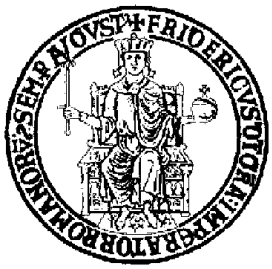
Università degli Studi di Napoli Federico II



Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Classe di Laurea in Ingegneria dell'Informazione, Classe n. L.8

Corso di Laurea in Ingegneria Elettronica

Elaborato di Laurea

Elaborazione di dati audio in real-time tramite l'utilizzo del DMA

Relatore:
Ch.mo Prof. Maresca Luca

Candidato:
Esposito Claudia Antonella
Matr. N43001590

Anno Accademico
2022/2023

INTRODUZIONE

L’ analisi ed elaborazione in tempo reale di segnali audio rivestono un ruolo di fondamentale importanza nell’ambito dell’ingegneria. Lo sviluppo di tecniche volte a catturare, manipolare ed interpretare i segnali audio in tempo reale riveste un ruolo cruciale non solo nelle applicazioni multimediali e di comunicazione, ma ha un vasto impiego in settori diversi, come ad esempio nelle diagnosi mediche.

Questa tesi si propone di implementare un possibile esempio di elaborazione di segnali audio in tempo reale, avvalendosi del microcontrollore STM32F401RE. L’obiettivo primario è implementare una serie di tecniche al fine di migliorare la velocità di esecuzione di tali operazioni.

Nel primo capitolo, si esploreranno le caratteristiche peculiari dei segnali audio, introducendo le varie tecniche di elaborazione e approfondendo il concetto di rumore. Il secondo capitolo prenderà in considerazione approfondita la scheda che verrà utilizzata nel corso della ricerca, con particolare attenzione ai vari componenti interni che saranno impiegati, con uno specifico focus sul componente DMA (Direct Memory Access).

Infine, il terzo capitolo si concentrerà sulla descrizione dei diversi test implementati per la verifica delle tecniche selezionate. Questi test hanno condotto alla progettazione del codice definitivo per l'elaborazione in tempo reale dei dati audio.

Questa tesi rappresenta un passo avanti verso la comprensione e l'applicazione avanzata delle tecniche di campionamento e elaborazione dei segnali audio in tempo reale, con l'obiettivo finale di migliorarne l'efficienza, la velocità e l'esperienza complessiva.

1. I SEGNALI AUDIO

1.1 INTRODUZIONE AI SEGNALI

Con il termine “segnale” si intende una grandezza fisica variabile nel tempo e/o nello spazio a cui è associata un’informazione registrata sotto forma di energia. I segnali possono rappresentare informazioni, come ad esempio la voce umana, la musica o le immagini. I sensori e i trasduttori rappresentano dispositivi che convertono segnali di natura fisica in segnali elettrici, il cui studio risulta più agevole. I sensori rilevano le variazioni di uno stimolo fisico, mentre i trasduttori trasferiscono potenza da un sistema all’altro. La loro classificazione può avvenire in base al fenomeno fisico su cui si basano (misura della lunghezza, della temperatura, della pressione, ecc.). Un esempio, il microfono è un sensore che converte onde sonore in segnali elettrici. In situazioni in cui il sistema di misura potrebbe alterare l'informazione, risulta necessario applicare un filtro per estrarre l'informazione contenuta nel segnale e migliorare il rapporto segnale-rumore. Per "rumore" si intende tutto ciò che non riguarda l'informazione da estrarre. Per incrementare il rapporto segnale-rumore, si applicano algoritmi di cancellazione del rumore e metodi di miglioramento del segnale. L'elaborazione del segnale è un campo complesso e in continua evoluzione. Le tecniche di elaborazione del segnale vengono utilizzate in svariate applicazioni, tra cui telecomunicazioni, informatica, ingegneria, medicina e scienze naturali.

1.2 SEGNALI AUDIO

I segnali audio sono rappresentazione di onde sonore convertite in segnale elettrici, ovvero è la rappresentazione di un suono, costituito dalla modulazione di tensione elettrica. L’onda sonora è prodotta dalla vibrazione di un corpo in un mezzo materiale come aria o acqua. Quando un corpo vibra, trasmette queste vibrazioni alle particelle del

mezzo circostante, creando un'onda elastica detta onda sonora o acustica. Le onde sonore sono onde longitudinali, poiché la direzione di vibrazione delle molecole d'aria è parallela alla direzione della propagazione dell'onda.

Le sorgenti di un'onda sonora possono variare: la vibrazione di una corda di chitarra, l'aria in strumenti a fiato che mette in vibrazione la colonna d'aria, o un altoparlante dove impulsi elettrici inducono la vibrazione di una membrana nell'aria, producendo suono.

In ambito acustico, il segnale audio è quello tradotto da un microfono, quindi rappresenta un'onda acustica. Altoparlanti e cuffie, invece, sono trasduttori che convertono segnali audio elettrici in suoni acustici.

L'analisi dei segnali audio ha rilevanza in diversi campi, come l'applicazione biomedica. Un segnale biomedico acustico è un suono generato o rilevato dal corpo umano, utilizzato per diagnosticare, monitorare e trattare una varietà di condizione mediche. Ad esempio, è possibile studiare i movimenti delle valvole cardiache tramite fonocardiogramma (PCG), oppure le onde sonore possono essere utilizzate nella risonanza magnetica.

Fonocardiogramma [PCG] (movimenti delle valvole cardiache)		20-lOOOHz
Suoni all'auscultazione (caratteri della respirazione)		20-2000Hz
Parola (alterazioni della laringe)		
Suoni di Korotkoff (valutazione collasso di vasi nello sfigmomanometro)		150-500Hz

Figura 1.2.1 Esempi di segnali biomedici acustici e le rispettive frequenze a cui lavorano

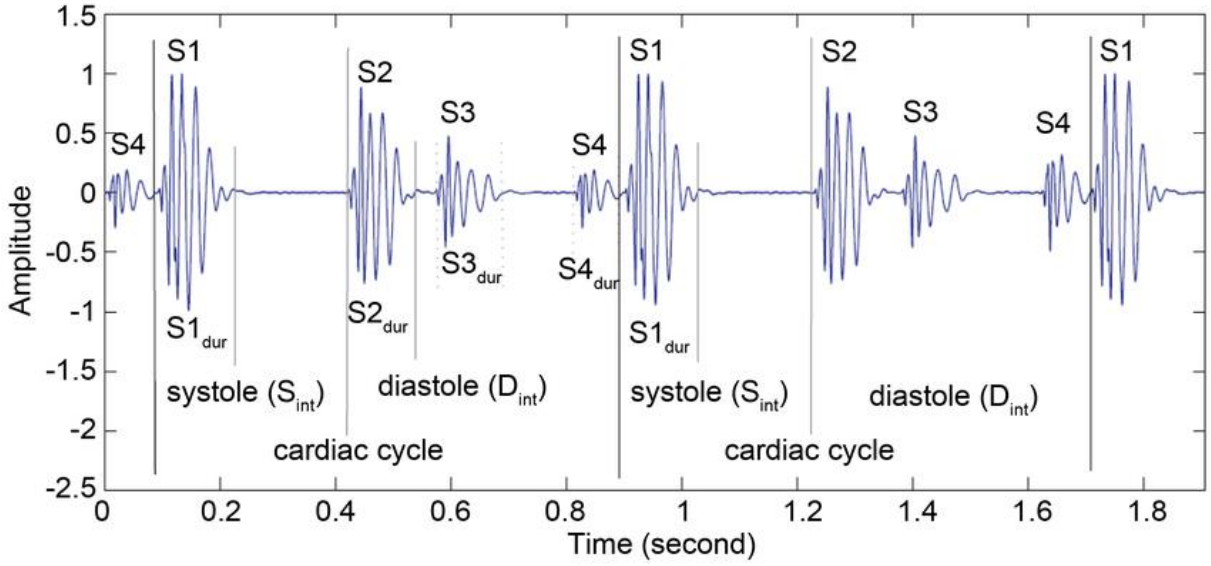


Figura 1.2.2 Esempio di PCG

L'auscultazione cardiaca mediante l'utilizzo di un fonendoscopio riveste un ruolo fondamentale nella rilevazione di numerose patologie cardiache; tuttavia, non si configura come un metodo esaustivo per identificarle tutte. L'efficienza delle diagnosi aumenta tramite le tecniche di elaborazione dei segnali digitali. In Figura 1.2.1 è osservabile un esempio di PCG, ovvero un fonocardiogramma che registra onde acustiche generata dall'azione meccanica delle valvole del cuore. Gli algoritmi possono separare le componenti del segnale cardiaco, individuando le sistoli, diastoli e il soffio cardiaco, primo segno di un cambiamento patologico.

1.3 CARATTERISTICHE SEGNALE AUDIO

Con "flusso di segnale" andiamo a descrivere il percorso del segnale audio dalla sua sorgente, ad esempio dal microfono, ad un dispositivo di registrazione. Questa caratteristica è di fondamentale importanza poiché il flusso spesso è lungo e complesso: il segnale attraversa molte sezioni di una console analogica e apparecchiature audio.

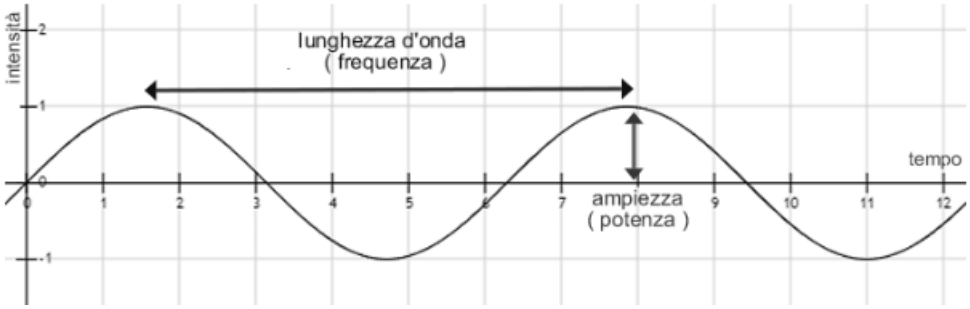


Figura 1.3.1 Rappresentazione dell'onda con evidenziate lunghezza d'onda e ampiezza

La frequenza del segnale rappresenta il numero di cicli completati in un secondo. Viene misurata in Hertz, corrispondente all'inverso di un secondo. La frequenza è inversamente proporzionale alla lunghezza d'onda λ , che è la distanza tra due creste consecutive.

$$f = \frac{1}{\lambda}$$

Quanto minore è la lunghezza d'onda, tanto maggiore è la frequenza. I suoni con frequenza minore sono detti "suoni gravi", mentre i suoni con frequenza maggiori sono detti "suoni acuti".

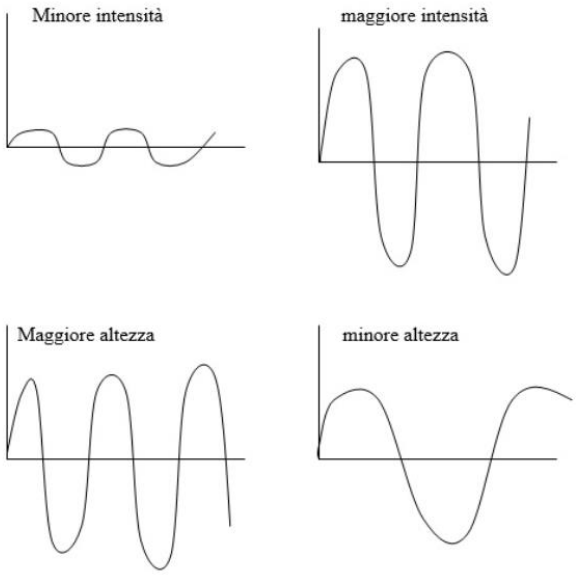


Figura 1.3.2 Esempi schematici di segnali a diversa intensità e altezza

I segnali audio adatti all'udito umano sono caratterizzati da una banda compresa tra 20 Hz e 20kHz. Suoni con frequenze inferiori a 20 Hz, noti come "infrasuoni", vengono percepiti come vibrazioni, mentre suoni con frequenze superiori a 20kHz sono chiamati "ultrasuoni".

Le caratteristiche principali del suono sono : Intensità, Altezza e Timbro. L'intensità acustica è l'ampiezza d'onda sonora ed è definita come il rapporto tra la potenza dell'onda e l'area di superficie attraversata da essa. Questo parametro è indirettamente correlato al volume sonoro, distinguendo suoni deboli da quelli forti. Si misura in $Watt/m^2$, ma viene spesso espressa in decibel per definire il livello di intensità acustica (IL) tramite la seguente formula

$$IL = 10 \log_{10} \frac{I}{Io}$$

Dove Io rappresenta il suono più basso udibile dall'uomo, che è 0 decibel (circa 10^{-12} $\frac{Watt}{m^2}$).

Il timbro del suono determina il "carattere" del segnale sonoro, ed è la sua qualità distintiva. Il timbro è determinato dalla forma dell'onda sonora e dipende dalla vibrazione da cui ha origine il suono. Per esempio, un diapason produce un suono puro noto come armonica (rappresentata come onda sonora A nella Figura 1.3.3). Un'altra sorgente potrebbe emettere un'onda con la stessa lunghezza d'onda e ampiezza, ma di forma diversa (rappresentata come onda B nella Figura 1.3.3). Queste due onde hanno effettivamente timbri differenti.

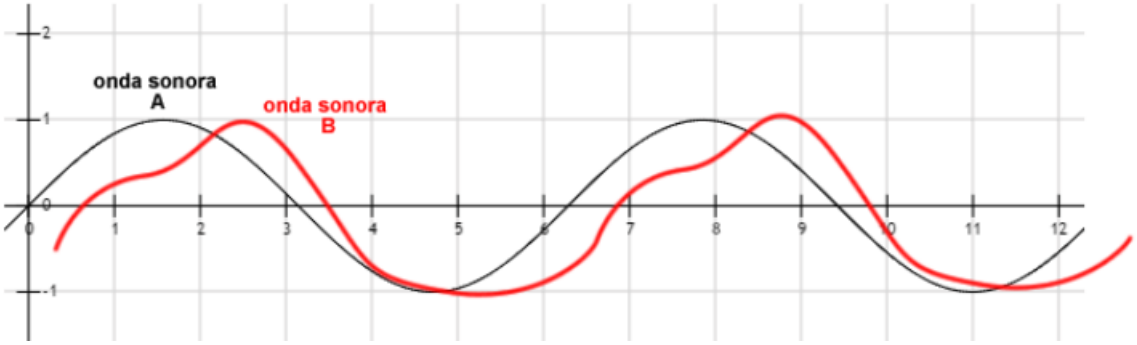


Figura 1.3.3 Esempio di due onde sonore a diverso timbro ma stessa lunghezza d'onda e ampiezza.

Le onde sonore complesse risultano dalla sovrapposizione di onde più semplici, dette armoniche. L'insieme di armoniche definisce il timbro. Le onde possono sovrapporsi in modo regolare o irregolare, ordinato o disordinato, il che contribuisce alla varietà di timbri possibili.

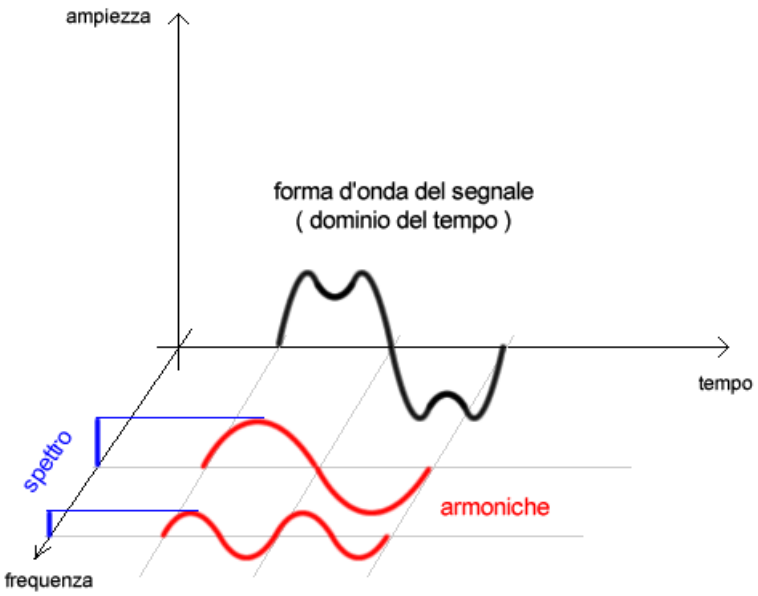


Figura 1.3.4 Rappresentazione su tre assi (tempo, ampiezza, frequenza) di un'onda complessa composta da due armoniche

1.4 STUDIO DEL SEGNALE AUDIO

Il metodo più efficace per codificare un segnale audio consiste nel convertirlo in un segnale elettrico analogico mediante l'uso di un microfono. Il microfono trasforma l'onda sonora meccanica continua in un segnale elettrico in cui i valori di tensione riflettono la forma dell'onda acustica originaria. Questo segnale elettrico analogico può quindi essere convertito in un segnale digitale utilizzando un convertitore analogico-digitale (A/D). Tale convertitore legge il segnale elettrico analogico in punti discreti nel tempo e assegna a ciascun punto un valore numerico. La frequenza di campionamento del convertitore A/D deve essere sufficientemente alta per evitare la perdita di informazione.

Il teorema del campionamento afferma che, se la frequenza di campionamento è almeno il doppio della frequenza massima del segnale audio, il segnale digitale risulta

indistinguibile dal segnale analogico originale. Di conseguenza, il segnale digitale può essere memorizzato, trasmesso o elaborato.

L’analisi audio prevede l’estrazione di informazioni e significato dai segnali audio per scopi di analisi, classificazione ed elaborazione.

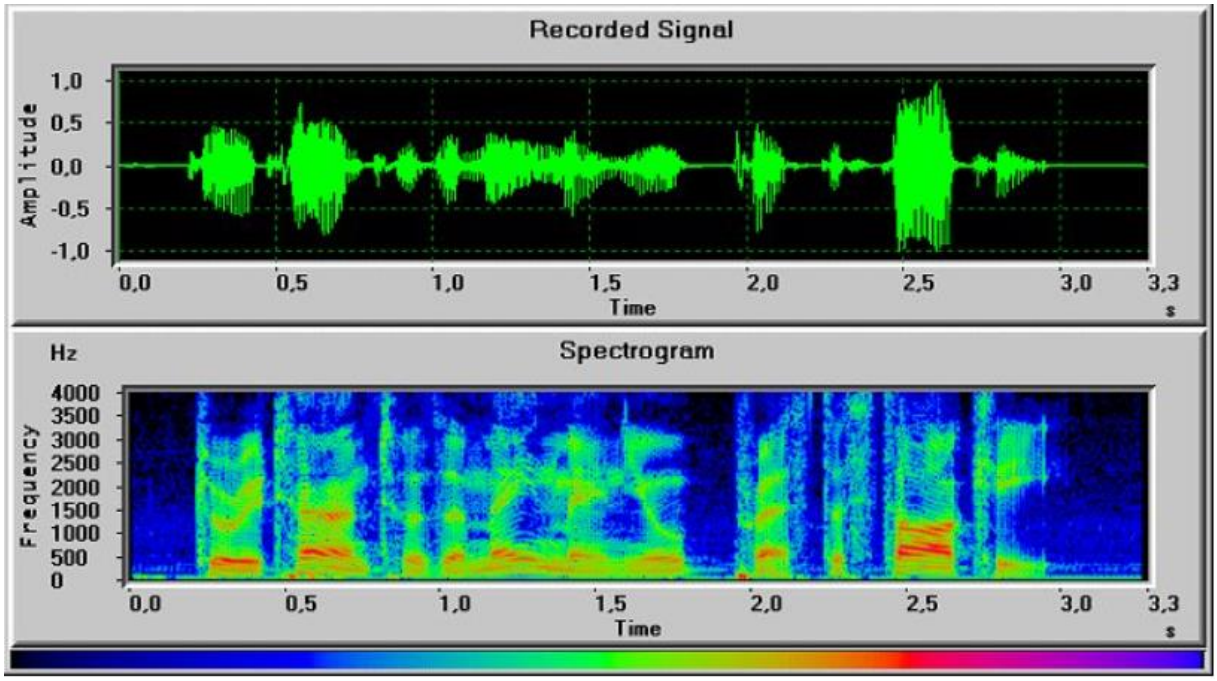


Figura 1.4.1 Analisi nel dominio del tempo dell'ampiezza e della frequenza di un segnale audio

L'analisi nel dominio del tempo si concentra su come il segnale varia nel tempo. Tale analisi può essere utilizzata per osservare caratteristiche quali l’ andamento, la lunghezza d'onda, l’altezza, l’intensità e il timbro. Questa metodologia consente di analizzare la dinamica del segnale e individuare eventuali difetti. Quando si verifica la sovrapposizione di più onde sonore, è utile studiare il segnale attraverso la "trasformata di Fourier rapida" (FFT), la quale scompone il segnale in diverse sinusoidi. Lo spettrogramma di un segnale audio è una rappresentazione visiva dell'evoluzione delle frequenze all'interno di un segnale sonoro nel corso del tempo. Questa forma di visualizzazione trova impiego in svariati campi, tra cui la musica, la linguistica, il sonar, il radar, l'elaborazione del parlato, la sismologia e altre discipline scientifiche.

La creazione di uno spettrogramma mediante l'utilizzo della Trasformata di Fourier Veloce (FFT) è un processo digitale. I dati acquisiti digitalmente nel dominio del tempo vengono suddivisi in blocchi, spesso sovrapposti, e su ciascun blocco viene applicata la

Trasformata di Fourier per determinare la distribuzione spettrale delle frequenze in quel momento specifico. Ogni blocco rappresenta una linea verticale nell'immagine risultante, indicando l'ampiezza delle frequenze rispetto al tempo. Infine, questi spettri vengono combinati o sovrapposti per generare un'immagine bidimensionale o una rappresentazione tridimensionale dello spettrogramma.

