc0` e `adc2` viene richiamata la funzione `mapADCtoFC()`, mentre per `adc1` e `adc3` la funzione `mapADCtoGain()`, descritte nella sezione 7.7.5.

Per quanto riguarda i segnali digitali, acquisiti tramite la funzione `std::ifstream`, si prende dalla relativa cartella il valore ricavato dalla lettura, per poi inserirlo nella variabile globale specifica. In questa sezione sono presenti solo i GPIO in ingresso, dato che l'unico in output è quello del LED che segnala il clipping, che però è gestito direttamente dal `processorThread`.

Una volta a disposizione di tutti i parametri si passa all'elaborazione di questi; nelle ultime righe del main si sceglie a seconda di quale switch è stato premuto la forma del filtro e il valore di q. Essendo iterato continuamente nel main, questo processo garantisce un'ottima risposta senza una latenza percepibile tra l'attuazione del trimmer e il riscontro di variazione sonora.

Viene anche eseguito come debug un output sul terminale in cui si plottano i valori delle frequenze di taglio attuali, usato anche ad esempio per verificare la corretta parametrizzazione dei risultati finali sui grafici del capitolo 8.

Ultima funzione implementata è l'aggiornamento dei coefficienti sui due canali indipendenti, con un controllo del flag `coeff_upt_flag` in modo che, come visto nella sezione 7.4, non si possa eseguire il filtraggio durante l'aggiornamento dei coefficienti.

7.7 Funzioni richiamate nel main

7.7.1 Configurazione scheda audio

```
snd_pcm_t* openPCM(const char* device, snd_pcm_stream_t stream, int channels) {
    snd_pcm_t* handle;
    snd_pcm_hw_params_t* params;

    int err = snd_pcm_open(&handle, device, stream, 0);

    snd_pcm_hw_params_alloca(&params);
    snd_pcm_hw_params_any(handle, params);

    snd_pcm_hw_params_set_access(handle, params, SND_PCM_ACCESS_RW_INTERLEAVED);
    snd_pcm_hw_params_set_format(handle, params, SND_PCM_FORMAT_S16_LE);
    snd_pcm_hw_params_set_channels(handle, params, channels);

    unsigned int rate = SAMPLE_RATE;
    snd_pcm_hw_params_set_rate_near(handle, params, &rate, 0);

    snd_pcm_uframes_t period = PERIOD;
    snd_pcm_hw_params_set_period_size_near(handle, params, &period, 0);

    snd_pcm_uframes_t buffer = BUFFER_SIZE;
    snd_pcm_hw_params_set_buffer_size_near(handle, params, &buffer);

    snd_pcm_hw_params(handle, params);
    snd_pcm_prepare(handle);

    return handle;
}
```

Questa sezione di codice stabilisce quali sono i parametri della scheda audio che verranno utilizzati da ALSA per la comunicazione. Si può notare un ulteriore controllo sulla comunicazione con la scheda audio alla riga 5, viene infatti riconosciuto il mancato collegamento e la conseguente segnalazione all'utente che non è stato possibile comunicare con la Behringer (non verrà plottata la scritta "RUNNING").

I parametri chiave sono:

- Frequenza di campionamento a 48 kHz (la massima consentita dalla Behringer)
- Configurazione stereofonica, quindi doppio canale indipendente sia in ingresso che in uscita
- Risoluzione della quantizzazione, quindi il formato che ne consegue a 16-bit è l'intero S16_LE ovvero una variabile signed (da -32768 a 32767) e little-endian (ovvero il byte meno significativo viene memorizzato prima)
- I campioni organizzati sotto forma interleaved ovvero in maniera alternata tra canale destro e sinistro
- Grandezza dei buffer utilizzati per la divisione a blocchi quali `bufferRaw[]` e `bufferProc[]` divisi a loro volta in PERIOD

7.7.2 Calcolo dei coefficienti

```
1  biquad updateFilter(int new_fc, char a, double q) {
2
3          biquad f;
4
5          if (a == 'L') {
6          // Calcolo dei coefficienti del filtro passa basso
7
8          double w0 = 2*PI*new_fc/SAMPLE_RATE;
9          double cos_w0 = cos(w0);
10         double alpha = sin(w0)/(2*q);
```

```
11          double b0 = (1-cos_w0)/2;
12          double b1 = 1-cos_w0;
13          double b2 = (1-cos_w0)/2;
14          double a0 = 1+alpha;
15          double a1 = -2*cos_w0;
16          double a2 = 1-alpha;
17          f.b0 = b0/a0;
18          f.b1 = b1/a0;
19          f.b2 = b2/a0;
20          f.a1 = a1/a0;
21          f.a2 = a2/a0;
22          }
23
24          else if (a == 'H') {
25          // Calcolo dei coefficienti del filtro passa alto
26
27          double w0 = 2 * PI * new_fc / SAMPLE_RATE;
28          double cos_w0 = cos(w0);
29          double alpha = sin(w0) / (2 * q);
30          double b0 = (1 + cos_w0) / 2;
31          double b1 = -(1 + cos_w0);
32          double b2 = (1 + cos_w0) / 2;
33          double a0 = 1 + alpha;
34          double a1 = -2 * cos_w0;
35          double a2 = 1 - alpha;
36          f.b0 = b0 / a0;
37          f.b1 = b1 / a0;
38          f.b2 = b2 / a0;
39          f.a1 = a1 / a0;
40          f.a2 = a2 / a0;
41          }
42
43          else if (a == 'B') {
44          // Calcolo dei coefficienti del filtro passa banda
45
```

```
46          double w0 = 2 * PI * new_fc / SAMPLE_RATE;
47          double cos_w0 = cos(w0);
48          double alpha = sin(w0) / (2 * q);
49          double b0 = alpha;
50          double b1 = 0;
51          double b2 = -alpha;
52          double a0 = 1 + alpha;
53          double a1 = -2 * cos_w0;
54          double a2 = 1 - alpha;
55          f.b0 = b0 / a0;
56          f.b1 = b1 / a0;
57          f.b2 = b2 / a0;
58          f.a1 = a1 / a0;
59          f.a2 = a2 / a0;
60          }
61
62          return f;
63  }
```

Nello specifico si susseguono 3 if/else che controllano il parametro "`a`" che può assumere lo stato "L" di Low-pass, "B" di Band-pass e "H" di High-pass, così da poter chiamare la stessa funzione mentre internamente si eseguono equazioni diverse ottimizzando il codice.

Il tipo della funzione è `biquad` ovvero la struttura contenente i coefficienti dichiarata nella sezione 7.2.

La funzione è chiamata con il passaggio dei parametri relativi alla frequenza di taglio, al tipo di filtro, e al parametro q, che verranno utilizzati per variare i coefficienti seguendo le formule del metodo RBJ [10] precedentemente viste nel capitolo 3.

7.7.3 Inizializzazione I2C

```
1
2  void init_i2c() {
3      fd = open("/dev/i2c-1", O_RDWR);
4      if (fd < 0) perror("i2c");
5      ioctl(fd, I2C_SLAVE, ADS1115_ADDR);
6
7      uint8_t buf[3];
8      // Hi_thresh = 0x8000 (abilita ALERT/RDY come conversion-ready)
9      buf[0] = 0x03;
10     buf[1] = 0x80;
11     buf[2] = 0x00;
12     write(fd, buf, 3);
13     //low
14     buf[0] = 0x02;
15     buf[1] = 0x00;
16     buf[2] = 0x00;
17     write(fd, buf, 3);
18 }
```

La funzione *open*() apre il dispositivo `i2c-1` in modalità "O_RDWR" ovvero sia in lettura sia in scrittura mettendo il risultato dentro il file-descriptor `fd` (si controlla anche la corretta apertura con l'if, e in caso si manda in output l'errore).

Con *ioctl*() (Input-Output control) si assegna l'indirizzo dello slave I2C per la comunicazione sul bus, così che da questo momento in poi ogni comunicazione attraverso questo protocollo dialoghi unicamente con il chip ADS1115 [14].

Quando si scrive su questo chip è necessario inviare in primo luogo l'indirizzo del registro, poi il byte più significativo e per ultimo il byte meno significativo, quindi si inizializza un buffer contenente 3 byte e si scrivono i numeri che consentono di impostare il pin ALRT/READY come flag di fine conversione.