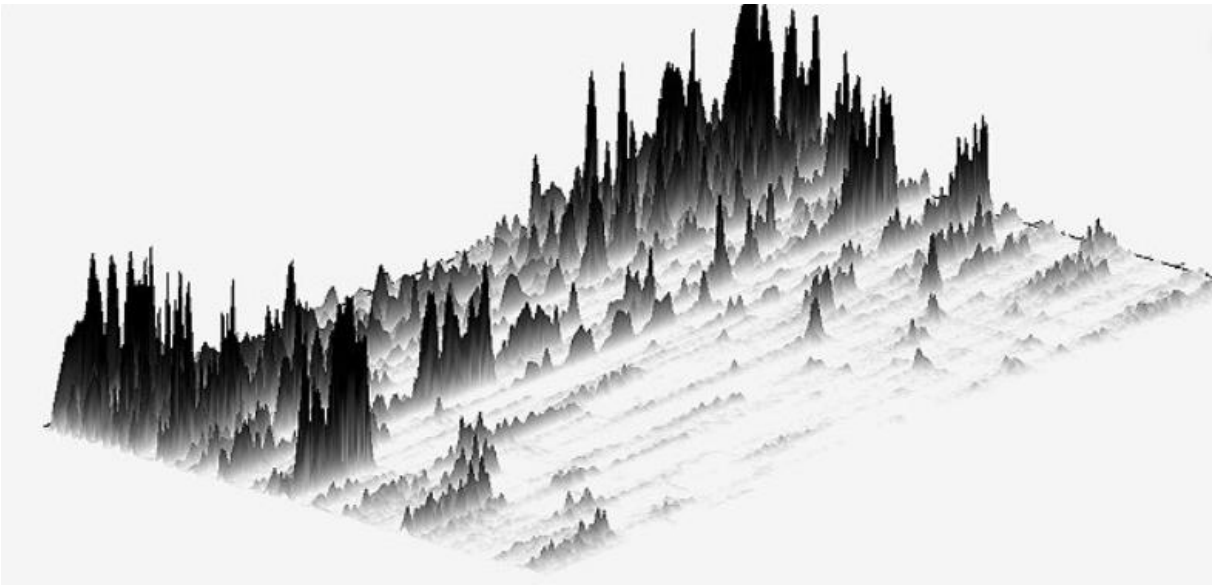


Figura 1.4.2 Spettrogramma di superficie 3D di una parte di un brano musicale

Un possibile utilizzo dello spettrogramma con i segnali biomedici acustici è il “Doppler dell’arteria ombelicale” , ovvero lo spettrogramma del flusso sanguigno, che stima la velocità massima sanguigna nell’arteria ombelicale materna. Non essendo un segnale stazionario richiede l’uso di tecniche adattive per stimare i parametri di interesse su intervalli di tempo ridotti (decine di ms).

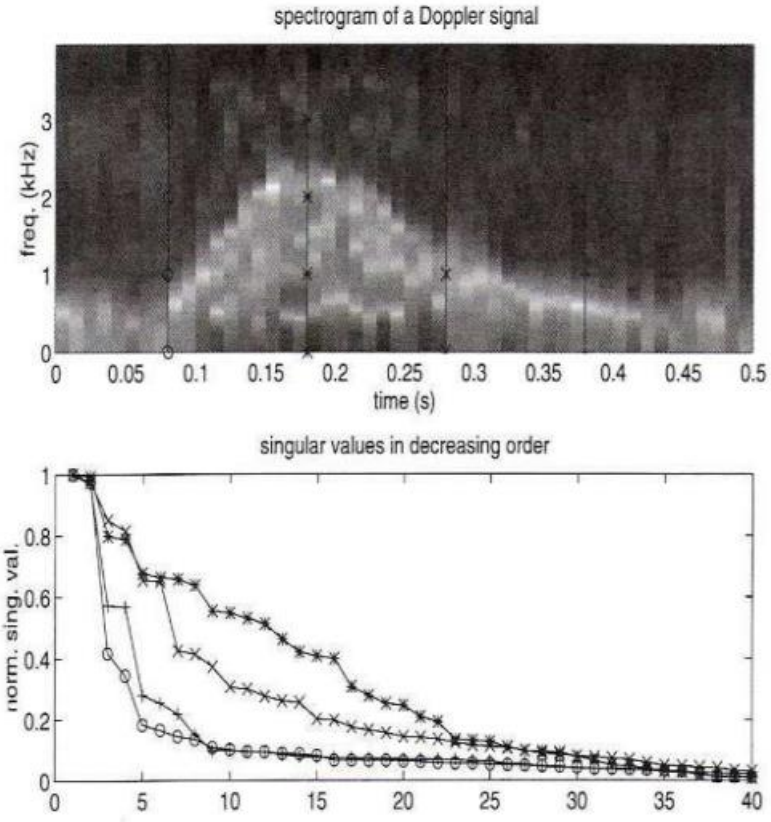


Figura 1.4.3 Esempio di uno spettrogramma di un segnale Doppler di un'arteria ombelicale

1.5 RUMORE

In generale, per estrarre l'informazione contenuta in un segnale audio, è necessario adottare procedure di elaborazione che permettano di aumentare il rapporto segnale-rumore. Il rumore rappresenta tutto ciò che non è rilevante ai fini dell'informazione da estrarre e può derivare da una varietà di fattori, come la distorsione, la compressione e il rumore ambientale. Il rumore è caratterizzato da un'entità imprevedibile e non può essere completamente previsto. Oltre al rumore additivo presente nei segnali stessi, i circuiti elettronici agiscono anch'essi come fonti di rumore. Ciò è dovuto alla natura quantistica, non continua, della corrente elettrica. Il rumore determina un limite inferiore per il segnale registrato in termini di ampiezza e un limite superiore per i guadagni del segnale, identificando il punto in cui avviene il "clipping". Il clipping si verifica quando il segnale di ingresso viene amplificato a un livello eccessivamente alto, causando un'ampiezza del segnale che supera la capacità massima del dispositivo di riproduzione. Ciò comporta la distorsione del suono. Il rapporto segnale/rumore,

detto anche SNR, viene espresso in decibel (dB) ed è definito come il rapporto tra il livello di potenza del segnale e il livello di potenza del rumore. Un elevato rapporto segnale/rumore rende meno udibile il rumore di fondo e aiuta il segnale desiderato a emergere nel mix.
Solitamente, il rumore viene classificato in base ad un colore :

- Rumore rosso : rappresenta un rumore a bassa frequenza ottenuto tramite un filtro passa-basso accentuato.
- Rumore bianco : caratterizzato da una costante ampiezza su tutto lo spettro di frequenze. È un insieme di tutti i possibili toni nello spettro sonoro, con la stessa ampiezza, ma senza periodicità temporale. Questo termine deriva dall'analogia con il fatto che una radiazione elettromagnetica nello spettro visibile apparirebbe come luce bianca agli occhi umani.
- Rumore rosa : noto come rumore 1/f o flicker, è un particolare rumore in cui le componenti a bassa frequenza hanno una potenza maggiore. Di solito, il rumore rosa è prodotto con un filtraggio passa-basso del rumore bianco.

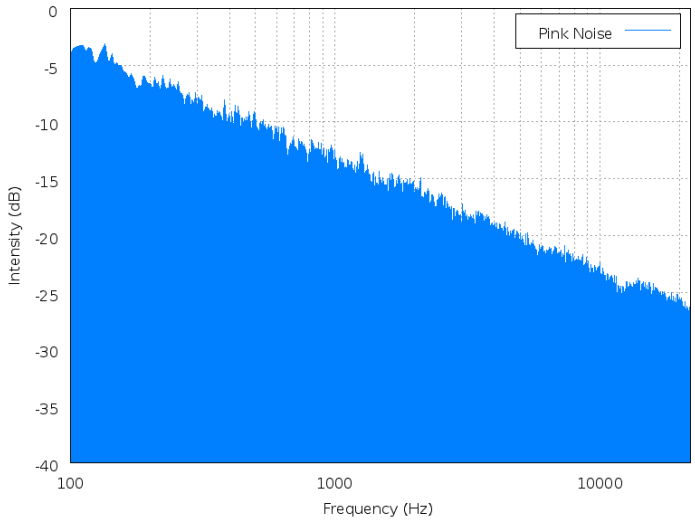


Figura 1.5.1 Visualizzazione dell'intensità in funzione della frequenza per il rumore rosa.

Diventa fondamentale la riduzione del rumore in casi come l'ECG (un segnale biologico non stazionario, che contiene importanti informazioni sul ritmo del cuore) dove l'interferenza della linea elettrica a 50 Hz (PLI) spesso altera il segnale originale durante la registrazione, e i medici sono del tutto incapaci di diagnosticare un paziente cardiaco con questo tipo di segnale rumoroso.

2. HARDWARE E SOFTWARE PER L’IMPLEMENTAZIONE DEL PROGETTO

2.1 INTRODUZIONE AI SISTEMI EMBEDDED

Un sistema embedded, tradotto come “incorporato”, è una forma di elaborazione che si trova all'interno di un oggetto o di un sistema informatico svolgendo funzionalità di monitoraggio e gestione. Tra le diverse categorie di sistemi embedded, i microcontrollori si distinguono per la loro capacità di integrare in un pacchetto le componenti di un microprocessore, tra cui la CPU (Central Processing Unit) e la memoria. Essi offrono diversi pin di ingresso e uscita, oltre vari sensori per interagire con l’ambiente esterno. La programmazione di un microcontrollore avviene attraverso un insieme di istruzioni, costituendo il firmware del sistema che risiede nella memoria del dispositivo. Le interfacce di comunicazione come I2C, SPI e UART consentono lo scambio di dati con altri dispositivi. All'interno di queste componenti, il microprocessore ospita la CPU, che svolge il ruolo di "cervello" del microcontrollore, responsabile delle istruzioni e del controllo delle operazioni di sistema. Il Bus, costituito da un insieme di linee, funge da canale di comunicazione per il trasferimento di dati. I Timer vengono utilizzati per la misura di intervalli di tempo e sono fondamentali per la sincronizzazione dei dispositivi tramite il Clock, un segnale digitale che oscilla tra i livelli basso e alto in modo continuo. I microcontrollori possono essere da 8-bit,16-bit o 32-bit, dove questi numeri si riferiscono alle dimensioni dei bus. In presenza di un bus più grande posso eseguire calcoli più complessi.

2.2 MICROCONTROLLORE STM32F401RE

I microcontrollori STM32F401RE, prodotto dall'azienda STMicroelectronics, sono parte del range dei dispositivi della STM32 per efficienza dinamica. Questi dispositivi offrono il miglior equilibrio tra consumo di potenza e performance. In particolare, i dispositivi STM32F401RE sono basati sul core RISC ARM® Cortex®-M4 ad alte prestazioni a 32 bit che opera a una frequenza fino a 84 MHz. Un microprocessore è l'unità di elaborazione principale all'interno del microcontrollore, responsabile dell'esecuzione delle istruzioni e delle operazioni di calcolo. Rappresenta il cuore del microcontrollore e determina in gran parte la sua capacità di elaborazione e funzionalità. Il suo core Cortex®-M4 include un'unità di punto mobile (FPU) a precisione singola che supporta tutte le istruzioni e i tipi di dati a precisione singola ARM. In aggiunta, è incorporato un completo set di istruzioni per il trattamento di segnali digitali (DSP) e un'unità di protezione della memoria (MPU) per potenziare la sicurezza delle applicazioni. Includono memorie incorporate ad alta velocità (512 Kbyte di memoria Flash, 96 Kbyte di SRAM) e una vasta gamma di I/O e periferiche potenziate, collegate a due bus APB, due bus AHB e una matrice di bus multi-AHB a 32 bit. Tutti i dispositivi offrono un convertitore ADC a 12 bit, un RTC a basso consumo energetico, sei timer a 16 bit per uso generale, incluso un timer PWM per il controllo del motore, e due timer a 32 bit per uso generale.

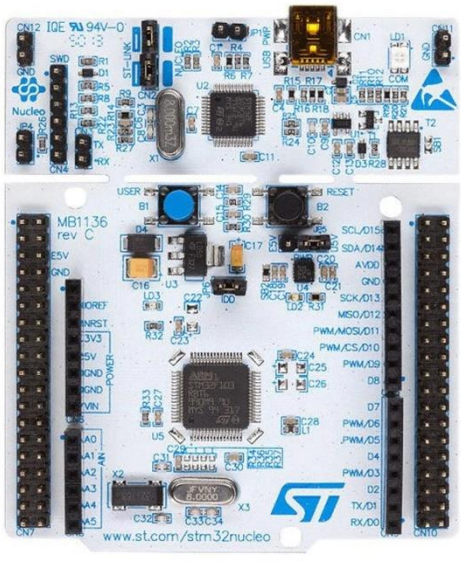


Figura 2.2.1 Microcontrollore STM32F401RE

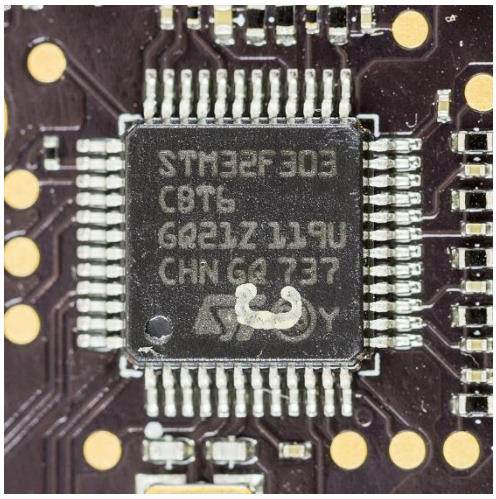


Figura 2.2.2 Microprocessore Arm based Cortex M4

2.3 BUS

È possibile stabilire una connessione tra due sistemi, all'interno dello stesso microcontrollore, attraverso l'uso di dispositivi, circuiti o architetture appositamente progettati per garantire una comunicazione accurata tra di loro. I bus nelle schede STM (STMicroelectronics) sono strutture di comunicazione interne che consentono il trasferimento di dati, indirizzi e segnali di controllo tra diverse componenti e periferiche all'interno del microcontrollore. Il bus di sistema è costituito da un insieme di connessioni in rame, di solito da 50 a 100, incisi sulla scheda madre, e presenta connettori distanziati uniformemente per consentire il collegamento dei moduli di memoria e dei dispositivi di input/output.

Nella scheda che utilizziamo, il sistema è composto da un AHB Bus Matrix multi-layer a 32 bit, basato sulla tecnologia AHB (Advanced High-Performance Bus) di ARM. Rappresenta un sistema di interconnessione avanzato che permette la comunicazione e il trasferimento di dati tra diverse unità funzionali all'interno del microcontrollore. Supporta fino a 32 bit di larghezza in un singolo ciclo bus. Multi-layer si riferisce alle diverse priorità e livelli di accesso supportati dal Bus Matrix. Permette l'accesso da un master ad uno slave, abilitando l'accesso concorrente ad alte velocità quando ho più periferiche che lavorano simultaneamente.

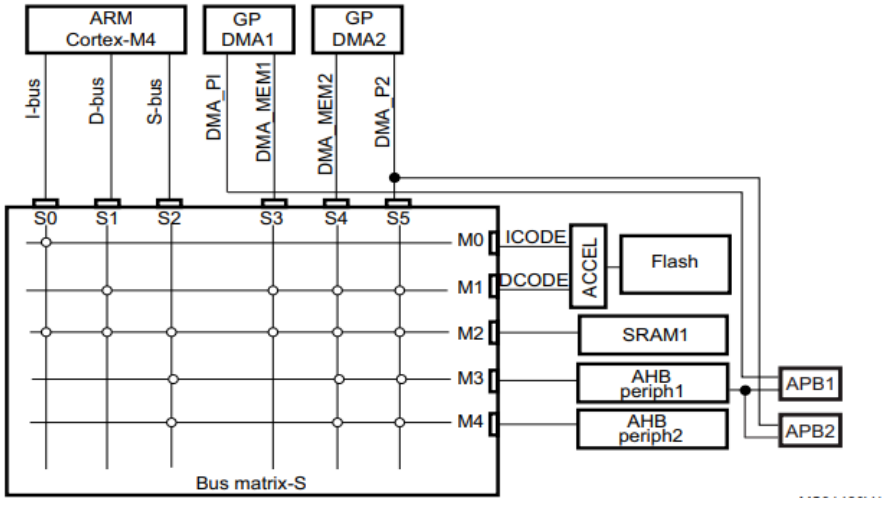


Figura 2.3.1 Architettura del Bus Matrix.

2.4 ADC

La scheda STM32F401RE è costituita da un convertitore analogico-digitale ad approssimazioni successive a 12 bit. L'ADC ad approssimazioni successive è una tecnica di conversione che valuta un segnale analogico attraverso un processo di confronto e approssimazione sequenziale per determinare il valore digitale corrispondente.

La conversione di un segnale da analogico a digitale è costituita da tre blocchi: campionamento, quantizzazione e codifica. Il campionamento è la discretizzazione dell'asse temporale del segnale analogico, mentre la quantizzazione rende discreti i valori che può assumere il segnale e infine la codifica che converte la sequenza numerica in un flusso di bit.

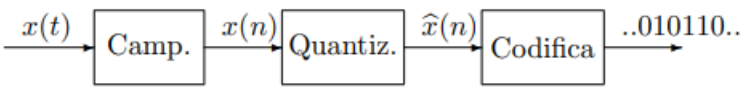


Figura 2.4.1 Schema a blocchi della conversione analogico-digitale

Il campionamento consiste nel convertire un segnale continuo nel tempo in un segnale discreto, valutando l'ampiezza a intervalli temporali regolari. L'intervallo che intercorre tra una valutazione e l'altra si chiama intervallo temporale di campionamento e di conseguenza, la frequenza di campionamento $f_c = \frac{1}{T}$ è il reciproco dell'intervallo temporale, misurata in campioni al secondo o Hertz. Il modo più semplice è tramite un treno di impulsi ideali, grazie all'importante proprietà che permette di estrarre il valore del segnale nel punto in cui è centrato l'impulso.

$$x(t)\delta(t - t_0) = x(t_0)\delta(t - t_0)$$

Moltiplicando il segnale per un treno di impulsi periodico al periodo di campionamento T, ottengo il segnale campionato, che sarà un treno di impulsi le cui ampiezze rappresentano i campioni di x(t) ad intervalli equispaziati da T.

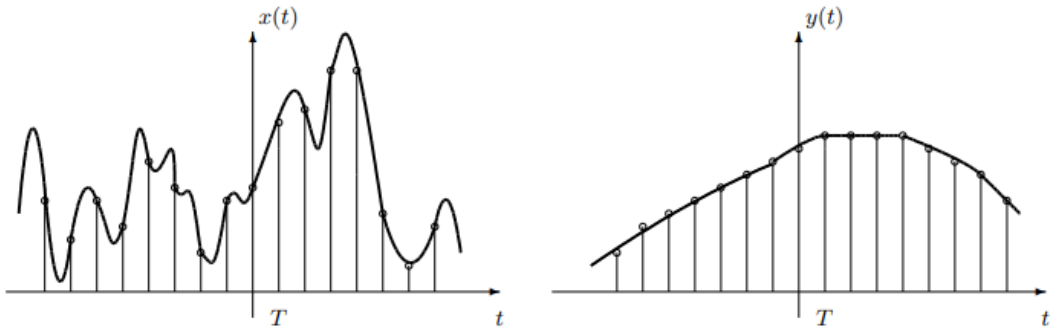


Figura 2.4.2 Schema di campionamento di un segnale analogico.

Per capire a quale frequenza di campionamento io non perdo informazioni, devo vedere nel dominio della frequenza.

La trasformata di Fourier di un treno di impulsi è ancora un treno di impulsi e ottengo lo spettro del segnale campionato che è data dalla sovrapposizione di X(f) centrate in 1/T.

$$X_\delta(f) = X(f) * \frac{1}{T}\sum_{k=-\infty}^{\infty} \delta\left(f - \frac{k}{T}\right) = \frac{1}{T}\sum_{k=-\infty}^{\infty} X\left(f - \frac{k}{T}\right) = f_c \sum_{k=-\infty}^{\infty} X(f - k\, f_c)$$

Facendo un esempio con X(f)= λ(f/B), la trasformata di Fourier la mostro in 3 casi ovvero

a) f_c>2B b) f_c=2B c) f_c<2B

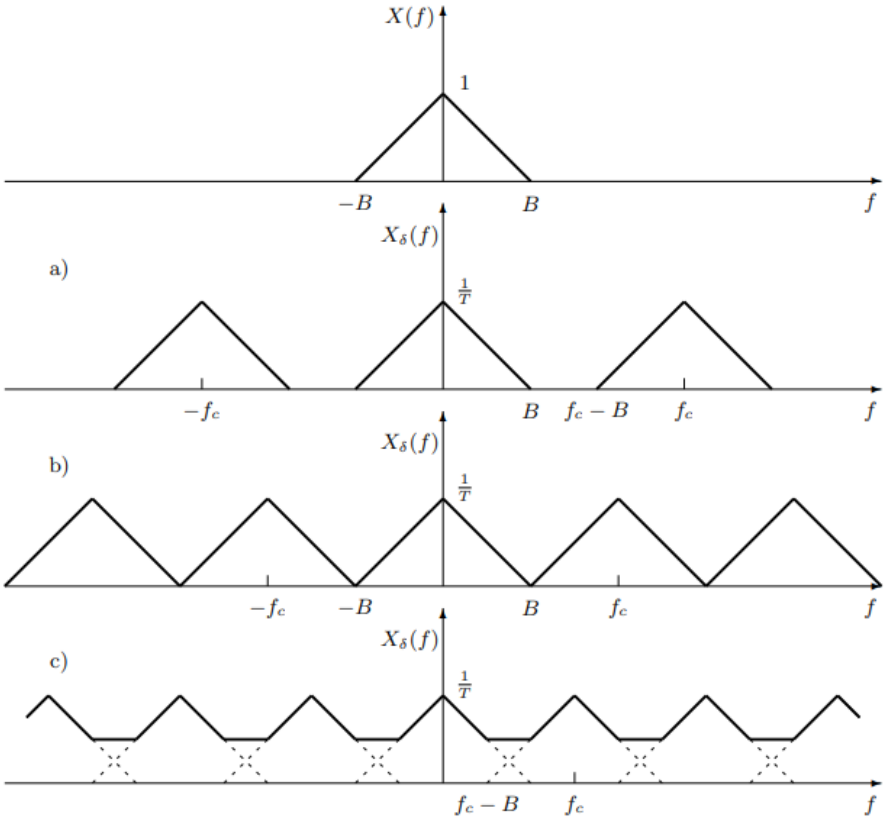


Figura 2.4.3 Trasformata di Fourier del segnale nei 3 diversi casi citati

Considerando l'ipotesi fondamentale che il segnale abbia uno spettro con banda rigorosamente limitata, si vede che la migliore condizione per recuperare il segnale sono i primi due casi dove le repliche non si sovrappongono tra di loro perciò $f_c \geq 2B$

Il teorema del campionamento (detto di Nyquist-Shannon) afferma che, per campionare correttamente (senza perdite di informazioni) un segnale a banda limitata la frequenza di campionamento deve essere proporzionale alla banda del segnale, ovvero è sufficiente campionarlo con una frequenza di campionamento pari almeno al doppio della banda del segnale (detta frequenza di Nyquist). Se sarà maggiore della frequenza di Nyquist ho il sovracampionamento, mentre se sarà minore si parla di sottocampionamento, in cui si verifica il cosiddetto fenomeno dell'aliasing, per cui le repliche adiacenti a quella centrale creano una distorsione che non può essere compensata in alcun modo.