7.7.4 Lettura dall'ADC

```
1  // Lettura adc
2  int read_ADC(int fd, uint8_t channel){
3
4      uint8_t config[3] = {REG_CONFIG, channel, 0x86};
5      write(fd, config, 3);
6
7      while (GPIO23 != 0) {
8          std::ifstream gpio23("/sys/class/gpio/gpio529/value");
9          gpio23 >> GPIO23;
10         usleep(10);
11     }
12
13     uint8_t buf[2], reg = REG_CONVERT;
14     if (write(fd, &reg, 1) != 1 || read(fd, buf, 2) != 2) return -1;
15
16     return static_cast<int16_t>((buf[0] << 8) | buf[1]);
17 }
```

A questa funzione viene passato il canale da leggere con formato specificato nel datasheet dell'ADS1115 [14], insieme al file descriptor per gestire la comunicazione tra Raspberry e ADC.

Si scrive un buffer con l'indirizzo del registro e del buffer dati, si aspetta che il pin ALRT/READY ritorni basso (il fronte di discesa è sinonimo di fine conversione), si scrive al modulo quale memoria si vuole leggere e si inserisce il valore convertito dal registro `REG_CONVERT` alla variabile locale `reg`.

7.7.5 Mappature dei segnali

```
// Mappatura valore letto dall'ADC per la frequenza di taglio
int mapADCtoFC(int adc){
    if (adc < 20) adc = 20;
    if (adc > 24000) adc = 24000;

    float norm = (float)adc / 24000.0f;

    // mapping esponenziale da 20 Hz a 20 kHz
    float fc = 20.0f * powf(1000.0f, norm);

    return (int)fc;
}

// Mappatura valore letto dall'ADC per il gain
double mapADCtoGain(int adc) {
    if (adc < 20) adc = 20;
    if (adc > 24000) adc = 24000;

    float norm = (float)(adc - 20) / (24000.0f - 20.0f);

    // mapping esponenziale da 0.1x a 3x
    float gain = 0.1f * powf(30.0f, norm);

    return gain;
}
```

In questa sezione si analizzerà il metodo scelto per mappare il valore contenuto nel buffer dell'ADC nel numero che identificherà la frequenza di taglio o il gain del relativo canale.

Appena chiamata la funzione viene fatto subito un casting della variabile, questo sia per evitare comportamenti irrazionali sia per definire un range di acquisizione fisso e stabile su cui normalizzare le operazioni (oltre che rimuovere il rumore agli estremi del range di misura).

Si noti che l'ADC teoricamente può arrivare fino a 32768 ma avendo scelto il PGA a 4.096 V (si rimanda alla sezione 4.4), non tutto il range di misura può essere ricoperto, data l'alimentazione dei trimmer a 3.3 V (si rimanda alla sezione 4.5.2).

Così, empiricamente, si è visto il numero associato al fondo scala del trimmer e approssimando per difetto si è scelto il numero 24000.

La Formula (7.1) esprime la normalizzazione del valore rispetto a 24000

$$\text{norm}_{f_c} = \frac{ADC}{24000} \tag{7.1}$$

La motivazione principale per cui si è deciso di adottare la scala logaritmica è per avere una risposta alle variazioni più simile possibile a quella dell'orecchio umano, che per l'appunto segue questo andamento, notando di più piccole variazioni in bassa frequenza rispetto che in alta frequenza, stessa cosa per il gain.

$$f_c = 20 \cdot 1000^{\text{norm}_{f_c}} \tag{7.2}$$

In uguale misura normalizzando sul range del gain che ha un offset anche sul minimo per evitare il rumore intorno lo zero

$$\text{norm}_G = \frac{ADC - 20}{24000 - 20} \tag{7.3}$$

Questa mappatura consente di avere valori in scala logaritmica da 0.1 a 3

$$G = 0.1 \cdot 30^{\text{norm}_G} \tag{7.4}$$

Si noti che il parametro della frequenza di taglio è un intero mentre il gain è un double, per migliorare la risoluzione nei calcoli intermedi del filtro.

Capitolo 8

Risultati sperimentali

L'ultima fase del progetto è stata la verifica del lavoro svolto, in maniera sperimentale, attraverso misurazioni metodiche, stabili e riproducibili. L'obiettivo è verificare che il sistema sia capace di acquisire, elaborare e riprodurre audio, con la modifica dei parametri impostati dall'utente che deve essere coerente con risposta in frequenza; per fare ciò è stata utilizzata un'analisi tramite FFT [7] [18] con il software NU-Tech, in modo da avere un riscontro oggettivo oltre alla semplice valutazione dell'ascolto. È stato quindi creato un sistema a ciclo chiuso che comprende l'output del rumore bianco da un computer, l'ingresso nel sistema DSP e l'uscita letta in input dallo stesso computer per analizzarne lo spettro.

8.1 Low-pass e High-pass

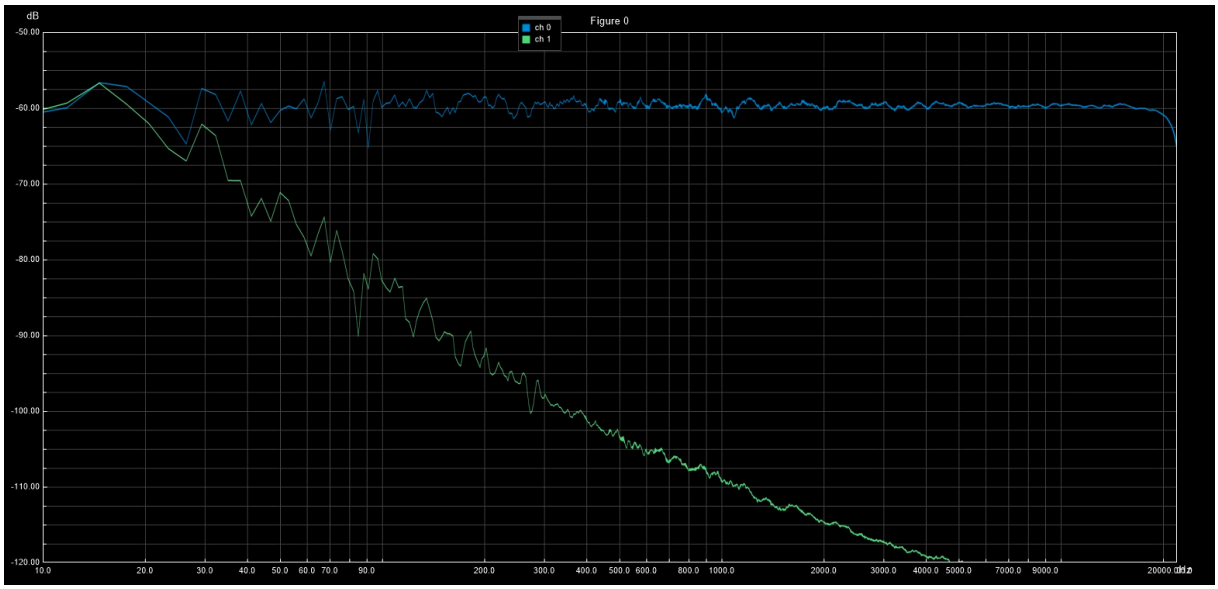


Figura 8.1: FFT Low-pass; fc_sx = 20 kHz; fc_dx = 20 Hz

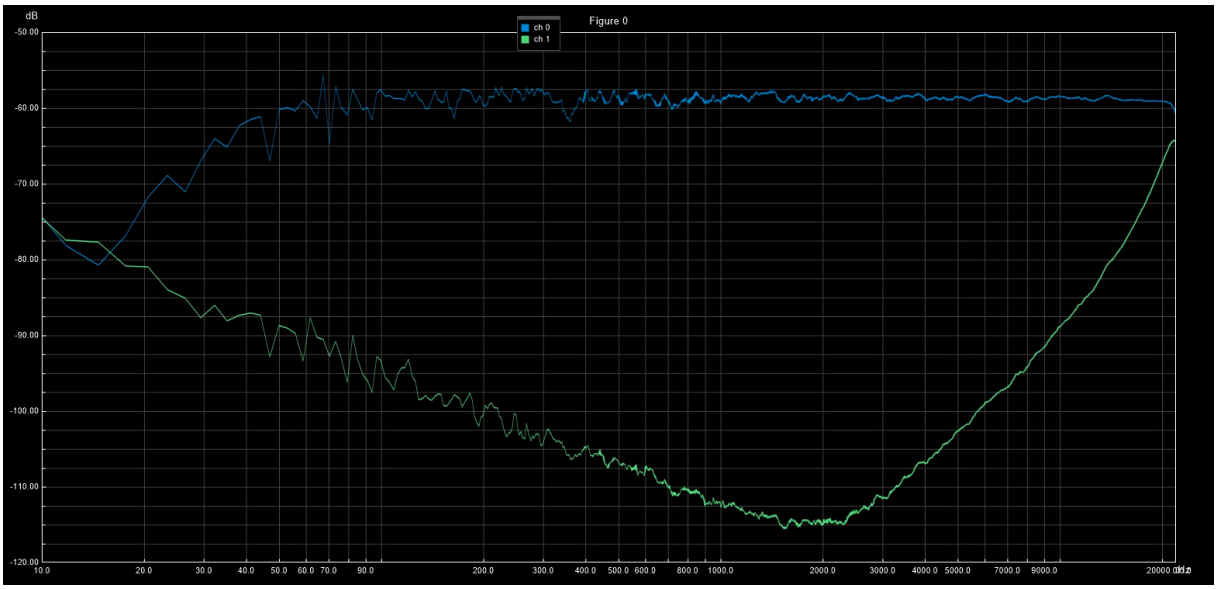


Figura 8.2: FFT High-pass; fc_sx = 20 Hz; fc_dx = 20 kHz

Le prime prove riportate nelle figure 8.1 e 8.2 riportano la risposta in frequenza dei filtri in configurazione passa basso e passa alto, si è deciso per migliorare la visualizzazione e mostrare al contempo l'indipendenza dei canali di assegnare a ogni canale la frequenza di taglio estrema e opposta, che sono proprio i casi dove i filtri digitali fanno più fatica e si comportano in maniera non ideale.

A livello applicativo si può dedurre dal grafico che implementando questo filtro HP per alimentare un tweeter si dovrà comunque adottare un condensatore di filtraggio analogico, dato che in estrema bassa frequenza, seppur le frequenze vengano attenuate di oltre 20 dB un componente di bassa potenza dal costo irrisorio può solo che migliorare la durabilità del driver audio.

È importante far notare che le caratteristiche auspicabili del filtro, che sono state raggiunte pienamente, sono:

- La frequenza di taglio deve trovarsi nella posizione prevista e avere un'attenuazione di 3 dB rispetto al modulo in banda passante.

- In banda passante il modulo deve rimanere costante quindi senza attenuazioni indesiderate (nei limiti del rumore)

- Il roll-off, quindi la pendenza di attenuazione, tipica della tipologia di filtro, in questo caso rispetta i $-40\,\mathrm{dB/decade}$ previsti dal biquad che è di secondo ordine.

- Simmetria e indipendenza dei due canali, quindi non devono influenzarsi a vicenda e invertendo le frequenze di taglio i relativi roll-off devono rimanere inalterati