Il campionamento genera un segnale tempo discreto le cui ampiezze rappresentano numeri reali caratterizzati da infinite cifre significative, perciò è necessario discretizzare questi valori. Nella Quantizzazione vengono raggruppati in un determinato numero di

insiemi, detti valori di quantizzazione, ognuno dei quali costituito da un unico valore detto livello di restituzione. L'operazione consiste proprio nell'approssimare ogni numero reale con una rappresentazione a precisione finita, che mi comporta una perdita di informazioni, che non può essere recuperata.

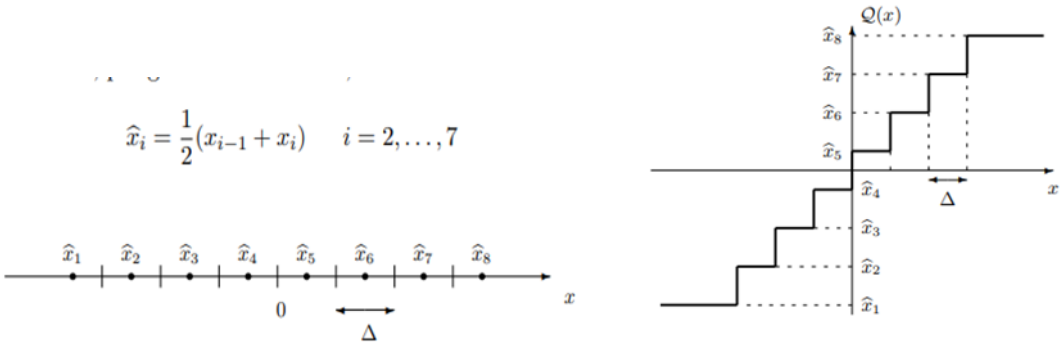


Figura 2.4.4 Esempio grafico di quantizzazione con 8 bit.

La codifica associa alla sequenza di valori in ingresso un flusso di bit. Dove il numero di bit b è dato dal valore del quantizzatore M=2^b

Livello	Codice
$\widehat{x}_0$	000
$\widehat{x}_1$	001
$\widehat{x}_2$	010
$\widehat{x}_3$	011
$\widehat{x}_4$	100
$\widehat{x}_5$	101
$\widehat{x}_6$	110
$\widehat{x}_7$	111

Figura 2.4.5 Esempio codifica ad 8 bit.

Gli ADC sono caratterizzati da una risoluzione, ovvero la minima variazione della grandezza analogica che provoca una variazione di un LSB(Least Significant Bit) nel numero di uscita : tale variazione detta Q che coincide con la risoluzione

$$Q = \frac{V_{FS}}{2^n}$$

Dove V_{FS} è la tensione del fondo scala, cioè una tensione fornita al convertitore che è il massimo valore in ingresso convertibile in binario. Perciò se l'ADC ha una risoluzione a 12 bit, in uscita saranno rappresentati N= 2^{12}=4096 livelli compreso lo 0.

2.5 TIMER E PWM

I timer della scheda STM32F401RE sono fondamentali per la misurazione del tempo e generazione di eventi periodici. Il loro funzionamento è quello di contare (in modalità up o down). Ad esempio, un timer a 8 bit conta da 0 a 255, per poi ripartire da 0. Questo dispositivo può funzionare in diverse modalità, come "Output Compare," che effettua una conta fino a un valore specifico, o in modalità PWM, cioè "Pulse Width Modulation". Quando il timer raggiunge il valore presente nell'ARR (Auto Reload Register), il pin in uscita associato al timer passa ad uno stato logico alto. Rimarrà in questo stato fino a quando il timer non raggiunge il valore nel registro CCRx. Successivamente l'uscita passerà ad un valore logico basso e il ciclo si ripete. La frequenza di questo segnale è determinata dal clock interno, dal Prescaler (CP) e dal valore nel registro ARRx (Auto Reload Register). Il Duty Cycle del segnale PWM è definito dal valore nel registro CCRx.

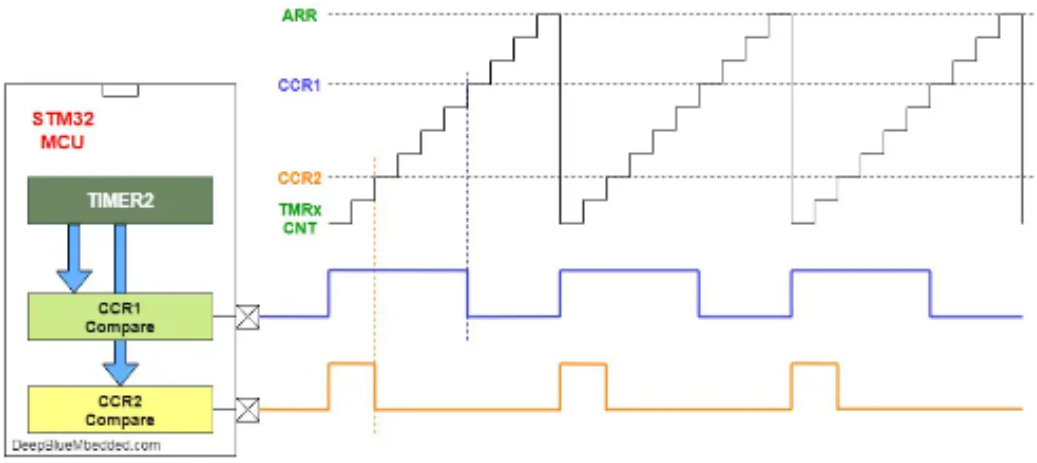


Figura 2.5.1 Schema funzionamento PWM.

La frequenza del PWM è data da

$$F_{PWM} = \frac{Fclock}{(ARR + 1)(CP + 1)}$$

E il Duty Cycle è dato dalla seguente formula

$$DC = \frac{CCR1}{ARR} \cdot 100\%$$

Un Duty Cycle del 10% intende la durata dell'impulso rimane ad un livello logico alto per 10% del periodo, mentre il 90% del periodo avrà un valore logico basso.

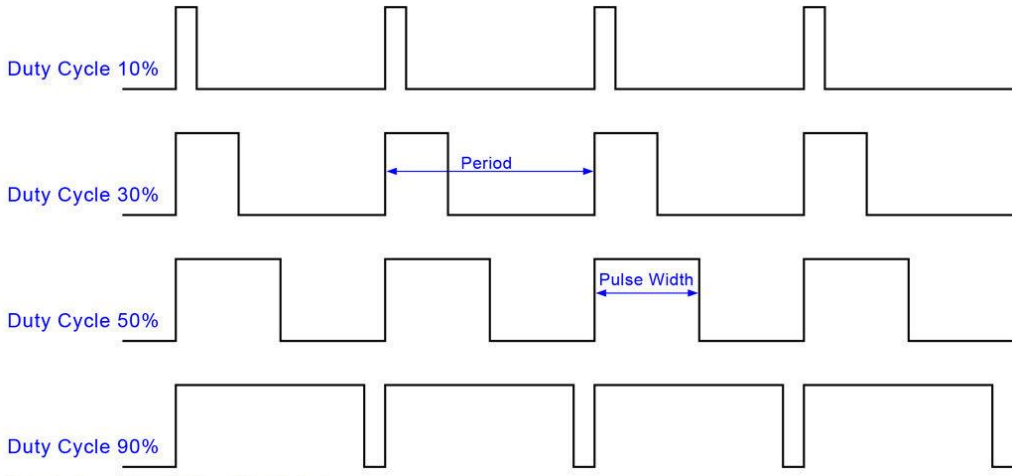


Figura 2.5.2 Esempi di segnali PWM con diversi Duty Cycle

Un importante proprietà è la risoluzione, ovvero il numero di livelli discreti che può assumere il Duty Cycle e si misura in bit. Più la frequenza di clock è alta, più la risoluzione PWM aumenta. È importante notare che una frequenza di clock più alta consuma anche più energia.

$$Risoluzione_{PWM} = \frac{\log \frac{f_{clock}}{f_{PWM}}}{\log 2} = \frac{\log \frac{84MHz}{2000Hz}}{\log 2} \sim 15\ bits$$

2.6 DMA

DMA, acronimo di "Direct Memory Access", rappresenta una tecnologia che agevola il trasferimento diretto dei dati dalla periferica alla memoria principale, senza richiedere l'intervento della CPU. Quando si tratta di spostare notevoli volumi di dati, questo

approccio evita di sovraccaricare il processore, che altrimenti sarebbe coinvolto in ogni singolo byte di trasferimento, potenzialmente rallentando il suo ciclo di elaborazione. La CPU è connessa a diverse periferiche e, ad esempio, quando si invia un dato da una periferica all'altra, la CPU legge tale dato dal registro dati della prima periferica e lo instrada verso la seconda periferica o la memoria. Nelle moderne architetture di microcontrollori, è incluso un controller DMA che può essere configurato per orchestrare il trasferimento automatico di grandi quantità di dati tra periferiche, senza richiedere l'esecuzione diretta del codice.

I moduli STM32 sono dotati di 2 DMA, ognuno con 2 porte, una periferica e una collegata alla memoria, tale che possono lavorare simultaneamente. Ogni controllore DMA offre 8 flussi (streams), ognuno dei quali rappresenta un percorso per il trasferimento di dati separati dedicato alla gestione delle richieste di accesso alla memoria da parte di uno o più dispositivi periferici. Ogni Stream è suddiviso in 8 canali selezionabili tramite software, detti anche “richieste”, ognuno dei quali è associato ad una periferica che può iniziare la richiesta di trasferimento dati. Per il funzionamento dello Stream, solo un canale per volta può essere attivo. Ogni porta DMA ha un arbitro per gestire la priorità tra i diversi Stream, dove la priorità è configurabile tramite software (ci sono 4 livelli: bassa, media, alta e molto alta). Se due Stream dello stesso DMA hanno la stessa priorità, saranno utilizzate le priorità dell’hardware, ovvero Stream 0 ha priorità sullo Stream 1.

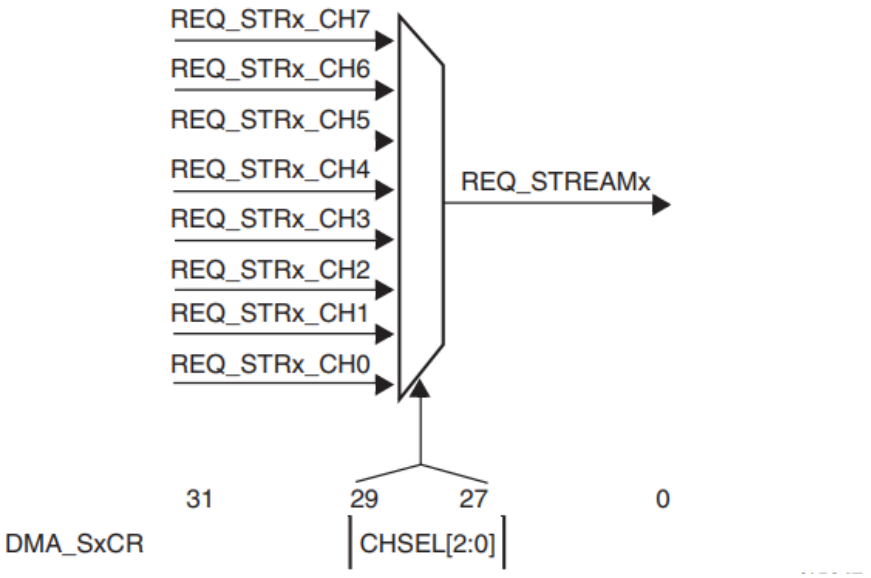


Figura 2.6.1 Esempio di selezione dei canali di uno Stream

Esiste un “Request Mapping”, che associa richieste DMA a dispositivi, periferiche o flussi di dati. È importante per l’ottimizzazione dei trasferimenti dati nel sistema e può influire sulla velocità e sulle prestazioni complessive . Il “DMA Request Mapping” è disegnato per offrire al software maggiore flessibilità nella rappresentazione di ciascuna richiesta del DMA in relazione alla richiesta della periferica

Il DMA opera in 3 modalità di trasmissione:

1. Da periferica a memoria
2. Da memoria a periferica
3. Da memoria a memoria

Ogni trasferimento dati richiede un indirizzo di origine, uno di destinazione ed entrambi dovrebbero essere presenti nella memoria dell’AHB. La dimensione dei dati è definita dal registro DMA_SxNDTR. Il valore della dimensione si riduce rispetto alla quantità dei dati trasferita, ovvero se la dimensione iniziale del trasferimento sarà di 100 byte e trasferisco un burst (insieme di dati) di 32 byte, la dimensione sarà ridotta a 68 byte per riflettere i dati già inviati. Questo processo continua fino a quando la dimensione è 0, ovvero il trasferimento è completo.

2.6.1 ARCHITETTURA BUS

Il controllore DMA opera direttamente condividendo il sistema bus con il Cortex®-M4 core. Quando il DMA effettua una richiesta, può temporaneamente interrompere l'accesso della CPU al bus di sistema per alcuni cicli, specialmente quando CPU e DMA hanno la stessa destinazione. Per questo, la matrice del bus implementa uno schema di pianificazione Round-Robin, garantendo almeno la metà della larghezza di banda del bus di sistema per la CPU. Il DMA è un AMBA Advanced High-Performance Bus(AHB), dove AHB è uno dei bus più comuni della famiglia di interfacce di bus AMBA(Advanced Microcontroller Bus Architecture) sviluppata da Arm Holding per consentire la comunicazione tra componenti all’interno di un sistem-on-a-chip(SoC).

L’AHB è un bus ad alte prestazioni utilizzato per la comunicazione tra diverse componenti del sistema come la CPU, dispositivi di memoria, controller DMA e altri dispositivi periferici. Esso offre un'interconnessione efficiente e a bassa latenza tra tali

componenti, agevolando lo scambio veloce e affidabile di dati e segnali di controllo. Ciò permette il funzionamento sincronizzato e la comunicazione a elevate velocità. Il DMA presenta 2 tipologie di porte AHB, ovvero le interfacce di connessione che agevolano la comunicazione all'interno del sistema tra i componenti collegati al bus AHB. Posso avere:

1. Porta Slave: una porta di I/O che permette ad un dispositivo di essere controllato e accessibile dal bus AHB. Può ricevere comandi di lettura o scrittura da altre unità.
2. Porta Master: una porta di output che abilita al DMA di controllare i trasferimenti di dati tra i vari moduli slave. Più moduli master possono essere collegati allo stesso bus, e ognuno di essi può avviare operazioni di lettura/scrittura.

Sulla scheda, il DMA è caratterizzato da una Slave Port e due porte Master, che permettono di inizializzare il trasferimento di dati tra i moduli Slave. Durante questa operazione, il processore Cortex-M4 può eseguire altre attività ed è interrotto solo se un blocco di dati è disponibile per essere elaborato. Il controllore DMA utilizza un sistema di bus multi-layer al fine di garantire una bassa latenza per i trasferimenti di DMA. Ogni controllore DMA è dotato di due porte AHB. Una di queste è l' "AHB Memory Port", connessa alle memorie, e l'altra è la "AHB Peripheral Port" connessa alle periferiche. Queste porte possono essere utilizzate simultaneamente con il sistema di Master tramite il Matrix Bus esterno e percorsi dedicati.

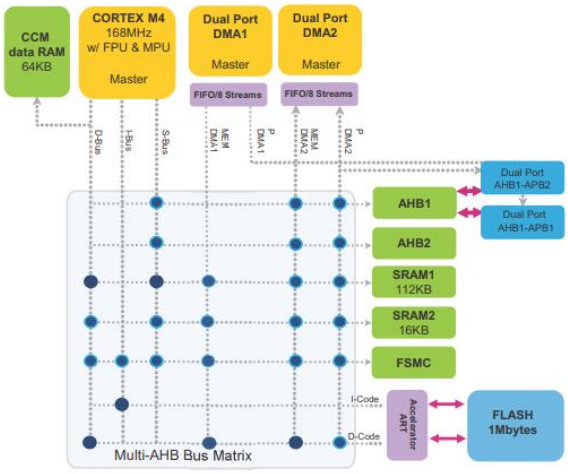


Figura 2.6.1.1 Schema del DMA Dual Port.

Gli accessi per i due controller DMA della scheda sono differenti.

Per il DMA1:

- La porta MEM(Memory Port) può accedere al AHB3 e alla memoria Flash tramite Bus Matrix.
- La porta periferica può accedere ad AHB-to-APB1 tramite un percorso diretto.

Per quanto riguarda il DMA2:

- La porta MEM(Memory Port) può accedere ad AHB1,AHB2,AHB3 e alla memoria Flash tramite il Bus Matrix.
- La porta periferica può accedere a : AHB1,AHB2,AHB3,SRAM,Flash memory tramite Bus Matrix e ad AHB-to-APB2 tramite un percorso diretto

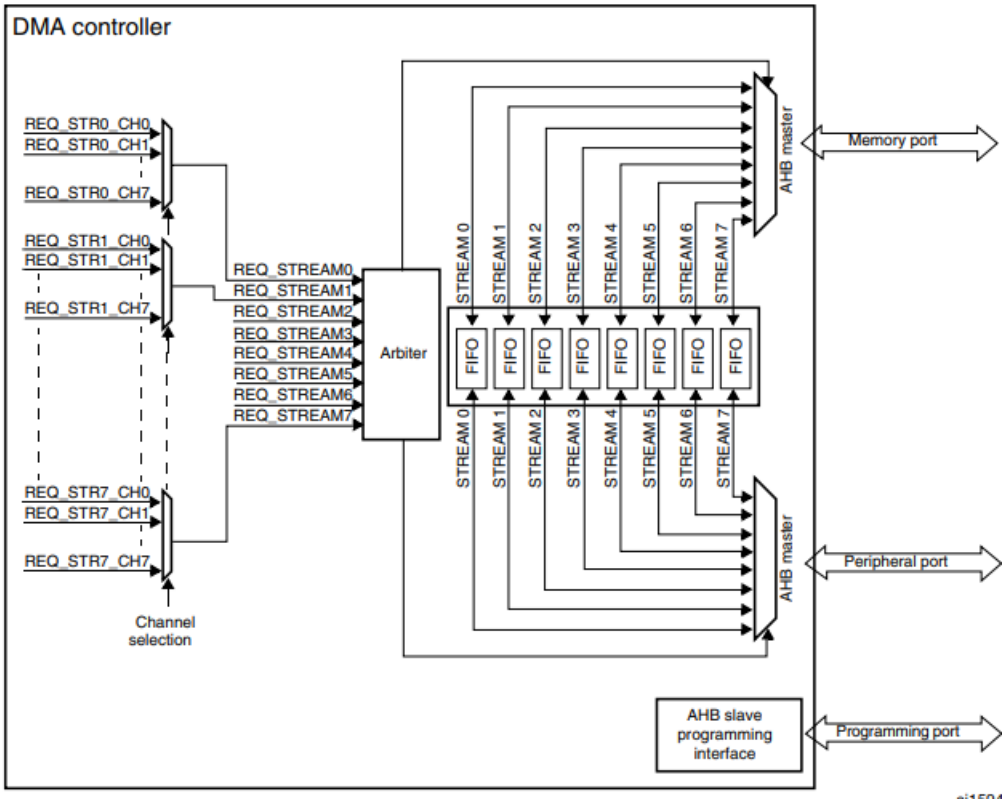


Figura 2.6.1.2 Block Diagram del DMA Controller

Il Multi-Layer Bus Matrix consente ai moduli Master di trasferire dati simultaneamente verso diversi moduli Slave. Ciò favorisce il parallelismo nel trasferimento dati, contribuendo a ridurre il tempo di esecuzione e ottimizzando l'efficienza del DMA, nonché il consumo di potenza. Quest'architettura è caratterizzata da:

1. AHB Master: un bus master è in grado di inizializzare la lettura e scrittura delle operazioni. Solo un Master può ottenere il controllo di questo bus per un periodo di tempo definito.
2. AHB Slave : risponde al master per operazioni di lettura e scrittura. Questo bus risponde con uno stato di successo, fallimento o attesa.
3. AHB Arbiter : assicura che solo un master può inizializzare un'operazione di scrittura e lettura alla volta. Utilizza un algoritmo di Round-Robin.
4. AHB Bus Matrix: un bus multi-layer che connette AHB Masters a AHB Slaves con un AHB Arbiter per ogni "strato".

Lo schema "Round-Robin Priority" è un algoritmo utilizzato per implementare le priorità del Bus Matrix per assicurarsi che ogni Master possa accedere ad ogni Slave a bassa latenza. Gli intervalli di tempo, noti come "quanti di tempo", sono assegnati a ciascun processo in porzioni uguali e in modo circolare, trattando tutti i processi senza priorità (detto ciclo esecutivo). Permette una equa distribuzione della larghezza banda dei bus, con latenza massima limitata.

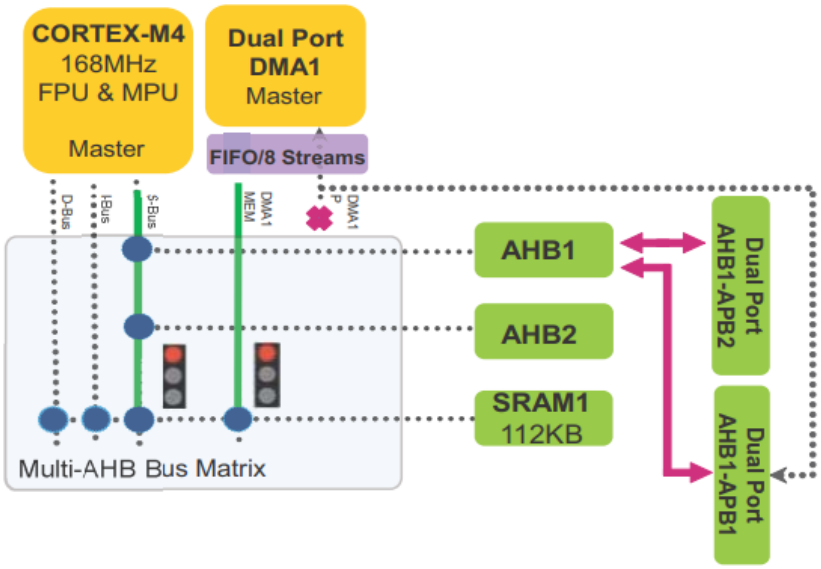


Figura 2.6.1.3 Schema del Multi-AHB Bus Matrix rappresentante DMA e CPU che richiedono contemporaneamente accesso al SRAM1

Se ho accesso concorrente, come illustrato in Figura 2.6.1.3 in cui sia la CPU che il DMA richiedono entrambi l'accesso a SRAM1, diventa necessario il Bus Matrix

Arbiter, che applicherà l'algoritmo Round-Robin per risolvere il problema: se l'ultimo Master ad ottenere il bus è stata la CPU, sarà poi la DMA1 ad accedere alla SRAM1. Da questo si deduce che la latenza di trasferimento associata ad un Master dipende dal numero di altre richieste pendente di accesso allo stesso Slave AHB. Infatti, se avessi 5 Master ad accedere simultaneamente alla SRAM1, la latenza associata al DMA1 per ottenere di nuovo l'accesso al SRAM1, è uguale al tempo di esecuzione di tutte le richieste pendenti dagli altri Master, come si vede in Figura 2.6.1.4.

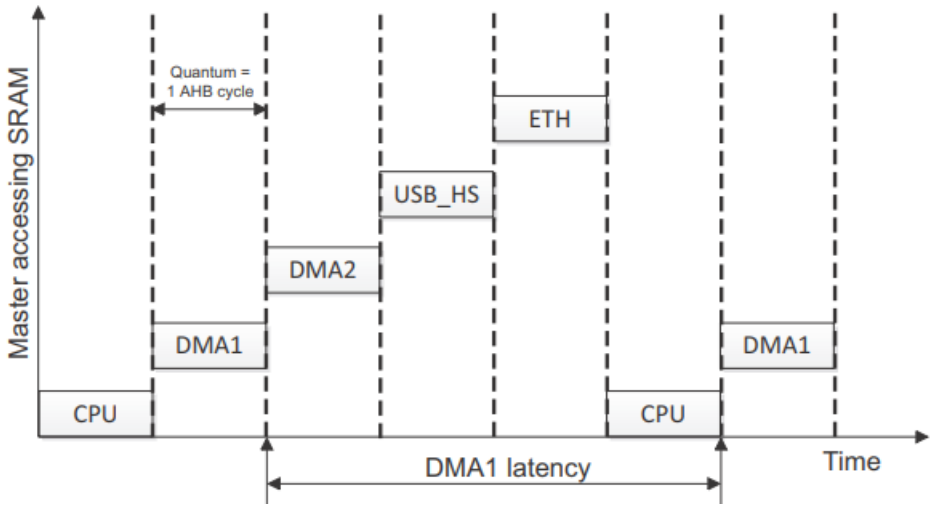


Figura 2.6.1.4 Grafico rappresentante la latenza del DMA1, con l'asse della x rappresentante il tempo e l'asse y i diversi componenti che accedono al SRAM1

Riprendendo l'esempio in Figura 2.6.1.3, dove DMA1 e CPU competono per l'accesso a SRAM1, la latenza sul trasferimento DMA varia anche in base alla lunghezza della transazione della CPU, che può bloccare l'AHB Bus, per un intervallo di tempo che può variare da 1 AHB ciclo per ogni singolo operazione di lettura/scrittura a N AHB cicli, dove N è il numero di data word usate dalla CPU.

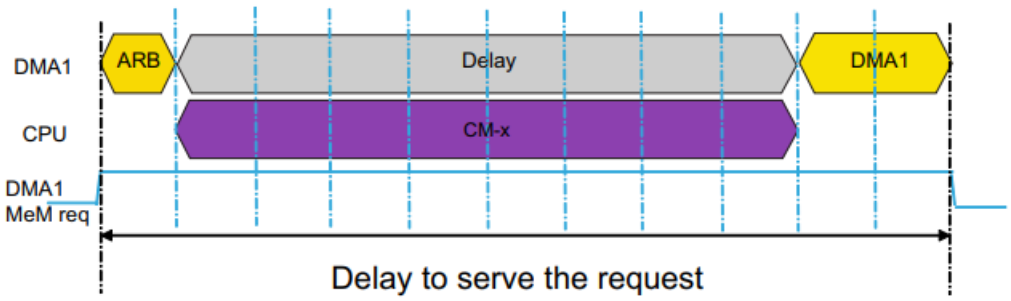


Figura 2.6.1.5 Latenza dell'accesso del DMA al SRAM1 dato dall'interrupt della CPU che richiede 8 AHB cicli, rappresentati dai segmenti verticali

È stato già spiegato che per gestire le otto richieste provenienti da uno Stream, il DMA incorpora un arbitro per avviare le sequenze in base alle loro priorità per ciascuna delle due AHB Master Port. Quando più di una richiesta DMA è attiva, il DMA deve arbitrare internamente tra le richieste attive per determinare quale richiesta debba essere soddisfatta prima. Nella figura seguente sono mostrate due richieste DMA circolari attivate simultaneamente dai flussi DMA "richiesta 1" e "richiesta 2". Al ciclo successivo dell'orologio AHB, l'arbitro DMA verifica le richieste pendenti attive e concede l'accesso al flusso "richiesta 1" che ha la priorità più alta. Il ciclo di arbitrato successivo avviene nell'ultimo ciclo di dati del flusso "richiesta 1". In questo momento, "richiesta 1" viene mascherata e l'arbitro rileva solo "richiesta 2" come attiva, riservando l'accesso a "richiesta 2" in tale occasione, e così via.

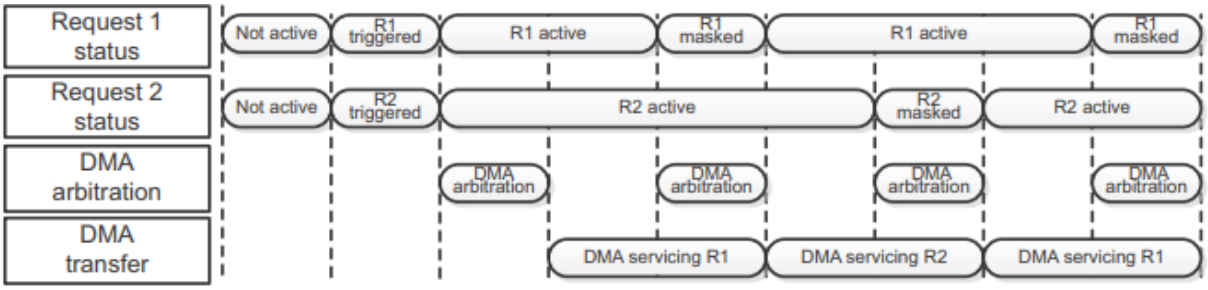


Figura 2.6.1.6 Esempio di richieste DMA

La scheda STM32F401RE dispone di due ponti AHB-to-APB, APB1 e APB2, per connettere le diverse periferiche.

Il ponte AHB-to-APB è un'architettura a due porte che permette l'accesso attraverso due diversi percorsi:

1. Un percorso diretto, che non coinvolge il Bus Matrix, può essere generato dal DMA1 al APB1 O dal DMA2 al APB2. In questo caso l'accesso non è penalizzato dall'arbitro del Bus Matrix.
2. Un percorso comune, attraverso il Bus Matrix, che può essere generato dalla CPU o DMA2, che necessita dell'arbitro del Bus Matrix

La figura illustra l'accesso concorrente al ponte AHB-APB1 generato sia dalla CPU attraverso il Bus Matrix che dal DMA1 tramite percorso diretto.

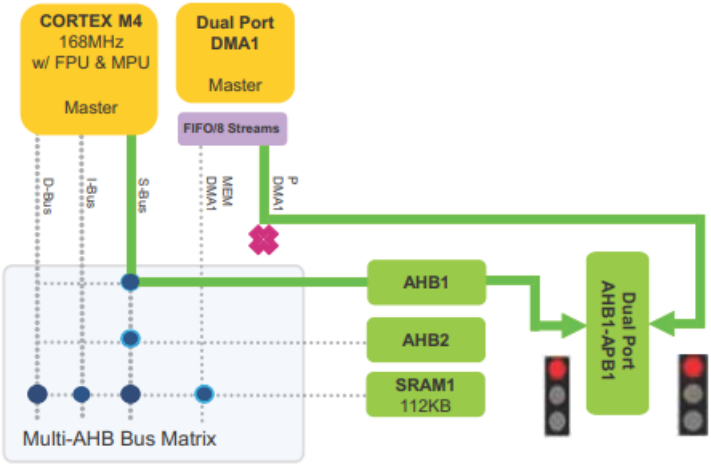


Figura 2.6.1.7 Ponte AHB-to-APB1 con richiesta di accesso da parte della CPU e del DMA1

2.6.2 MODALITA' E TIPI DI TRASFERIMENTO

Ci sono tre modalità di trasferimento per quanto riguarda l'origine e la destinazione dei dati:

1. Da periferica a memoria
2. Da memoria a periferica
3. Da memoria a memoria

La terza modalità è possibile con il DMA2, dove le modalità circolare e diretta non sono consentite. Per quanto riguarda i tipi di trasferimento, si intendono le tecniche con cui il DMA sposta i dati tra la sorgente e il buffer di destinazione. Ci sono 3 tipi:

1. Modalità Circolare : è disponibile per gestire buffer circolari, ovvero una struttura dati in cui gli elementi vengono inseriti e prelevati in modo circolare. Questo è utile

per gestire flussi continui di dati, permettendo un utilizzo ottimale della memoria senza interruzioni di flusso. Caratteristicamente, quando il buffer raggiunge la sua capacità massima, i nuovi dati sovrascrivono le posizioni iniziali, creando un ciclo continuo di aggiunta e prelievo dei dati.

2. Modalità Normale: i dati vengono trasferiti in modo unidirezionale e sequenziale.
3. Modalità Double Buffer : lo Stream lavora come un singolo buffer ma dispone di due puntatori lato memoria. Quando viene completata una transazione, i due puntatori vengono scambiati e un'altra transazione ha inizio, consentendo all'applicazione di elaborare i dati in un buffer mentre l'altro viene riempito o utilizzato dal controller DMA.

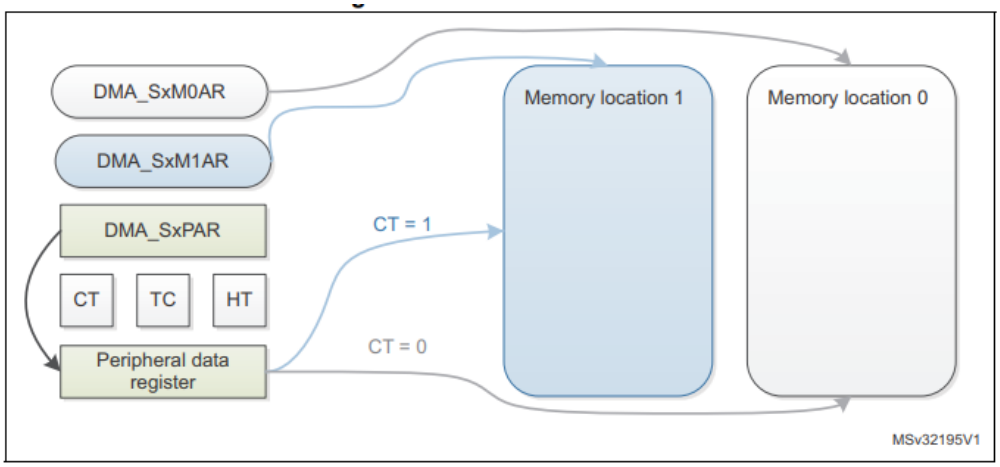


Figura 2.6.2.1 Schema della modalità Double-Buffer

Oltre ai tipi di trasferimento precedentemente visti, esistono altre due modalità che caratterizzano il trasferimento dei dati:

1. Modalità “Direct”: ogni singola richiesta proveniente dalle periferiche attiva un singolo trasferimento di dati alla/dalla memoria. In questa modalità è necessario che la dimensione della sorgente e della destinazione sia la stessa ed entrambi uguale al PSIZE, ovvero la dimensione della “Peripheral Port”.
2. Modalità “FIFO” : ovvero “First In First Out”, implica che gli elementi che prima vengono inseriti nel buffer siano anche i primi ad essere processati.

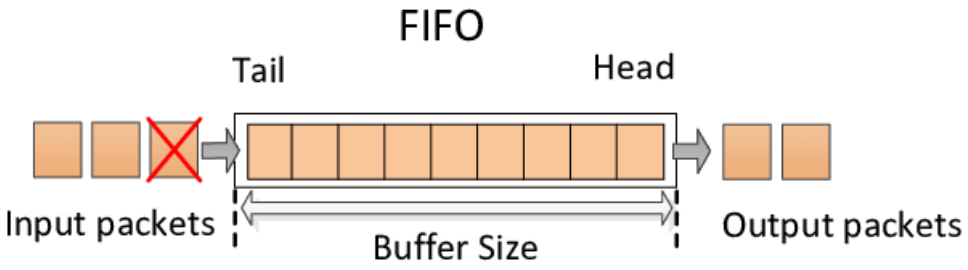


Figura 2.6.2.2 Illustrazione modalità FIFO in un buffer

Nella modalità FIFO, ciascuno Stream è dotato di una propria FIFO indipendente, che può contenere fino a 4 parole (ovvero 4 sequenze di 32 bit). È possibile configurare la soglia limite di questa FIFO tramite software, scegliendo tra ¼, ½, ¾ o piena capacità del buffer. La funzione principale di questa FIFO è quella di temporaneamente contenere i dati provenienti dalla sorgente prima di trasmetterli alla destinazione. L'utilizzo della modalità FIFO permette l'implementazione della modalità Burst. La soglia impostata per la FIFO del DMA definisce quando verrà generata la richiesta di trasferimento verso il "Memory Port" del DMA. In sintesi, la modalità FIFO e la configurazione della soglia della sua FIFO associata giocano un ruolo cruciale nell'ottimizzazione dei trasferimenti di dati, consentendo un controllo più preciso e una gestione efficiente del flusso di dati tra sorgente, buffer e destinazione attraverso il DMA. Il vantaggio di questa modalità è che dà più tempo ai Master di accedere al Matrix Bus senza concorrenza aggiuntiva.

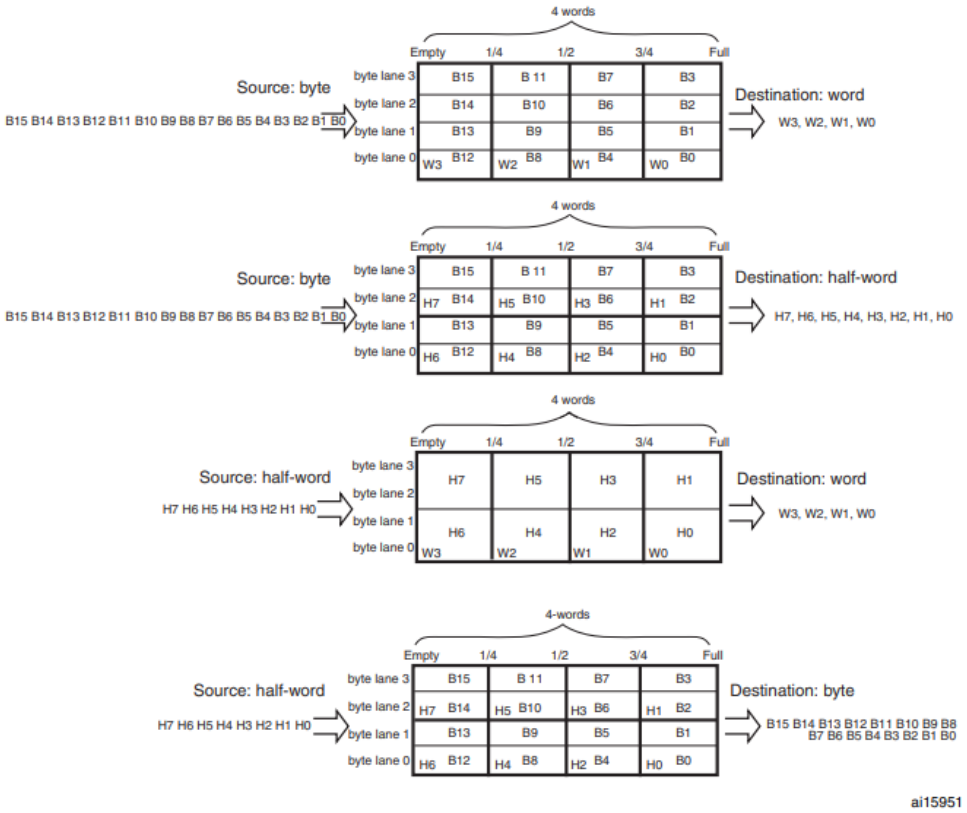


Figura 2.6.2.3 Struttura della modalità FIFO con diverse dimensioni per l'origine e la destinazione

Una caratteristica aggiuntiva della modalità FIFO è la sua capacità di essere utilizzato insieme al "Burst Transfer", che rappresenta una sequenza di trasferimenti di dati consecutivi eseguiti senza interruzioni o latenze significative. Quando si avvia un'operazione DMA, i dati vengono trasferiti in una sequenza di "burst", ognuno dei quali rappresenta un gruppo di trasferimenti di dati in rapida successione senza interruzioni, al fine di ottimizzare l'efficienza del trasferimento. La dimensione del Burst sulla porta periferica del DMA va impostata in accordo alla capacità della periferica e deve coincidere con il livello di soglia del FIFO, ciò permette allo stream di DMA di avere abbastanza dati nel FIFO quando è iniziato il trasferimento burst nella memoria. Il DMA opera in modo tale che, una volta avviato il trasferimento, mantenga il controllo del bus principale, limitando l'accesso da parte della CPU fino al completamento dell'operazione. Questa modalità richiede che sia la periferica di origine che la memoria supportino trasferimenti continui ad alta velocità per il tempo necessario al DMA Controller.

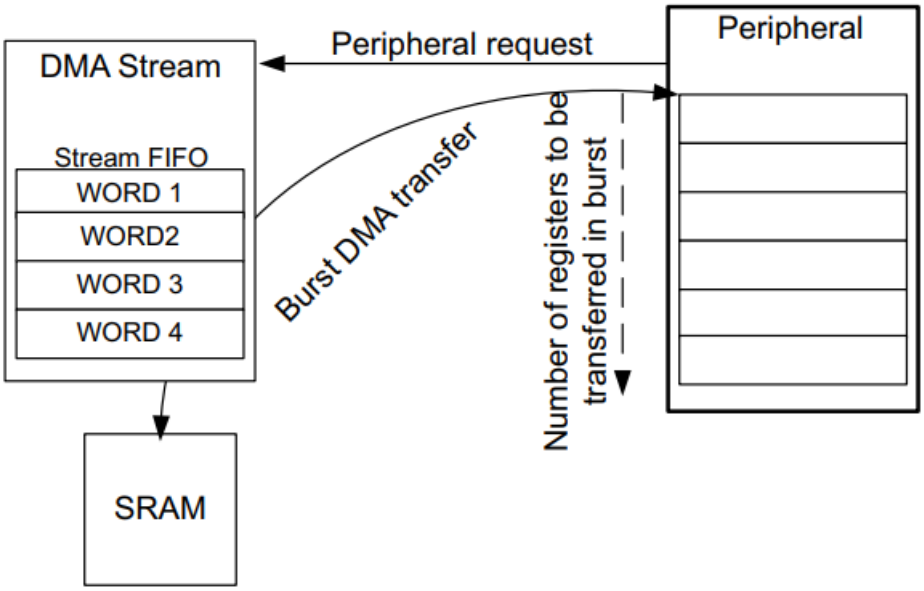


Figura 2.6.2.4 Schema della Modalità FIFO combinata con la modalità Burst.

2.7 STRUMENTI DI SVILUPPO FIRMWARE E ANALISI

Il presente capitolo discute gli strumenti utilizzati nella presente ricerca. Sono stati utilizzati una varietà di strumenti, tra cui CUBE IDE, un ambiente di sviluppo integrato. È un'avanzata piattaforma di sviluppo in C/C++ che permette la configurazione delle periferiche, generazione del codice e possibilità di debugging da utilizzare con i microcontrollori STM32.

Tra le varie funzionalità di Cube IDE, c'è la possibilità di andare a configurare il sistema di clock del microcontrollore, che nella nostra scheda può essere settato fino ad un massimo di 84MHz. Ciò andrà ad influire sulla temporizzazione e sulle prestazioni complessive del progetto.