8.2 Band-pass

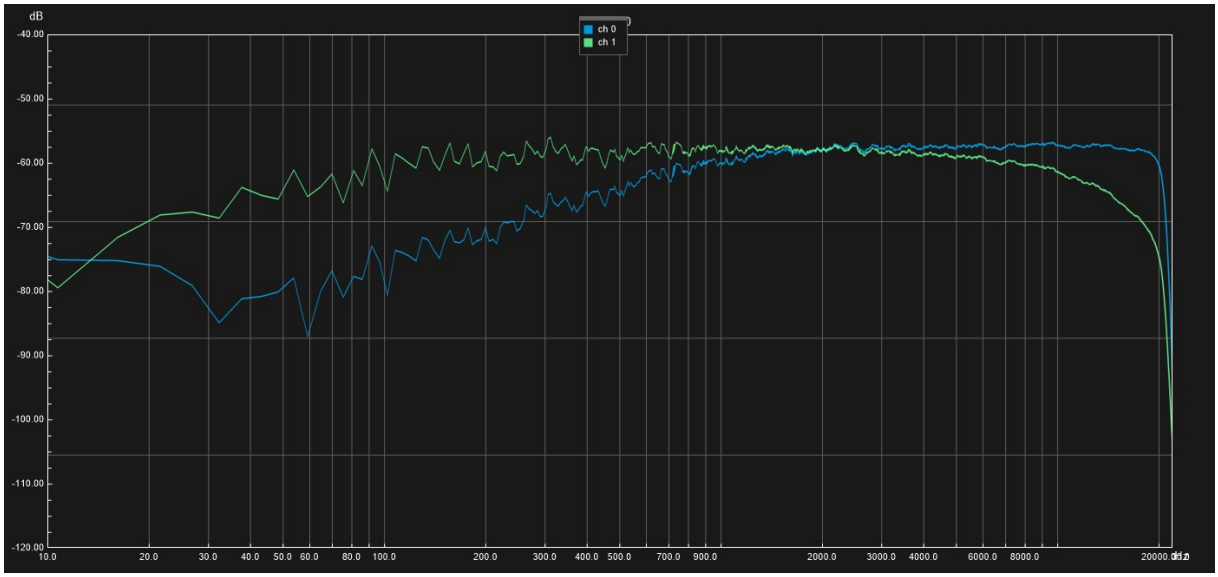


Figura 8.3: FFT Band-pass; fc_sx = 10 kHz; fc_dx = 1 kHz; q = 0.1

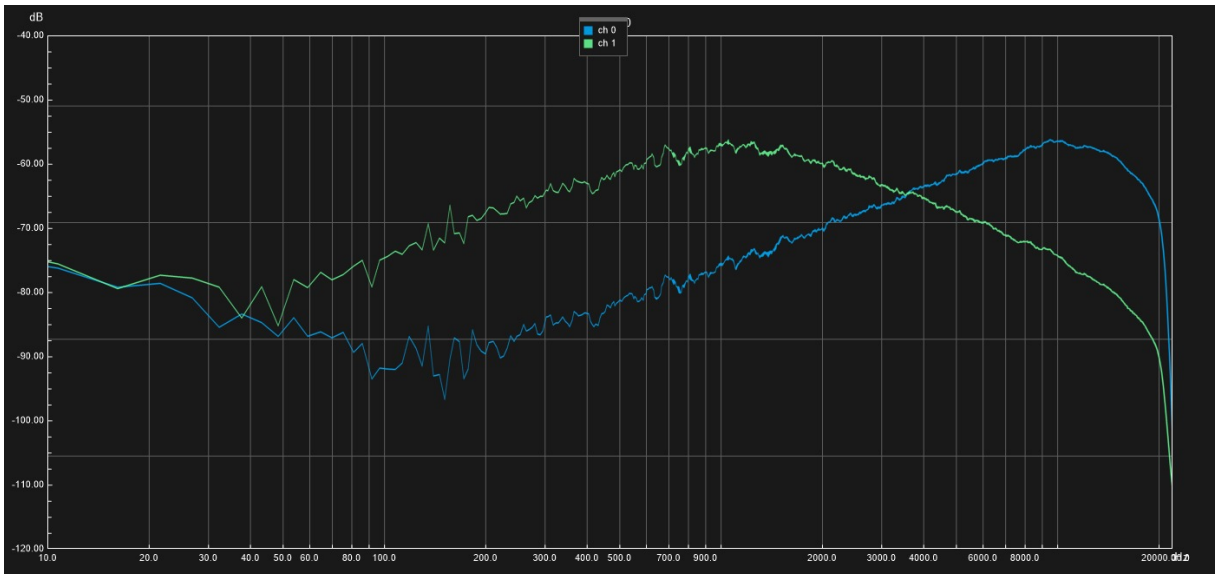


Figura 8.4: FFT Band-pass; fc_sx = 10 kHz; fc_dx = 1 kHz; q ≈ 0.707

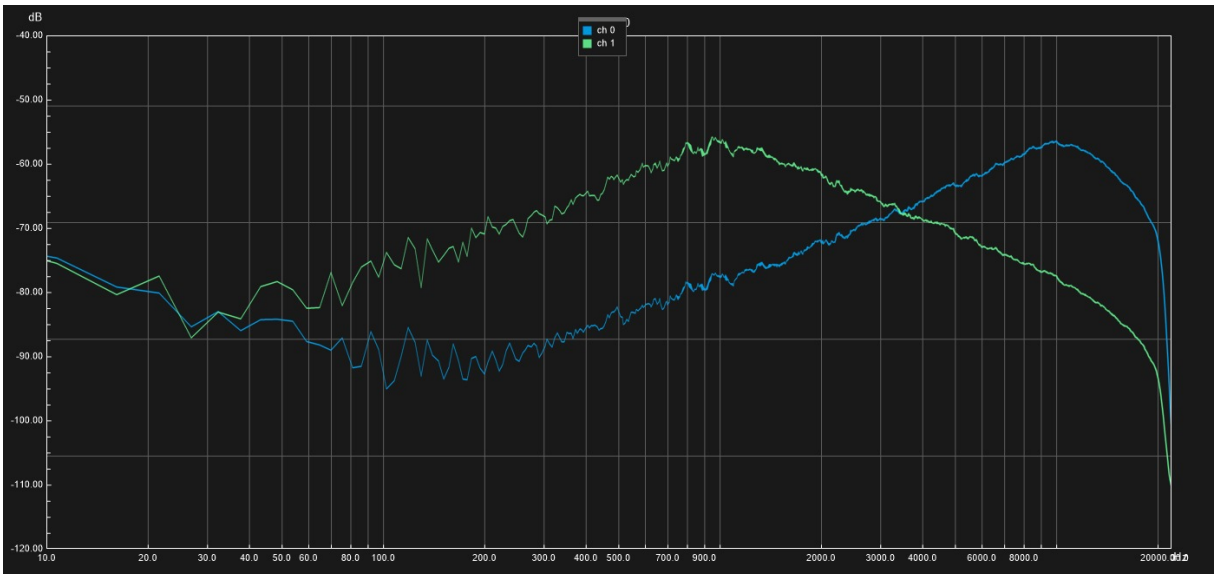


Figura 8.5: FFT Band-pass; fc_sx = 10 kHz; fc_dx = 1 kHz; q = 0.99

Per quanto riguarda i filtri passa banda le prove sperimentali sono state eseguite mantenendo le frequenze di taglio fisse a 1 kHz e 10 kHz (comunque diversificando tra i due canali) e variando il parametro q rispettivamente nei valori 0.1, 0.707 e 0.99, che nel caso dei passa-banda stabilisce la selettività del filtro.

Il comportamento complessivo della FFT evidenzia la formazione della banda passante intorno alla frequenza di taglio determinata; come si può notare dalla figura 8.3 l'andamento in frequenza è molto arrotondato e poco filtrante, utile magari per equalizzazioni di precisione che richiedono piccole variazioni in modulo, si passa poi alla figura 8.4 che è una via di mezzo (il valore di q è proprio quello perché risale al filtro di Butterworth ovvero $\frac{1}{\sqrt{2}}$ che produce una risposta in banda passante massimamente piatta). Infine si arriva alla figura 8.5, utile in casi di filtraggio di protezione (come potrebbe accadere su dei tweeter per le basse frequenze) o per evitare una distorsione molto forte (come avverrebbe per le alte frequenze su un sub-woofer, poiché avendo una sospensione molto morbida sarebbe poco preciso per riprodurre quel range di frequenze e creerebbe distorsione).