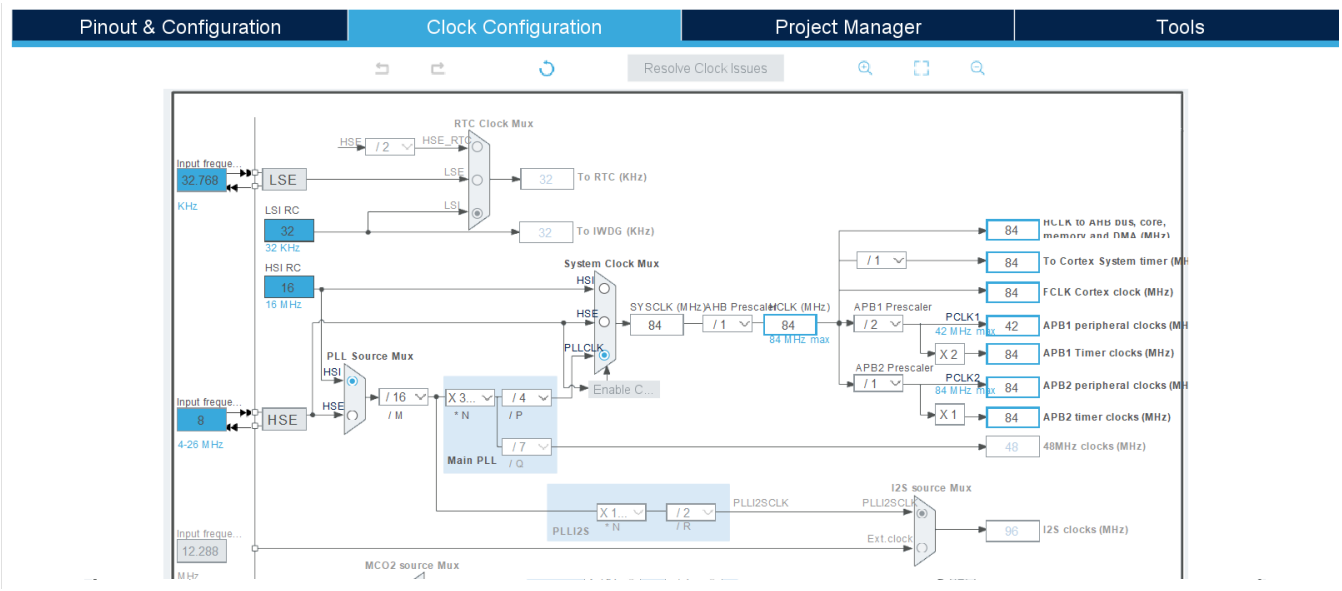


Figura 2.7.1 Schermata Clock Configuration di Cube IDE

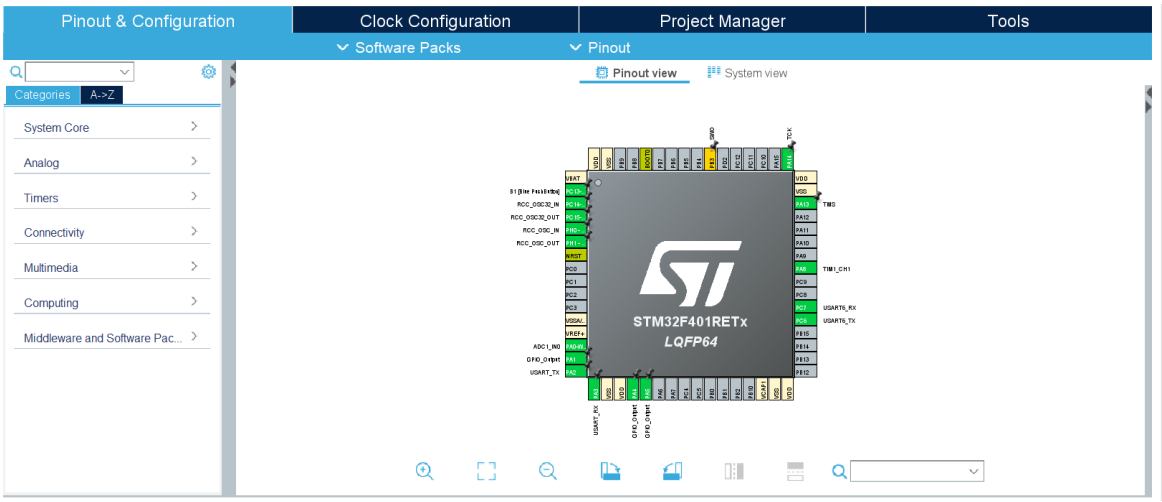


Figura 2.7.2 Schema Pinout & Configuration di Cube IDE

È possibile assegnare, tramite la schermata di "Pinout & Configuration", i pin di ingresso/uscita sul microcontrollore, consentendo di mappare le funzionalità desiderate ai pin fisici in base alle nostre esigenze. L'efficienza distintiva di Cube IDE risiede nella sua capacità di consentire la programmazione simultanea sia dell'hardware che del firmware. Questa caratteristica unica permette di ottimizzare il processo di sviluppo incorporato, consentendo agli sviluppatori di lavorare in parallelo sulla configurazione hardware e sulla scrittura del firmware.

Per la visualizzazione dei dati attraverso l'UART, è opportuno utilizzare un software in grado di leggere dati dalla connessione seriale e generarne un grafico in tempo reale, come ad esempio MATLAB.

Uno strumento di grande utilità è l'AZDelivery Logic Analyzer, un analizzatore logico a 8 canali. È possibile collegarlo al PC tramite porta USB, mentre i suoi 8 ingressi saranno impiegati per registrare segnali, visualizzabili attraverso il software Saleae Logic. L'analizzatore è in grado di rilevare variazioni di livello logico da basso ad alto.

Figura 2.7.3 Logic Analyzer Az-Delivery.

3. SVILUPPO DEL PROGETTO

3.1 PRESENTAZIONE PROGETTO

Lo scopo principale di questa tesi è sviluppare un sistema per l'elaborazione in tempo reale di dati audio mediante l'impiego efficiente del DMA (Direct Memory Access).

L’ obiettivo è di eseguire campionamenti ad alta velocità attraverso il convertitore analogico-digitale, al fine di riprodurre con precisione il segnale audio in ingresso. È stato fondamentale trovare un equilibrio tra la velocità di campionamento e le successive operazioni di elaborazione sui dati acquisiti. Programmando le conversioni dell'ADC in intervalli predefiniti, l'acquisizione dei campioni dalla periferica ADC alla memoria viene gestita tramite il DMA, il che accelera tali operazioni e solleva la CPU dal compito, consentendole di concentrarsi sull'elaborazione dei dati.

Se avessimo scelto di elaborare i dati uno per uno, ci saremmo dovuti confrontare con il noto problema del "jitter", che rappresenta variazioni imprevedibili nell'intervallo di acquisizione dei dati. Questo fenomeno sorge dal fatto che il tempo richiesto per eseguire l'elaborazione dopo ogni singola acquisizione avrebbe impedito di avviare le successive conversioni nel preciso istante definito dal timer. Pertanto, è stato adottato un approccio differente: i dati vengono accumulati in un buffer e saranno elaborati solo quando il buffer sarà riempito con un numero appropriato di campioni.
Un’ulteriore strategia chiave per ottimizzare l'efficienza operativa consiste nell'adozione della tecnica denominata 'Ping-Pong' per l'elaborazione dei dati nel buffer. Questa tattica sfrutta il fatto che, mentre il DMA sta popolando, ad esempio, la seconda metà del buffer, la CPU ha la libertà di elaborare la prima metà. Tramite questa metodologia, si riesce a sfruttare al massimo le risorse disponibili, assicurando un'elaborazione dei dati acquisiti più rapida ed efficiente.

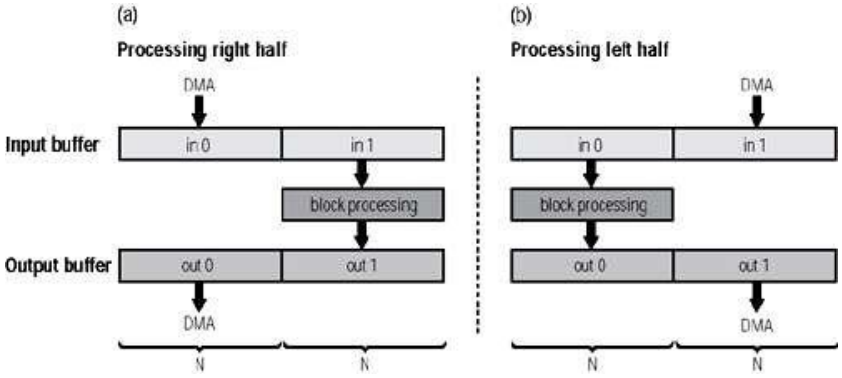


Figura 3.1.1 Schema della tecnica di elaborazione "Ping-Pong"..

3.2 PROGETTAZIONE IN CUBE IDE PER L'ELABORAZIONE DEI DATI AUDIO

Per questo progetto è stato utilizzato un componente Microfono Iot AZDelivery GY-MAX4466, con guadagno regolabile e un ottimo rapporto di reiezione di modo comune in ambienti rumorosi. Lo scopo di questo componente è convertire le onde sonore in segnali elettrici e successivamente in tensione analogica, che andremo a porre in ingresso all'ADC. Come rappresentato in Figura3.2.2, il pin GND del microfono è collegato alla massa della scheda, il pin VCC alla tensione di riferimento (nel nostro caso 3.3V) e il pin OUT all'ingresso dell'ADC (PA0).

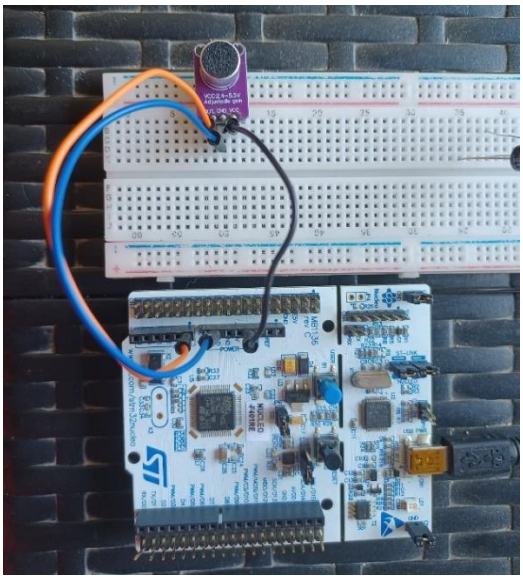


Figura 3.2.1 Microfono utilizzato *Figura 3.2.2 Assemblaggio fisico della scheda e del microfono*

Nel contesto del progetto Cube IDE, la prima operazione da compiere è la modifica delle impostazioni del 'RCC Mode and Configuration', come illustrato nella Figura 3.2.3.

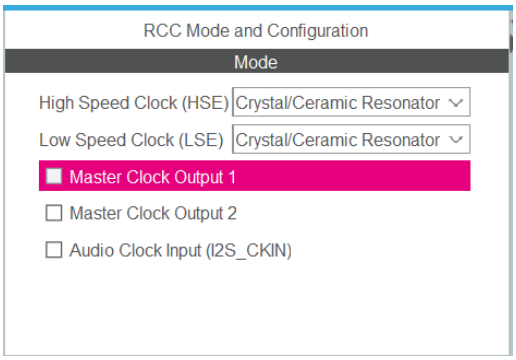


Figura 3.2.3 "RCC Mode and Configuration"

L'ADC opererà con una risoluzione di 12 bits, generando così 4096 livelli di quantizzazione. È essenziale abilitare il "DMA Continuos Requests" e "Continuos Conversion Mode" , come mostrato in Figura 3.2.4.

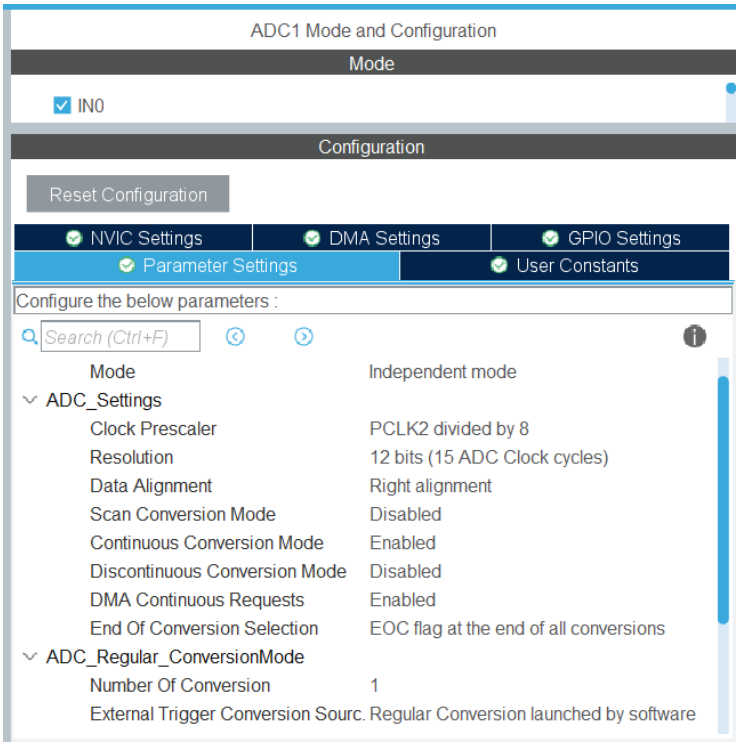


Figura 3.2.4 Impostazioni ADC

In "DMA Settings" dell'ADC è necessario abilitare il "DMA2 Stream0" tale che indirizzi i dati dalla periferica alla memoria, con una priorità bassa e in modalità Circolare, con la dimensione dei dati da inviare di "Half-Word".

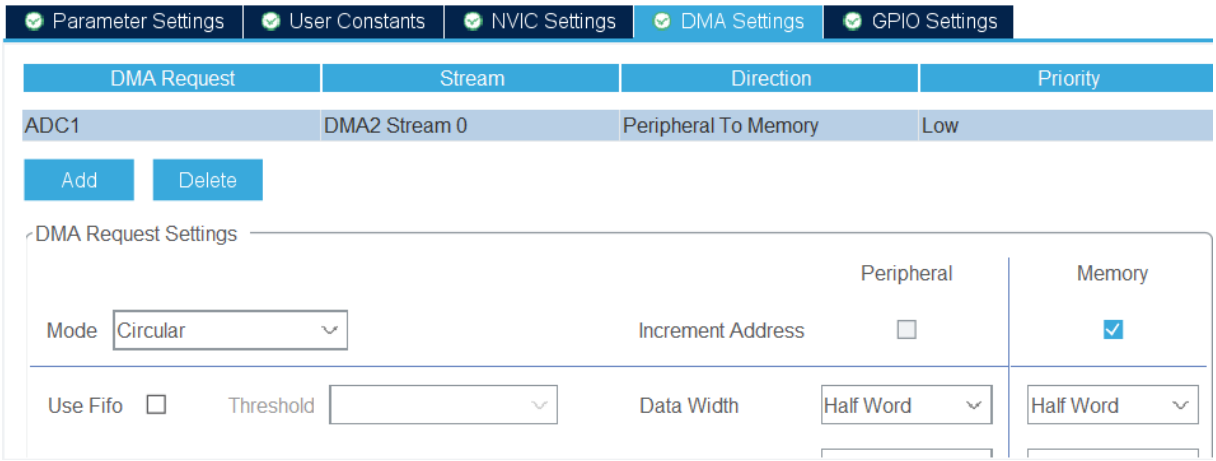


Figura 3.2.5 "DMA Settings" dell'ADC

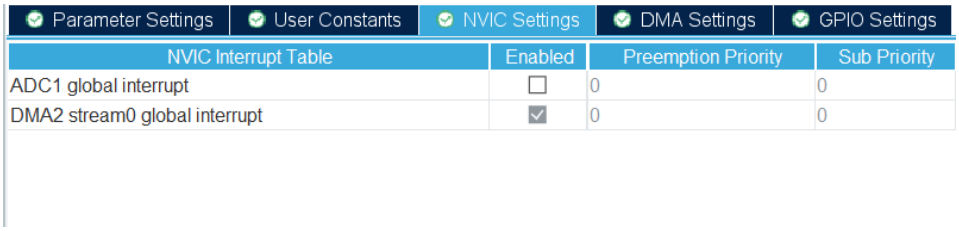


Figura 3.2.6 "NVIC Settings" dell'ADC

Per quanto riguarda la temporizzazione delle conversioni, si è scelto di agire direttamente sul clock dell'ADC. È possibile modificare il tempo di conversione dell'ADC in base alle esigenze di progetto intervenendo sulle impostazioni di 'APB2 peripheral clocks' nel Clock Tree, insieme al 'Clock Prescaler' e al 'Sampling Time'. Il periodo di campionamento, che è di circa 754 microsecondi, è calcolato utilizzando la seguente formula, considerando una risoluzione di 12 bit, equivalente a 15 cicli di clock. $T = \frac{Prescaler}{APB2\ peripheral\ Clock} * (Resolution + Sampling\ Time) =$

$$= \frac{8}{5.25M} * (15 + 480\ ) = 754\mu s$$

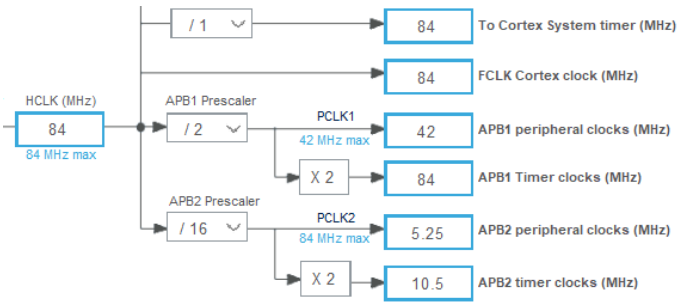


Figura 3.2.7 "APB2 peripheral clocks" del clock Tree.

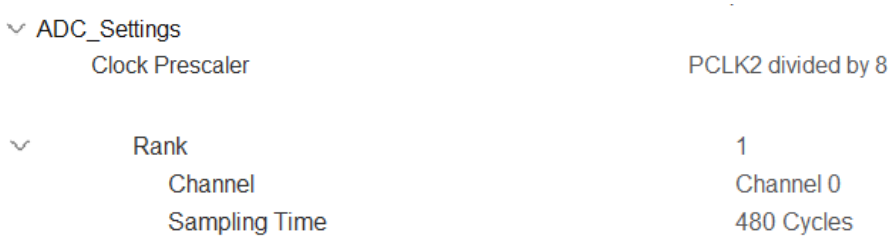


Figura 3.2.8 Impostazioni "Clock Prescaler" e "Sampling Time" dell'ADC.

Questa scelta è stata determinata dalla necessità di parallelizzare le operazioni di conversione e di elaborazione dei dati, garantendo che il periodo di conversione sia uguale o superiore al tempo richiesto per l'elaborazione e l'invio dei singoli dati.

Per confermare i tempi di conversione ed elaborazione, è stato implementato alcune verifiche all'interno delle nostre funzioni di callback. In particolare, nella funzione HAL_ADC_ConvCpltCallback, che viene attivata quando il buffer di conversione è pieno, configuriamo il pin analogico PA1 in modo che oscilli tra i due livelli di tensione logica ogni volta che questa funzione viene eseguita. Successivamente, per garantire che il tempo necessario per elaborare e trasmettere i 200 valori non superi il tempo richiesto per acquisirli dall'ADC, così da evitare sovrapposizioni di operazioni, si è introdotto una verifica aggiuntiva. Questa verifica si basa sul pin analogico PA6, il cui stato logico viene modificato all'interno della funzione HAL_UART_TxCpltCallback, richiamata alla fine di ogni trasmissione. La Figura 3.2.9 mostra chiaramente queste dinamiche nel Canale 0, dove sono visualizzate le 200 trasmissioni suddivise in due blocchi. Il primo blocco inizia in concomitanza con l'avvio della conversione, mentre il secondo blocco ha inizio quando la prima metà delle conversioni è stata completata. Nel Canale 4, invece, è possibile monitorare la durata complessiva della conversione del buffer

Figura 3.2.9 Software Logic Analyzer . "Channel 0" analisi temporale dell'elaborazione e trasmissione dei dati, "Channel 4" analisi temporale della conversione di 200 elementi.

Infine, sarà necessario abilitare il protocollo di comunicazione USART2. Anche in questo caso verrà abilitato il DMA, tale che i dati, di dimensione di 1 Byte, siano diretti dalla memoria alla periferica, con una priorità bassa.

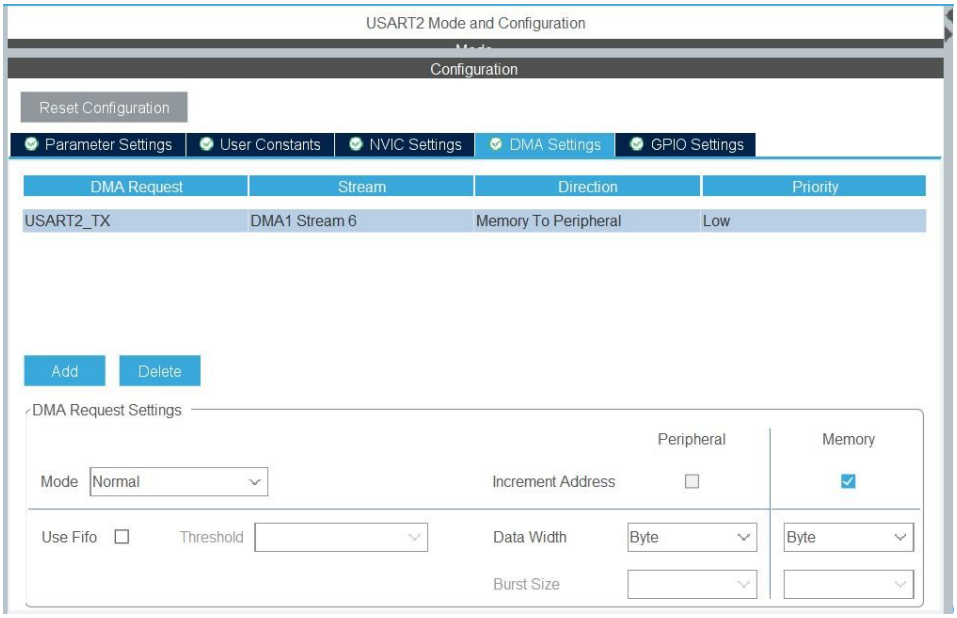


Figura 3.2.10 "DMA Settings" di USART2.

Per implementare l'elaborazione dei dati tramite tecnica "Ping-Pong" sono state utilizzate le due funzioni nel seguente codice, che vengono richiamate rispettivamente quando il buffer dell'ADC sarà riempito per metà della sua lunghezza e quando avrà completato tutte le conversioni. La variabile 'flag' sarà impiegata per notificare nel While(1) che la prima metà del buffer è pronta per l'elaborazione, mentre la seconda

metà è in fase di acquisizione. Invece, la variabile 'flag1' segnalerà che la seconda metà è stata elaborata, consentendo al buffer di essere riempito ripartendo dalle posizioni iniziali.

```
void HAL_ADC_ConvHalfCpltCallback (ADC_HandleTypeDef* hadc1){
      flag=1;
}
void HAL_ADC_ConvCpltCallback (ADC_HandleTypeDef* hadc1){
        flag1=1;
}
```

Al fine di garantire una trasmissione dei dati senza sovrapposizioni, è stata implementata una funzione di callback addizionale per il trasferimento dei dati attraverso UART. Questa funzione viene richiamata al termine della trasmissione di un singolo valore, assicurando una gestione efficace e sicura dei dati in arrivo e in uscita. La variabile ‘myflag’ segnalerà se la trasmissione precedente è stata completata, consentendo l’esecuzione della successiva.

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart2)
{     myflag=1;


}
```

Si può osservare il codice implementato all'interno del ciclo principale (While(1)). Questo codice è progettato per gestire l'elaborazione e la trasmissione dei dati tramite la connessione UART. I dati da trasmettere sono i valori acquisiti dall'ADC. Questo processo di elaborazione e trasmissione avviene in due fasi distinte. Le due condizioni if servono a separare la gestione delle due metà del buffer in modo indipendente. Ad esempio, quando la variabile flag viene attivata, l'ADC inizia l'acquisizione della seconda metà del buffer, mentre la CPU richiama il corrispondente blocco if ed entra in un ciclo for con una lunghezza pari alla metà del buffer. Durante questo ciclo, inizia l'elaborazione dei singoli valori. I dati in uscita dall'ADC vengono convertiti in valore di tipo float e vengono riportati ai loro valori di tensione originali moltiplicandoli per la tensione di riferimento di 3.3 V e dividendo il risultato per 4096, che rappresenta i livelli di quantizzazione dell'ADC. Dopo aver elaborato un valore, il codice attende finché la variabile myflag non diventa 1, il che indica che la trasmissione precedente è stata completata con successo. A questo punto, il codice trasmette i 4 byte che compongono un singolo valore di tipo float, insieme a un ulteriore byte separatore. Una volta che tutti i valori sono stati inviati, la variabile flag viene riportata a 0, pronta

per essere richiamata nuovamente quando l'ADC inizia una nuova acquisizione. Questo ciclo di acquisizione, elaborazione e trasmissione continua in modo sincronizzato e regolato, garantendo che i dati siano trasmessi in modo coerente attraverso la connessione UART.

```
 while (1)
  {
        if(flag == 1){
              flag=0;
                    for(int i =0; i<len/2; i++){
             data[0] = (float)buffer[i] * 3.3f / 4096.0f;
                                 while (myflag == 0){;}
                                        if(myflag==1){
                                             if
(HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK)
                                                  {

                                                    Error_Handler();
                                                  }


                                             myflag = 0;
                                        }}

              }
               if(flag1==1){
                    flag1=0;
                   for(int i =len/2; i<len; i++){
                   data[0] = (float)buffer[i] * 3.3f / 4096.0f;

                                             while (myflag == 0){;}
                                                    if(myflag==1){
                                                         if
(HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK)
                                                              {

Error_Handler();

                                                              }

                                                        myflag = 0;
                                                   }}

              }
```

Per ricevere in tempo reale i dati, è stato utilizzato MATLAB, come si vede nel seguente codice acquisiamo i valori e li conserviamo nell'oggetto 'serialport'. Contestualmente, eseguiamo un ciclo 'While' per leggere i dati, utilizzando la funzione 'readline' ad ogni iterazione. I 5 byte ricevuti, venogno, poi, convertiti in un numero a virgola mobile di tipo 'single' e visualizzato in un grafico.

```
s = serialport('COM4', 115200);

count = 0;
dati=[];
format long
figure
h1 = animatedline;
xlabel('Time [s]'),
ylabel('Voltage[V]');
sendCount = 0;

while (1)
    count = count + 1;

    try
        Afloat = typecast(uint8(convertStringsToChars(readline(s))), 'single');
    catch
        warning('Problem using function.  Assigning a value of 0.');
        Afloat = [0 0];
    end

    if count > 200
        xlim([(count - 200), count]);
    end

    x = double(Afloat(1));
    addpoints(h1, count, x);
    drawnow

    dati(count)=x;
end
```

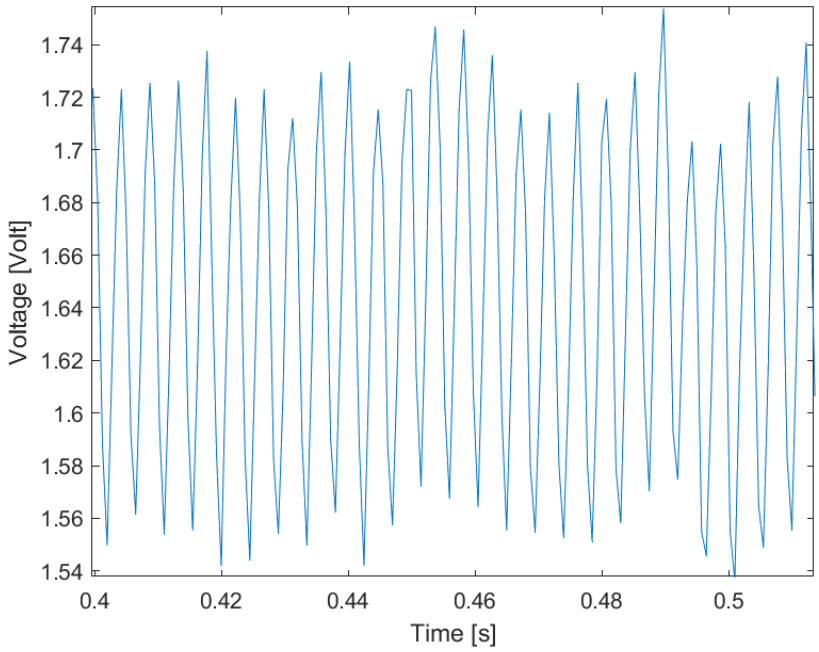


Figura 3.2.11 Rappresentazione grafica su Matlab del segnale audio a 226 Hz in ingresso al microfono.

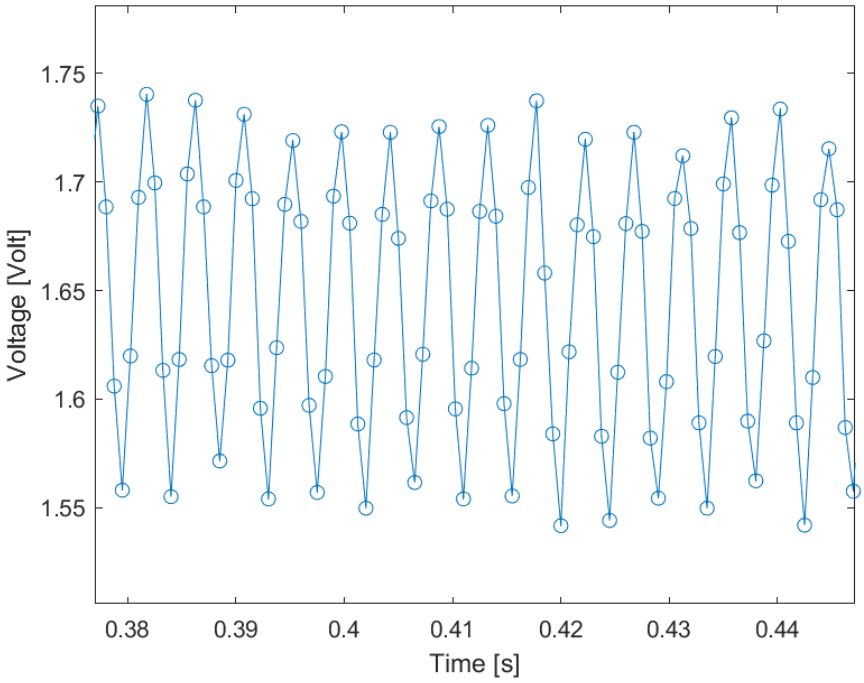


Figura 3.2.12 Rappresentazione grafica del segnale audio a 226 Hz in ingresso al microfono, con i 'marker' che indicano i punti campionati.

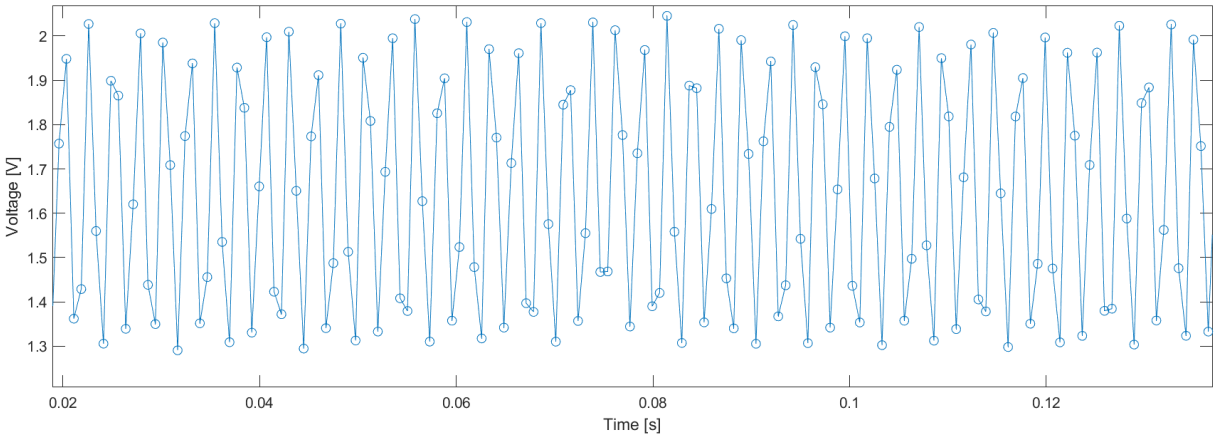


Figura 3.2.17 Rappresentazione grafica su Matlab del segnale audio a 400 Hz in ingresso al microfono, con i 'marker' che indicano i punti campionati.

È possibile grazie a Matlab fare successivamente elaborazioni sui dati ricevuti in ingresso, come ad esempio la FFT.

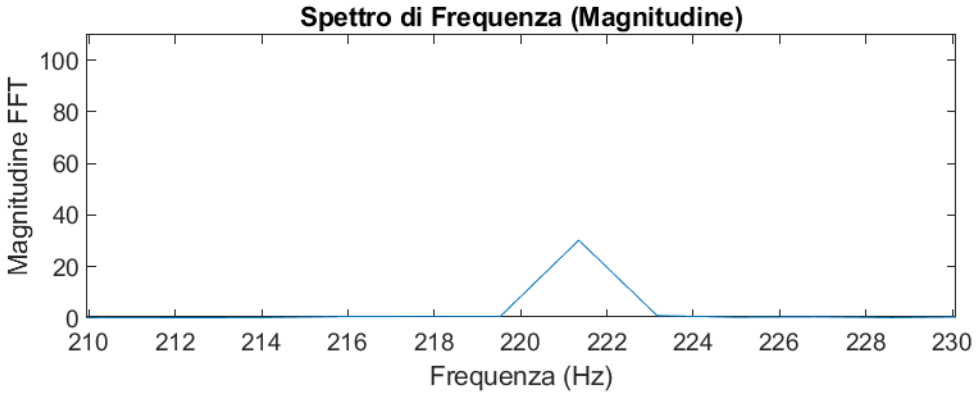


Figura 3.2.13 Magnitudo della FFT del segnale audio a 226 Hz.

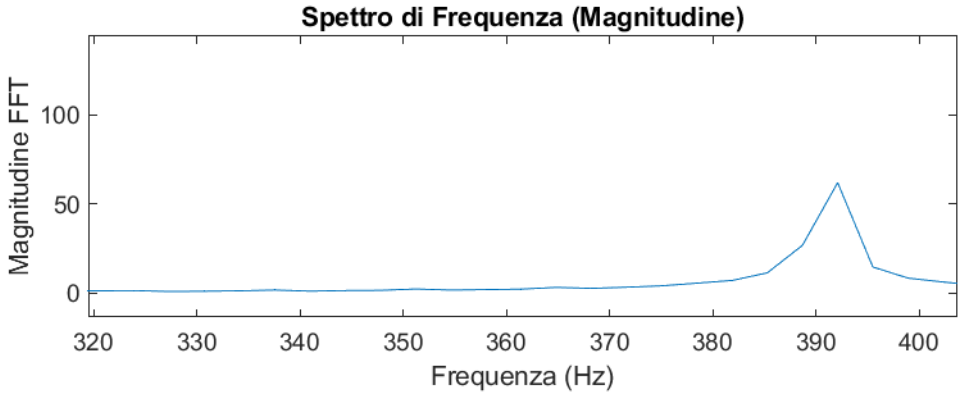


Figura 3.2.14 Magnitudo della FFT del segnale a 400 Hz.

Una scelta di progetto che è stata successivamente abbandonata a causa di risultati insoddisfacenti riguarda l'uso di un trigger da parte di un timer per temporizzare le operazioni dell'ADC. In Figura 3.2.15 è possibile vedere le impostazioni del Timer 3 usato come trigger per il convertitore analogico-digitale.

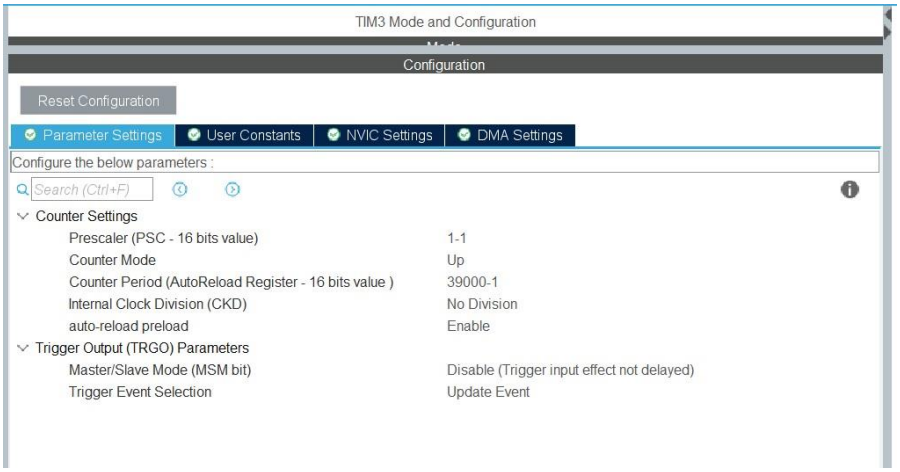


Figura 3.2.15 Impostazioni TIM3

I valori dell'ARR(Auto Reload Register) e CP (Counter Period) di TIM3 sono stati scelti tramite la seguente formula.

$$T_{TIM} = \frac{(ARR+1)(CP+1)}{Fclock} = \frac{(39000)(1)}{84M} \approx 460\mu s$$

È necessario in "NVIC Settings" abilitare l'interrupt generato dal timer.

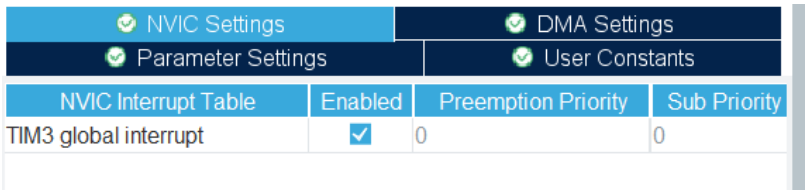


Figura 3.2.16 NVIC Setting di TIM3

Per l'ADC le impostazioni si differenziano solo per porre il TIM3 come tirgger esterno.

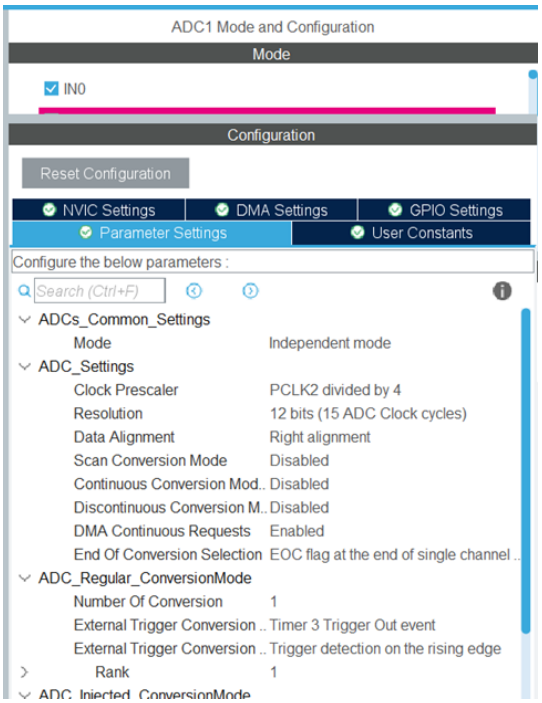


Figura 3.2.17 Impostazioni ADC.

Le impostazioni restanti e il codice rimangono invariati. Tuttavia, come mostrato nella Figura 3.2.18, è evidente che il grafico presenta punti in cui il valore di tensione sembra rimanere costante, ripetendo il campione precedente, dato da un probabile errore di sincronizzazione delle operazioni.

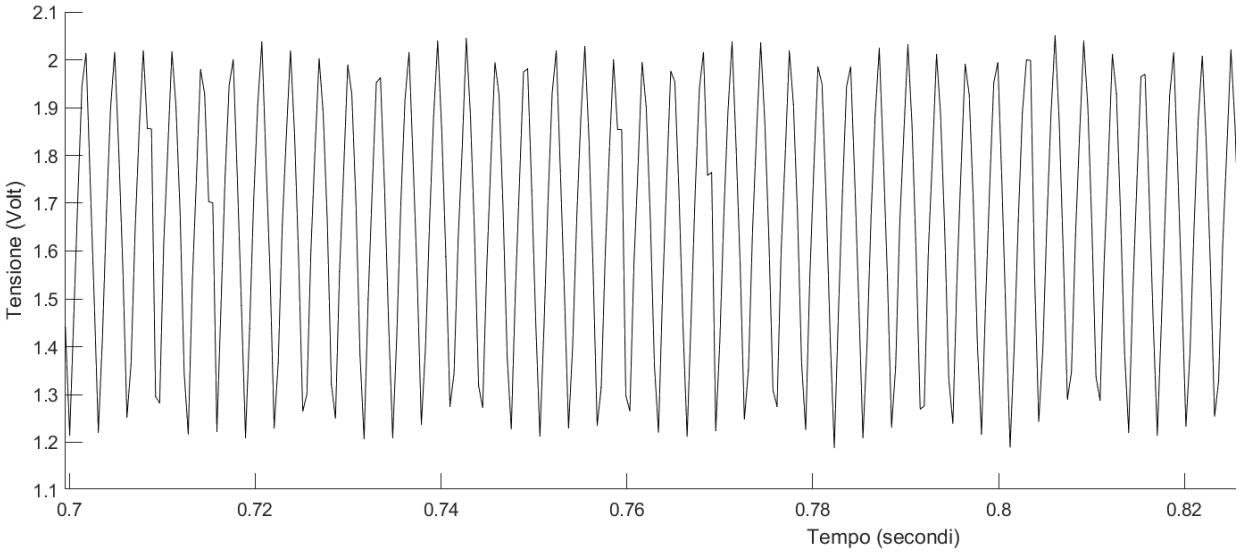


Figura 3.2.18 Rappresentazione grafica del segnale audio tramite microfono utilizzando il timer per temporizzare le operazioni.

3.3 PROGETTAZIONE IN CUBE IDE PER L'ELABORAZIONE DATI TRAMITE L'UTILIZZO DI UN BOTTONE

Per convalidare l'efficacia della tecnica "Ping-Pong" nell'elaborazione dei dati, è stato sviluppato un codice preliminare. Nell'implementazione dell'ADC, è stata configurata una modalità in cui vengono eseguite 200 conversioni ad intervalli prestabiliti ogni volta che viene premuto un pulsante. L'elaborazione da valori analogici a digitali viene effettuata mediante la tecnica "Ping-Pong", e i dati risultanti sono inviati a un terminale dopo il completamento di tutte le conversioni. Questa strutturazione del codice ci consente di utilizzare velocità di campionamento più elevate, poiché non si verifica più sovrapposizione tra le conversioni e l'invio dei dati tramite UART. Questo approccio è stato adottato per mettere alla prova la robustezza del codice, al fine di valutare come il sistema gestisce l'acquisizione e l'elaborazione dei dati. Tale test svolge un ruolo cruciale nel rilevare eventuali problematiche nella gestione dei buffer o comportamenti anomali. Nel contesto del progetto Cube IDE, le impostazioni di USART rimangono invariate, mentre è necessario apportare delle modifiche alla configurazione dell'ADC. In particolare, è essenziale configurare l'ADC in modo che si interrompa automaticamente una volta che il buffer ha raggiunto la sua dimensione. Per raggiungere questo obiettivo, nella sezione 'DMA Settings', è necessario abilitare la modalità Normale.

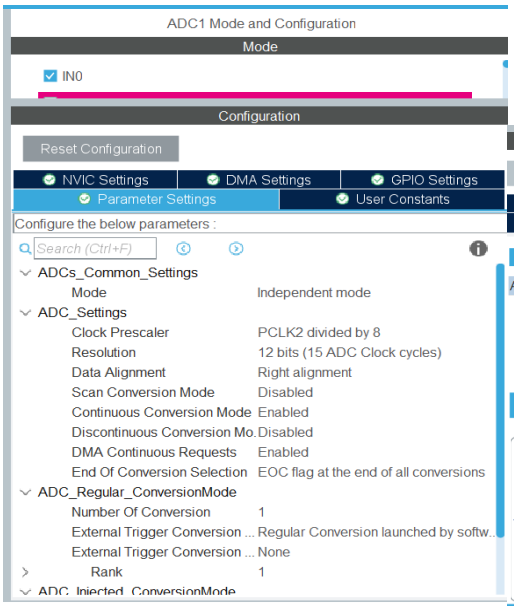


Figura 3.3.1 Impostazioni ADC.

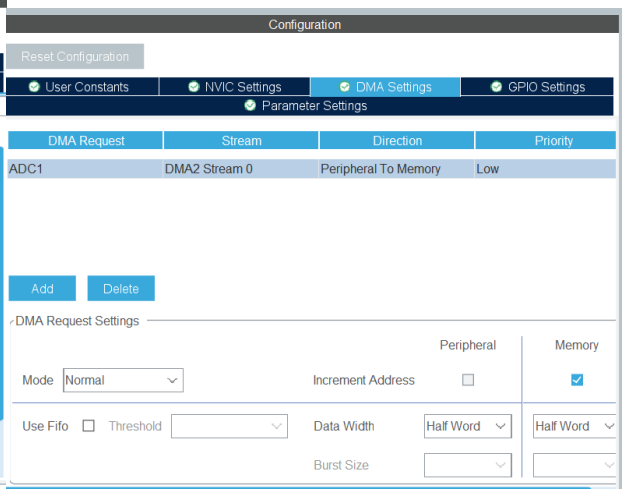


Figura 3.3.2 "DMA Setting" dell'ADC.

Come si è visto precedentemente, modificando "APB2 Peripheral Clock" , "Clock Prescaler" e "Sampling Time" possiamo variare il periodo di conversione dell'ADC.

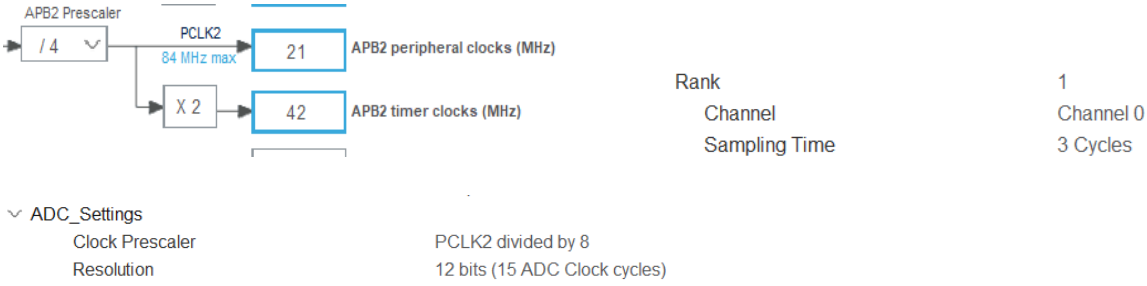


Figura 3.3.3 "APB2 Peripheral Clock" , "Clock Prescaler" e "Sampling Time" dell'ADC.

Sostituendo i seguenti valori della Figura 3.3.3 nella la seguente formula si ottiene un periodo di campionamento di circa 6.8 microsecondi.

$$T = \frac{Prescaler}{APB2\ peripheral\ Clock} * (Resolution + Sampling\ Time) =$$

$$= \frac{8}{21M} * (15 + 3) \cong 6.8\mu s$$

Sulla Pinout View in Figura 3.3.4 sarà possibile osservare l'ingresso dell'ADC, ADC_IN0 ovvero PA0 e il tasto PC13 (Blue PushButton).

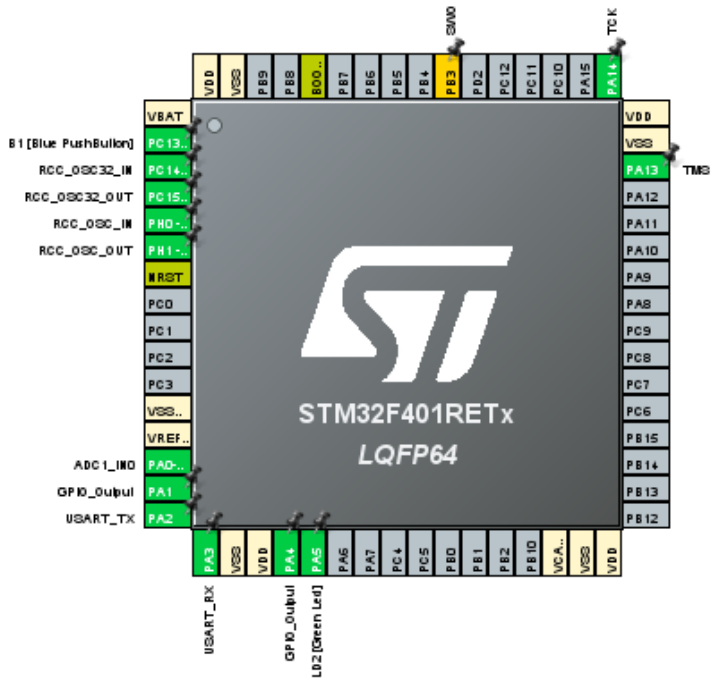


Figura 3.3.4 PinOut View.

Come si vede l'implementazione del seguente codice prevede che ogni volta che il bottone viene premuto sulla scheda, ovvero il pin corrispondente PC13 passa ad un livello logico basso, allora l'ADC inizierà le conversioni.