Capitolo 9

Conclusioni

Il lavoro svolto ha portato alla progettazione e all'implementazione di un DSP audio real-time sulla piattaforma offerta dalla Raspberry Pi 4 model B, mediante l'ausilio della scheda audio Behringer UCA-222 e del modulo ADC ADS1115, sviluppato interamente in C++ con l'integrazione dei driver ALSA, per dialogare nel modo più efficiente possibile con la scheda audio tramite USB.

L'obiettivo principale era verificare la fattibilità di questo approccio, usando una scheda a basso costo per un'elaborazione non banale a un livello molto basso di controllo dei campioni via software, mantenendo allo stesso modo la possibilità di configurare dinamicamente i parametri del sistema in tempo reale. L'algoritmo permette attualmente di processare segnali 16-bit a 48 kHz, mostrando un carico davvero contenuto sulle risorse computazionali grazie ai tanti accorgimenti adottati per l'ottimizzazione di sistema.

La struttura implementata costituisce inoltre una base per lo sviluppo futuro verso una soluzione maggiormente integrata, progettando un ADC/DAC embedded per raggiungere i 192 kHz di frequenza di campionamento, con l'obiettivo di raggiungere una risoluzione effettiva dell'ordine dei ≈ 24 bit effettivi (ENOB), comunicando con un'interfaccia CODEC che utilizza la tecnica TDM per la gestione di un numero maggiore di canali, in particolare fino a 8 uscite digitali.