```
if( HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET){

      HAL_ADC_Start_DMA(&hadc1,(uint32_t*) buffer,len);
                                    }
```

L'elaborazione dei dati proveniente dall'ADC è la stessa spiegata nel progetto precedente, con la differenza che i dati vengono trasmessi solo una volta completate tutte le elaborazioni, come è possibile nel codice sottostante.

```
        if(flag == 1){
                    flag=0;
                          for(int i =0; i<len/2; i++){
                    buffer2[i] = (float)buffer[i] * 3.3f / 4096.0f;

                    }}

      if(flag1==1){
                           flag1=0;
                         for(int i =len/2; i<len; i++){
                         buffer2[i] = (float)buffer[i] * 3.3f / 4096.0f;

                         }
      for(int i =0; i<len; i++){
                              data[0] = buffer2[i];

                                      while (myflag == 0){;}
                                       if(myflag==1){
```

```
if (HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK)
                                                        {

                                                          Error_Handler();
                                                        }


                                                     myflag = 0;
                                                  }}
                        }
```

Nel contesto di MATLAB, è possibile rappresentare graficamente i risultati ottenuti. Si è collegato il nostro convertitore analogico-digitale a un generatore di segnali, con il quale sono stati prodotti segnali sinusoidali ad alte frequenze. Contestualmente, è stato collegato il convertitore a un oscilloscopio per l'acquisizione e la visualizzazione dei dati.

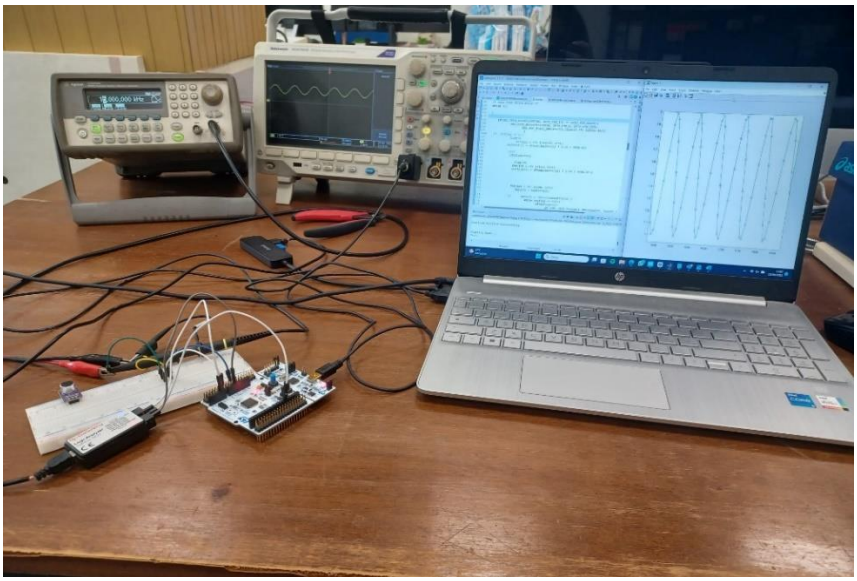


Figura 3.3.5 Generatore di segnale collegato all'oscilloscopio e l'ADC della scheda.

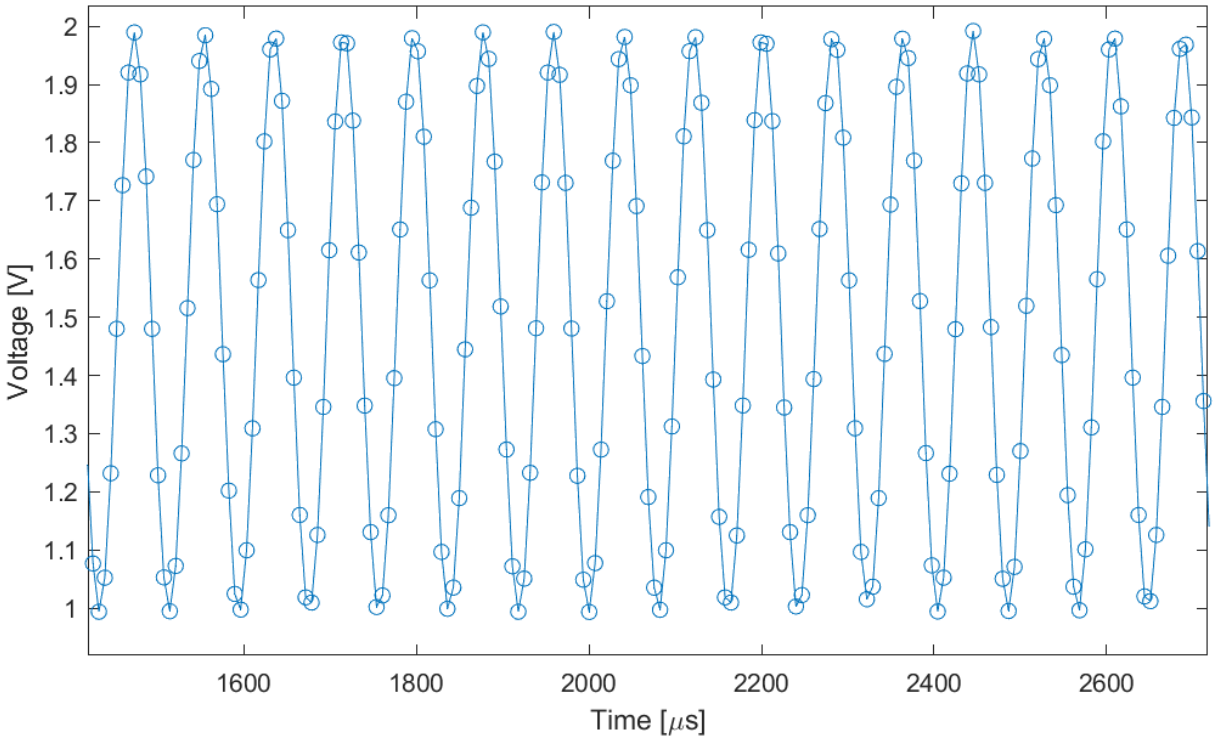


Figura 3.3.6 Rappresentazione grafica di un segnale a 13kHz, con marcatori che indicano i punti campionati.

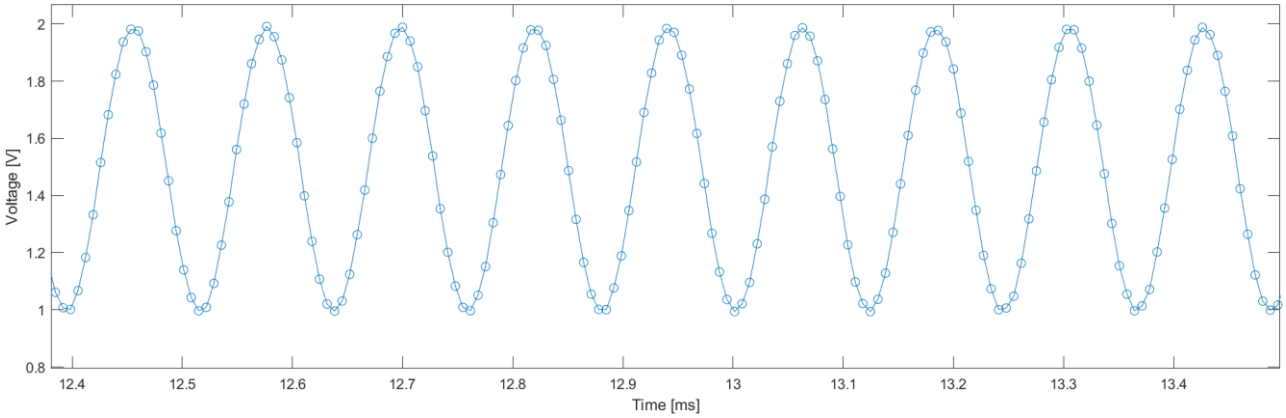


Figura 3.3.7 : Rappresentazione grafica di un segnale a 8kHz, con marcatori che indicano i punti campionati.

Successivamente, è stato collegato l'ADC al microfono e acquisito un campione di 2000 elementi audio di una voce per poi elaborarla tramite MATLAB.

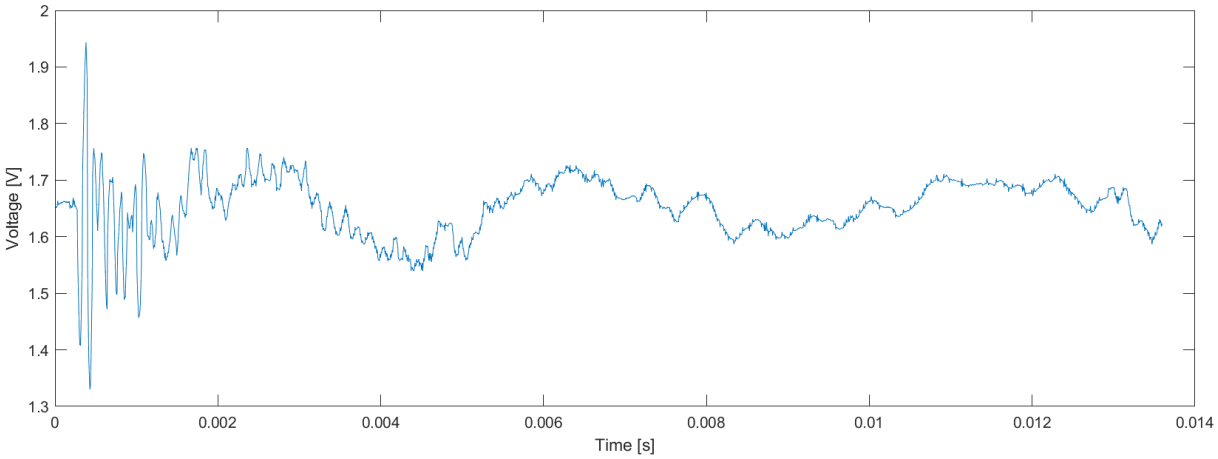


Figura 3.3.8 Rappresentazione grafica di un segnale audio vocale.

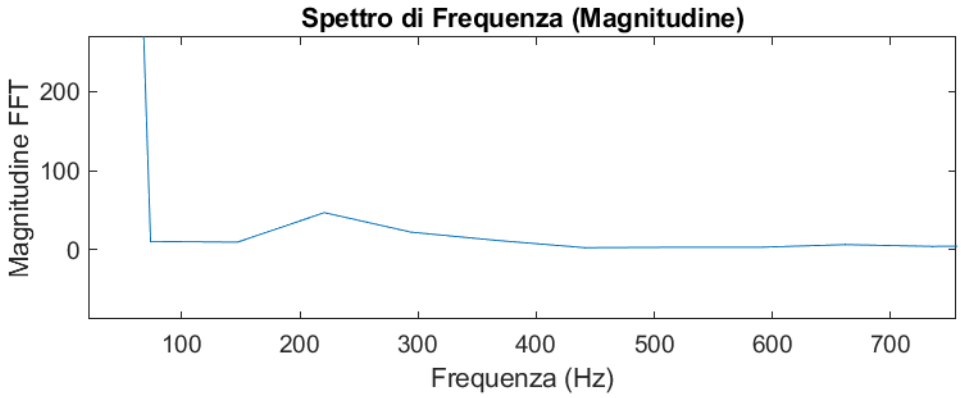


Figura 3.3.9 Magnitudo della FFT del segnale audio vocale.

Nelle stesse condizioni precedenti, è stato variato solo il “Sampling Time” a 480 cicli, per ottenere un tempo di campionamento maggiore di 188 microsecondi, di cui è stato preso un campione di 10mila elementi di un segnale audio vocale e la rispettiva FFT elaborata tramite MATLAB.

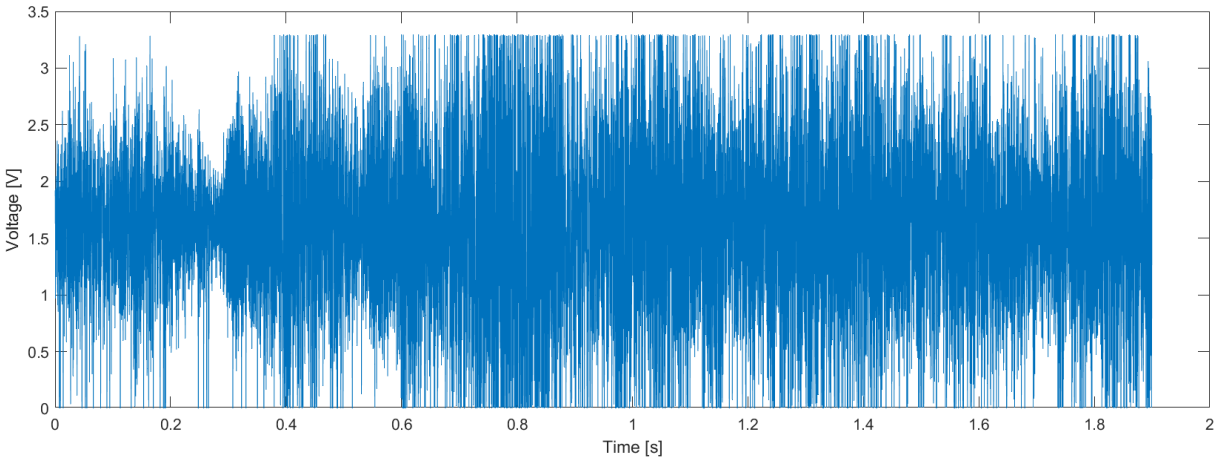


Figura 3.3.10 Rappresentazione grafica di un segnale audio vocale.

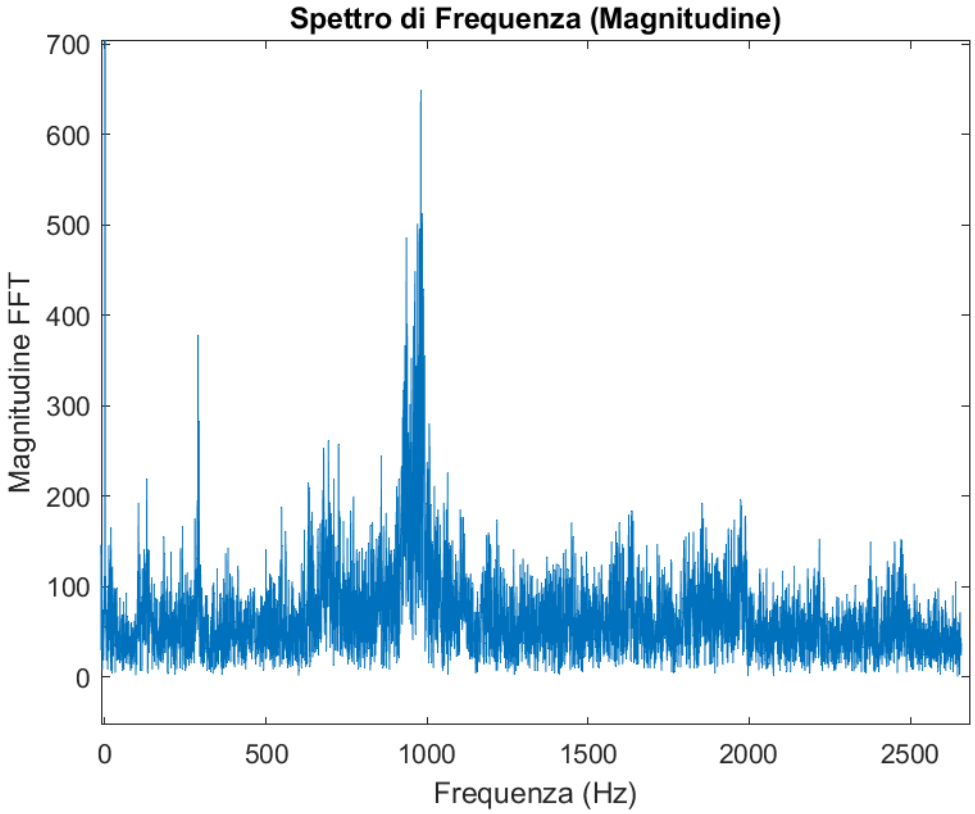


Figura 3.3.11 Magnitudo della FFT del segnale audio vocale.

Per lavorare ad alte velocità di campionamento, sono state condotte varie prove per adattare il periodo di campionamento dell'ADC, agendo sui parametri "APB2 Peripheral Clock" e "Sampling Time". Tuttavia, si nota che quando è stato cercato di aumentare notevolmente il "Sampling Time" per raggiungere velocità di campionamento più elevate, la qualità del grafico in uscita è peggiorata notevolmente come si vede in una

prova in Figura 3.3.11. Ponendo “APB2 Peripheral Clock” a 42MHz, e “Sampling Time” a 56 cicli, si ottiene un tempo di conversione di circa 13microsecondi.

$$T = \frac{Prescaler}{APB2\ peripheral\ Clock} * (Resolution + Sampling\ Time) =$$

$$= \frac{8}{42M} * (15 + 56) \cong 13\mu s$$

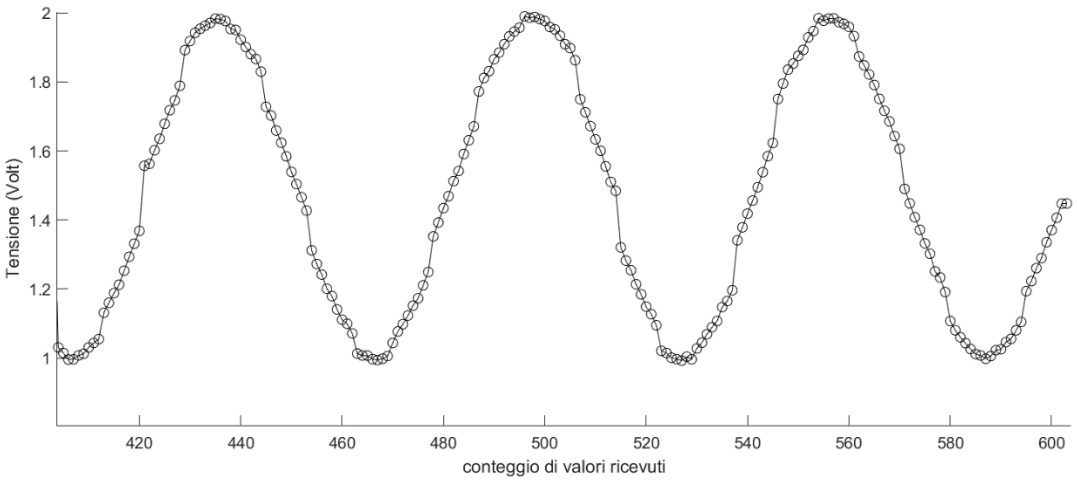


Figura 3.3.12 : Rappresentazione grafica di un test eseguito per evidenziare la non linearità di un segnale a 1 kHz, con marcatori che indicano i punti campionati.

Aumentando il Clock dell’ADC comporta conversoini più rapide, ma una frequenza di campionamento troppo elevata può causare problemi di jitter (variaizone nei tempi di campionamento), mentre diminuire il tempo di campionamento (“Sampling Time”) significa che il segnale analogico viene tenuto più a lungo prima di essere convertito in digitale, ed è utile per segnali a basse frequenze, perché aumento la velocità complessiva di conversione. Per segnali ad alte frequenze è utile avere un tempo minore di campionamento. È comunque importante trovare un equilibrio tra i due fattori in base al segnale che si sta acquisendo.

3.4 PROGETTAZIONE IN CUBE IDE PER L’ELABORAZIONE DI UN SEGNALE PWM

Prima di implementare il codice con una sorgente audio, per verificare la capacità dell’ADC di acquisire segnali ad alte velocità e nello stesso tempo elaborarli in tempo

reale, si è preferito utilizzare un segnale sinusoidale a frequenza stabile. Questo cambiamento consente di determinare se l'ADC possa garantire una risposta precisa e stabile anche in contesti dinamici. Siccome la scheda STM32F401RE non prevede un convertitore digitale-analogico (DAC), è stato impiegato un timer per generare un segnale PWM con frequenza costante ma con variazione del duty cycle. Questo segnale sarà sottoposto a un filtro passa-basso, in modo da ottenere un segnale sinusoidale che servirà come ingresso per l'ADC attraverso l'uso del DMA. Tale segnale verrà poi elaborato seguendo il medesimo approccio esaminato nei codici precedenti. Per la generazione del segnale PWM, è stato configurato il timer TIM2 per generare un'uscita sul pin PA15 (il pin del TIM2_CH1 è visualizzabile sul Pinout View). I valori dell'Auto Reload Register e Counter Period per il Timer1 sono stati scelti seguendo la seguente formula, tale da generare un segnale a frequenza di circa 1kHz.

$$F_{PWM} = \frac{Fclock}{(ARR+1)(CP+1)} = \frac{84M}{(8400)(10)} = 1000\ Hz$$

In questo caso è fondamentale considerare la risoluzione in bit del segnale PWM, data dal rapporto di questi due logaritmi in base 10.

$$Risoluzione_{PWM} = \frac{\log \frac{f_{clock}}{f_{PWM}}}{\log 2} = \frac{\log \frac{84MHz}{1000Hz}}{\log 2} \sim 16\ bits$$

Nelle successive figure sono illustrate le impostazioni del timer TIM2.

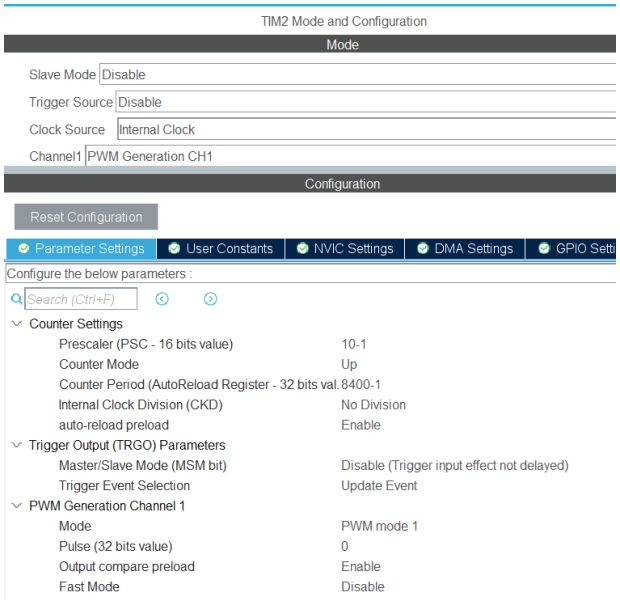


Figura 3.4.1 Impostazioni TIM2

È necessario abilitare il DMA tale che la generazione del PWM non interrompa le elaborazioni della CPU sui dati in ingresso all'ADC.

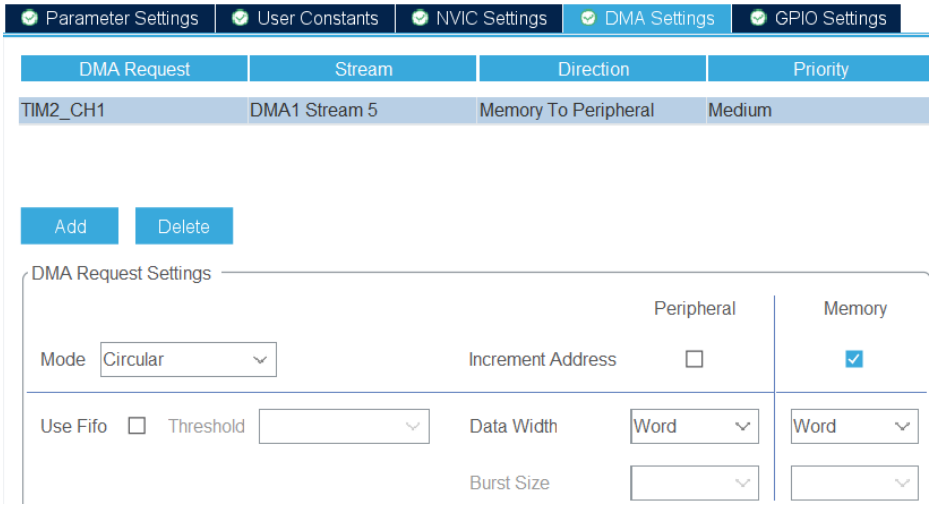


Figura 3.4.2 "DMA Settings" del TIM2

Sul PinOutView è possibile osservare l'ingresso dell'ADC (PA0) e l'uscita del segnale PWM (PA15).

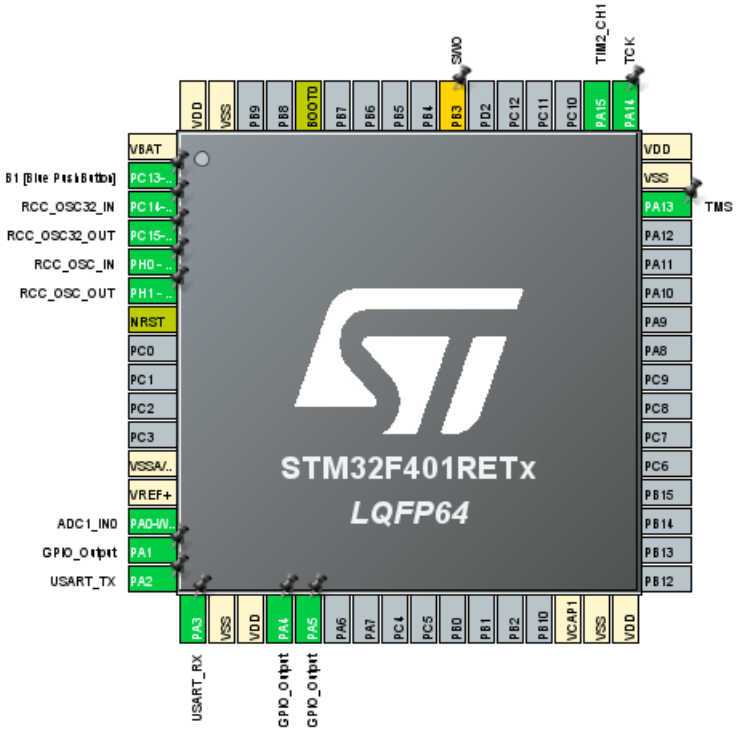


Figura 3.4.3 PinOut View

Per ottenere un segnale sinusoidale dalla forma quadra generata dal PWM, è necessario applicare un filtro passa-basso RC con una frequenza di taglio inferiore a quella del PWM. Questa configurazione consente di estrarre l'armonica corrispondente alla sinusoide risultante, contemporaneamente attenuando le altre frequenze presenti nel segnale in ingresso. La scelta della costante di tempo $\tau = RC$ deve essere tale da "smussare" il segnale PWM, ottenendo dalla forma quadra in ingresso una forma d'onda simile ad una sinusoide.

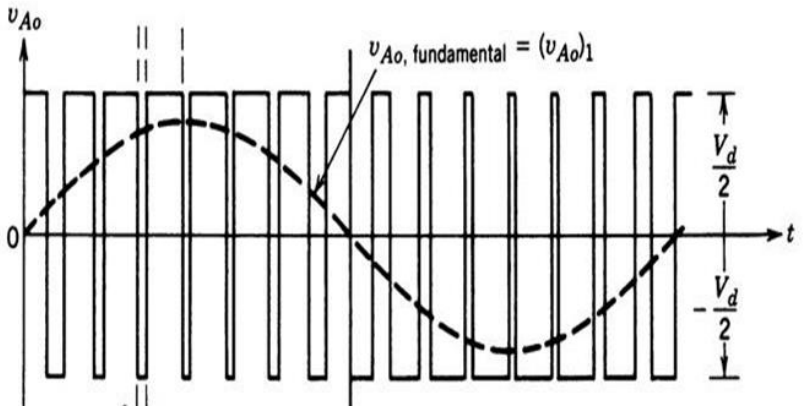
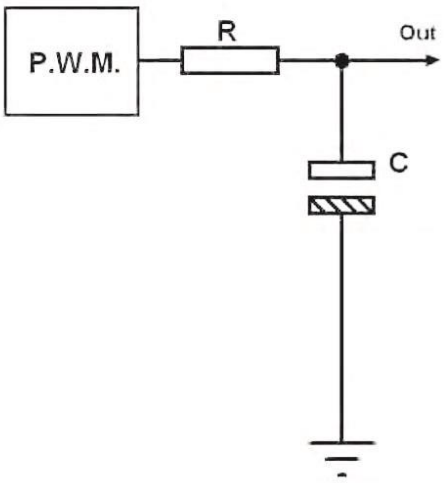


Figura 3.4.4 Diagramma illustrativo della trasformazione di un segnale PWM in una forma d'onda sinusoidale.

La frequenza di taglio del filtro passa-basso si calcola tramite la seguente formula

$$f_c = \frac{1}{2\pi RC}$$

Non esiste una regola precisa per determinare i valori della resistenza e del condensatore. Pertanto, sarà necessario ipotizzare una frequenza di taglio e sperimentare con i componenti disponibili al fine di trovare i valori adeguati che diano in uscita una forma d'onda soddisfacente. Il circuito è implementato sulla basetta millefori come è visibile in Figura 3.4.6. L'uscita del segnale PWM (PA15) è connessa alla resistenza, la quale è in serie con la capacità. Uno dei lati della capacità è connesso a massa (GND). Il segnale in uscita viene quindi letto dal condensatore e posto in ingresso all'ADC (PA0). Tramite diverse prove, è stato determinato che i valori ottimali per ottenere circa una forma d'onda pulita sono una resistenza di 1000 ohm e un condensatore da 10 microfarad. Questa combinazione produce una frequenza di taglio di circa 16 Hertz.



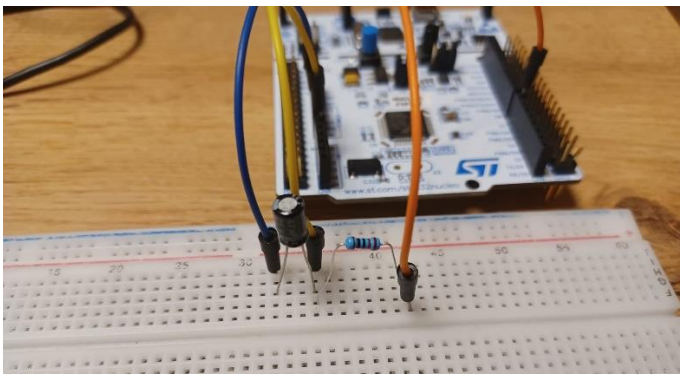


Figura 3.4.5 Schema del circuito RC

Figura 3.4.6 Implementazione del circuito RC

Per generare una sinusoide, è necessario che il Duty Cycle del mio PWM vari in modo continuo. Questo può essere ottenuto tramite una Look-Up Table, ovvero un elenco di valori pre-calcolati.

uint32_t DUTY[20]={ 8400, 7560, 6720, 5880, 5040, 4200, 3360, 2520, 1680, 840, 0, 840, 1680, 2520, 3360, 4200, 5040, 5880, 6720, 7560};

Questi valori sono ottenuti secondo la seguente formula del Duty Cycle conoscendo il valore dell'ARR(Auto-Reload Register), che è pari a 8400. I valori dell'array DUTY sono tali che, quando vengono sostituiti nel registro CCR1, consentano la variazione del Duty Cycle, dal 100% allo 0% e nuovamente al 100%, con variazioni del 10%.

$$DC = \frac{CCR1}{ARR} \cdot 100\%$$

Possiamo confermare il corretto funzionamento del segnale collegando l'uscita del TIM2_CH1, ossia il pin D7 al canale 0 del Logic Analyzer. In questo modo, otterremo un segnale digitale a 1kHz con una forma simile a quella mostrata nella Figura 3.4.7, che rappresenta la variazione del Duty Cycle.

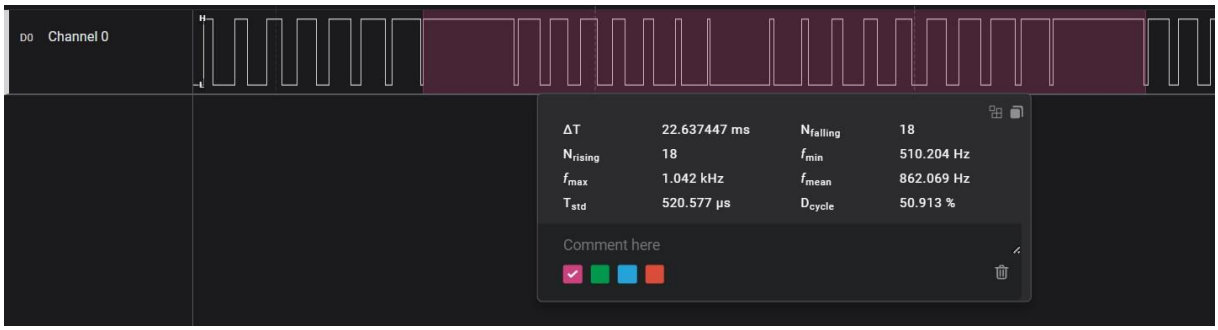
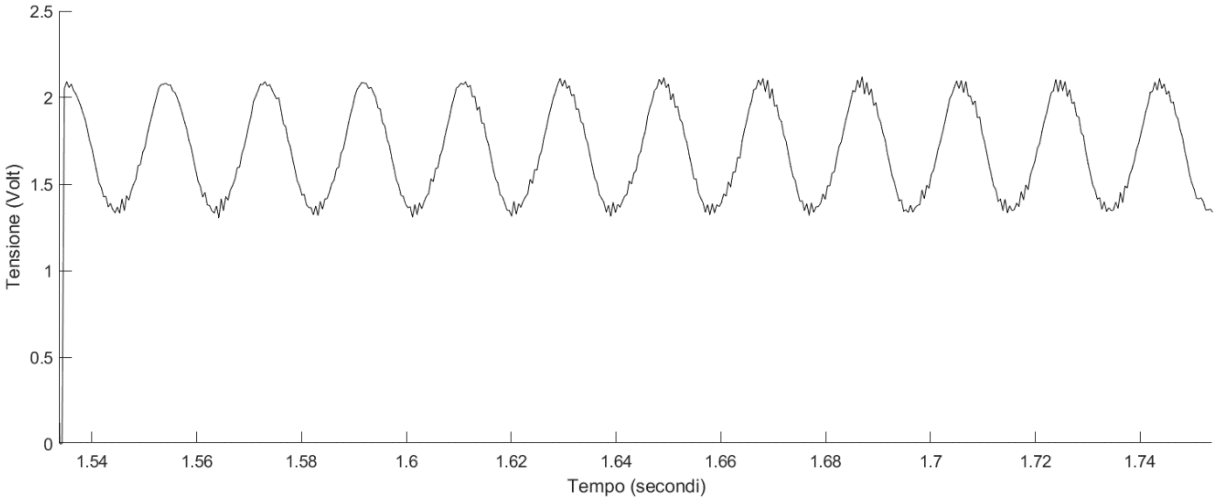


Figura 3.4.7 Analisi temporale mediante Logic Analyzer del segnale PWM a 1kHz con Duty Cycle che varia in 22ms.

Il corpo del codice rimarrà immutato rispetto al progetto originale, come si può osservare nel seguente codice.

```
   HAL_TIM_PWM_Start_DMA(&htim2, TIM_CHANNEL_1, DUTY, 20);
  HAL_ADC_Start_DMA(&hadc1,(uint32_t *) buffer,len);
/* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
        if(flag == 1){
      for(int i =0; i<len/2; i++){
        data[0] = (float)buffer[i] * 3.3f / 4096.0f;

              while (myflag == 0){;}
         if(myflag==1){
 if (HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK)
                                          {
                                    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_4);
                                             Error_Handler();
                                          }
                                       myflag = 0;
                                    }}
                          flag=0;
                    }
 if(flag1==1){
 for(int i =len/2; i<len; i++){
             data[0] = (float)buffer[i] * 3.3f / 4096.0f;

while (myflag == 0){;}
if(myflag==1){
  if (HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK) {

HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_4);
                               Error_Handler();}
                           myflag = 0;
                                                  }}
                          flag1=0;


                    }
    /* USER CODE END WHILE */
```

Nel contesto di MATLAB, è possibile visualizzare graficamente i risultati ottenuti. La Figura 3.4.8 mostra il grafico ottenuto dopo le elaborazioni.Le irregolarità nella sinusoide sono risultato della carenza di componenti con valori adeguati per la selezione di un filtro RC appropriato.



. *Figura 3.4.8 Rappresentazione grafica del segnale PWM in ingresso all'ADC*

CONCLUSIONI E SVILUPPI FUTURI

L'elaborazione dei segnali audio svolge un ruolo cruciale in molteplici settori, inclusi le telecomunicazioni, l'industria musicale e il campo biomedico. In questi ambiti, il monitoraggio e l'elaborazione dei dati in tempo reale riveste un'importanza vitale, come ad esempio la cancellazione del rumore, per garantire risultati accurati e affidabili. Il percorso che ha portato al progetto finale è stato guidato con successo dalla metodologia del "Test Driven Development". Questa strategia ha richiesto l'implementazione di test prima della stesura del codice di produzione, permettendo di stabilire con chiarezza le aspettative in termini di output atteso. Questo approccio non solo ha contribuito a ridurre gli errori nel prodotto finale, ma ha anche ottimizzato il processo di debugging. I test creati si sono concentrati sulla valutazione della tecnica di elaborazione iniziale e successivamente sulla gestione efficiente dei dati, operando a velocità elevate. I test condotti hanno fornito l'opportunità di sperimentare con diverse configurazioni delle impostazioni dell'ADC. Questo ci ha permesso di osservare le variazioni ottenute e di selezionare la migliore in base alle nostre specifiche esigenze.

Si è fatto ricorso al componente DMA, che ha permesso la parallelizzazione delle fasi di acquisizione ed elaborazione, portando a un notevole risparmio di potenza computazionale e tempo impiegato. Una volta garantita l'efficienza del progetto, è stata affrontata l'applicazione ad un segnale acustico.

Possiamo affermare che, sebbene la nostra analisi abbia confini definiti, siamo stati in grado di riprodurre fedelmente e in tempo reale diversi tipi di segnali, potendo così eseguire operazioni di elaborazione su tali segnali con accuratezza e tempestività.

BIBLIOGRAFIA

[1] https://www.digikey.it/en/maker/projects/getting-started-with-stm32-working-with-adc-and-dma/f5009db3a3ed4370acaf545a3370c30c Datasheet DMA : https://www.st.com/resource/en/application_note/dm00046011-using-the-stm32f2-stm32f4-and-stm32f7-series-dma-controller-stmicroelectronics.pdf

[2] https://www.digikey.it/en/maker/projects/getting-started-with-stm32-working-with-adc-and-dma/f5009db3a3ed4370acaf545a3370c30c

[3] Reference Manual per STM32F401RE : rm0368-stm32f401xbc-and-stm32f401xde-advanced-armbased-32bit-mcus-stmicroelectronics (2).pdf

[4] https://deepbluembedded.com/stm32-adc-tutorial-complete-guide-with-examples/?utm_content=cmp-true

[5] Appunti di Teoria dei Segnali, Prof.ssa Verdoliva Luisa : 410879 (unina.it)

[6] STM32 Change PWM Duty Cycle With DMA For Sine Wave Generation (deepbluembedded.com)

[7] https://www.dei.unipd.it/~pel/MicroC_e_DSP/Materiale_da_scaricare/Lezioni/Settimana%204/Lezione_12_(6_x_pagina).pdf

[8] L'onda sonora - Andrea Minini

[9] TESI_GAIACIOFFI_1203202.pdf

[10] Elaborazione dati e segnali biomedici | Appunti di Elaborazione Di Dati E Segnali Biomedici | Docsity

[11] Come fare lo spettrogramma di un segnale audio - ESPERIMENTANDA

APPENDICE A – FIRMWARE CUBE IDE

ELABORAZIONE DATI AUDI REAL-TIME

```
/* USER CODE BEGIN Header */
/**

******************************************************************************
***
  * @file           : main.c
  * @brief          : Main program body

******************************************************************************
***
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE
file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *

******************************************************************************
***
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------
---*/
#include "main.h"

/* Private includes ----------------------------------------------------
---*/
/* USER CODE BEGIN Includes */
#include <string.h>
#include <stdio.h>
/* USER CODE END Includes */
```

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_tx;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC1_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

```
/* Private user code ---------------------------------------------------
---*/
/* USER CODE BEGIN 0 */
volatile int flag = 0;
volatile int flag1 = 0;
volatile int myflag=1;
/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------
---*/

  /* Reset of all peripherals, Initializes the Flash interface and the
Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC1_Init();
  MX_USART2_UART_Init();
```

```
  /* USER CODE BEGIN 2 */
#define len 200
volatile uint16_t buffer[len];
float data[2];
uint8_t header[4]={0x0A,0x0A,0x0A,0x0A};
data[1]=*(float *)&header;
if ( HAL_ADC_Start_DMA(&hadc1,(uint32_t*)buffer,len) != HAL_OK)
                                                    {

                                                      Error_Handler();
                                                    }
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
        if(flag == 1){
              flag=0;
                    for(int i =0; i<len/2; i++){
             data[0] = (float)buffer[i] * 3.3f / 4096.0f;
       while (myflag == 0){;}
        if(myflag==1){
  if (HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK)       {

                                  Error_Handler();
                                                    }


                                            myflag = 0;
                                       }}   }

 if(flag1==1){
                     flag1=0;
                   for(int i =len/2; i<len; i++){
                   data[0] = (float)buffer[i] * 3.3f / 4096.0f;
 while (myflag == 0){;}
        if(myflag==1){
if (HAL_UART_Transmit_DMA(&huart2, (uint8_t *)&data, 5) != HAL_OK)
                                                          {
```

```
Error_Handler();
                                                                    }

 myflag = 0;
                                                          }} }

    /* USER CODE END WHILE */

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

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 16;
  RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
  RCC_OscInitStruct.PLL.PLLQ = 7;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
```

```
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV16;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data
Alignment and number of conversion)
  */
  hadc1.Instance = ADC1;
```

```
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = ENABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel its corresponding rank
in the sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */
```

```
  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA2_CLK_ENABLE();
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Stream6_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Stream6_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Stream6_IRQn);
  /* DMA2_Stream0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);
```

```
}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1|GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6,
GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : PA1 PA4 PA5 PA6 */
  GPIO_InitStruct.Pin = GPIO_PIN_1|GPIO_PIN_4|GPIO_PIN_5|GPIO_PIN_6;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}
```

```
/* USER CODE BEGIN 4 */
void HAL_ADC_ConvHalfCpltCallback (ADC_HandleTypeDef* hadc1){
      flag=1;}
void HAL_ADC_ConvCpltCallback (ADC_HandleTypeDef* hadc1){
    flag1=1;
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_6);

}
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart2)
{     myflag=1;
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_1);
}

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
```

```
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line
number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file,
line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```