È stata verificata la robustezza e la stabilità nel tempo della struttura biquad Transposed Direct Form II che consente di eseguire il filtraggio mantenendo un numero ridotto di variabili di stato; anche il metodo RBJ ha avuto un riscontro positivo garantendo un flusso costante e senza interruzioni dei nuovi campioni in maniera rapida e senza distorsione (grazie anche all'avanzato sincronismo adottato).

Dal punto di vista dell'implementazione real-time un ruolo fondamentale è stato svolto sicuramente dallo scheduling dei thread, suddividendo i processi in acquisizione, processamento, riproduzione e gestione (main), permettendo una distribuzione ottimale del carico di lavoro mantenendo anch'esso costante il flusso dei campioni lungo tutta la catena; inoltre la gestione a blocchi con un PERIOD lungo 256 frame e un Buffer contenente 4 PERIOD, garantisce il giusto compromesso tra latenza e carico computazionale.

Le prove sperimentali effettuate mediante FFT hanno infine confermato il comportamento previsto dai filtri mostrando variazioni della risposta in frequenza coerenti con i parametri assegnati dall'utente confermando l'indipendenza e la selettività dei canali.

In conclusione il progetto ha raggiunto gli obiettivi inizialmente stabiliti, dimostrando la fattibilità dell'utilizzo di una Raspberry per gestire un flusso audio configurabile e real-time. Il sistema sviluppato fornisce una base solida per sviluppi futuri, in ambito hardware attraverso la progettazione dell'interfaccia audio embedded contenente anche la gestione dei parametri e in ambito software un'ulteriore ottimizzazione con la gestione di più canali per allargare le possibilità implementative del sistema.

Bibliografia

[1] Gianni Pasolini, Alessandro Bazzi, and Flavio Zabini. A raspberry pi-based platform for signal processing education. *IEEE Signal Processing Magazine*, 34(4):151–158, 2017.

[2] Ngoc Thy Pham. Real time audio processing on a raspberry pi using partitioned convolution. Bachelor's thesis, HAW Hamburg, 2017.

[3] Cirrus Logic. *Cirrus Logic Audio Card*. Cirrus Logic, 202X. Datasheet / User Guide.

[4] Juan Emilio Noriega-Linares and Juan Miguel Navarro Ruiz. On the application of the raspberry pi as an advanced acoustic sensor network for noise monitoring. *Electronics*, 5(4):74, 2016.

[5] Manuel Caldas-Morgan, Alexander Alvarez-Rosario, and Linilson Rodrigues Padovese. An autonomous underwater recorder based on a single board computer. *PLoS ONE*, 10(6):e0130297, 2015.

[6] Texas Instruments. *Stereo Audio CODEC w/USB Interface, Single-Ended Analog Input/Output & S/PDIF*. Texas Instruments, 2008. Rev. E.

[7] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Pearson, 3rd edition, 2009.

[8] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[9] Sophocles J. Orfanidis. *Introduction to Signal Processing*. Pearson Education, 2010.

[10] Robert Bristow-Johnson. The equivalence of various methods of computing biquad coefficients for audio parametric equalizers. In *97th Convention of the Audio Engineering Society*, San Francisco, CA, USA, November 1994. AES Preprint 3906.

[11] Christopher Hallinan. *Embedded Linux Primer*. Pearson, 2nd edition, 2011.

[12] Raspberry Pi Ltd. *Raspberry Pi 4 Model B Product Brief*. Raspberry Pi Ltd, 2026.

[13] Behringer. *U-CONTROL UCA222 User Manual*. Behringer, 2010. Ultra-Low Latency 2 In/2 Out USB Audio Interface with Digital Output.

[14] Texas Instruments. *ADS111x Ultra-Small, Low-Power, 16-Bit ADCs with Internal Reference*. Texas Instruments, 2024. Datasheet.

[15] ALSA Project. Advanced Linux Sound Architecture. `https://www.alsa-project.org/`. Accessed: 2026-09-30.

[16] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[17] Anthony Williams. *C++ Concurrency in Action*. Manning Publications, 2nd edition, 2019.

[18] Richard G. Lyons. *Understanding Digital Signal Processing*. Pearson, 3rd edition, 2011.