INFORMATICA MUSICALE

UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
LAUREA TRIENNALE IN INFORMATICA
A.A. 2019/20
Prof. Filippo L.M. Milotta

ID PROGETTO: 09

TITOLO PROGETTO: Nucleo-Recorder

AUTORE 1: Scalisi Davide

Il progetto consiste nell'acquisizione e salvataggio di un segnale audio analogico tramite apposite apparecchiature analogiche e digitali.

Indice

1. Obiettivi del progetto 3

2. Metodo Proposto 3

2.1. Descrizione del circuito analogico 3

2.2. Descrizione del circuito digitale 4

2.3. Il firmware 5

2.3.1. Configurazione Timer 5

2.3.2. Configurazione ADC 6

2.3.3. Configurazione della UART 6

2.3.4. Acquisizione e salvataggio campioni 6

2.4. Il software ricevente 7

3. Risultati Ottenuti 7

4. Riferimenti esterni 8

1. Obiettivi del progetto

Grazie alle conoscenze acquisite in sede universitaria, all'esperienza ed alla curiosità personale, l'obiettivo principale di questo progetto prevede la realizzazione di un semplice dispositivo di digitalizzazione e salvataggio di un segnale audio o, più semplicemente, di un comune registratore digitale.

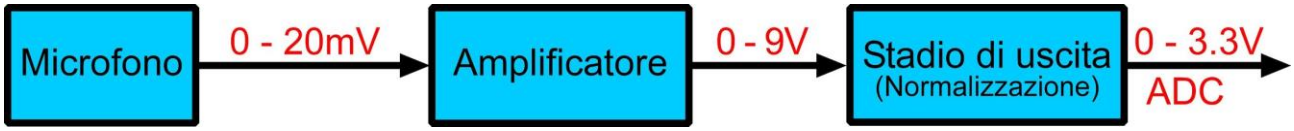
2. Metodo Proposto

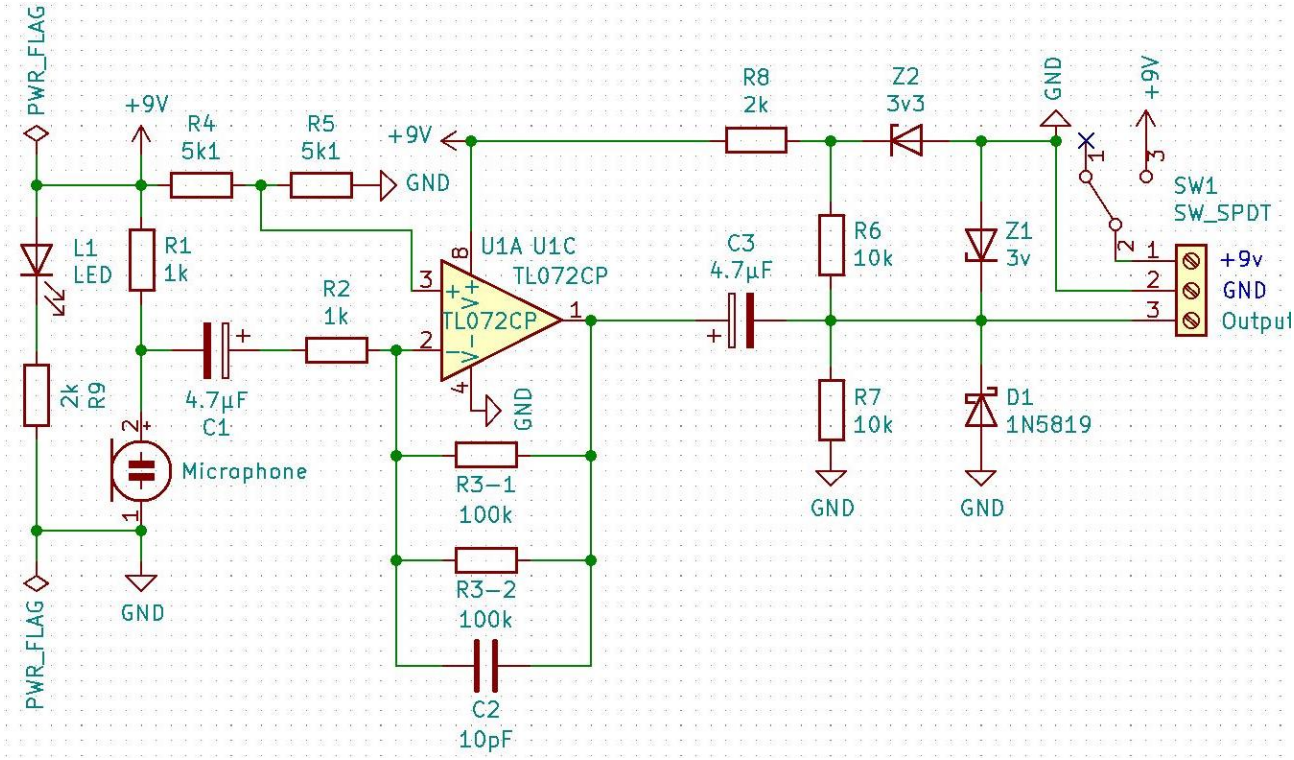
In questa fase, lo sviluppo del progetto sarà spiegata ed illustrata il più esaustivamente possibile, cercando di non tralasciare alcun dettaglio.

Verranno anche fatti riferimenti a librerie e tool esterni utilizzati: per maggiori informazioni su quest'ultimi, per i vari link di riferimento e per il download dei codici sorgenti, si rimanda il lettore all'ultimo capitolo, dove saranno citati in dettaglio tutti i riferimenti utilizzati in questo progetto.

2.1. Descrizione del circuito analogico

L'acquisizione del segnale audio sotto forma di differenza di potenziale avviene tramite un microfono, un circuito analogico di amplificazione ed infine, di un circuito di ripartizione (o normalizzazione) del segnale, il cui funzionamento viene brevemente spiegato nel seguente diagramma a blocchi e, più dettagliatamente, tramite il diagramma del circuito elettrico analogico:





Nel primo stadio, tramite un classico microfono a capsula o a condensatore, verrà generato il segnale di partenza che, sotto forma di differenza di potenziale, rappresenterà le onde sonore emesse dalla sorgente del suono. Questo segnale però avrà due principali problematiche:

1) Sarà troppo debole ed avrà la necessità di essere amplificato, in quanto varierà in un range di valori di soli **[V(R1)-10m,V(R1)+10m]V**.
2) Il segnale avrà un valore in continua (o valore costante) sommato, in quanto lo zero della tensione del microfono sarà pari alla tensione ai capi di **R1** (Vd. intervallo al punto 1).

Per risolvere tali problematiche, abbiamo inizialmente bisogno di "disaccoppiare" il segnale in ingresso: in breve, dobbiamo portare lo zero del segnale prodotto a 0 tramite l'utilizzo di **C1**: infatti, una delle tante proprietà dei condensatori è quella di filtrare la componente costante di un segnale, lasciando di conseguenza "passare" soltanto la parte variabile di quest'ultimo, centrandola proprio nello 0.
Una volta riportato il nostro segnale tra **[-10,10]mV**, siamo pronti per amplificare il segnale tramite un componente apposito: l'amplificatore operazionale **U1A**.

Tralasciando dettagli che esulano dal funzionamento di base del circuito, questo stadio amplifica il segnale di un fattore 200, dato dal rapporto **(R3_1 + R3_2) / R2**, portando però lo zero del segnale a 4.5V, valore determinato dividendo la tensione di alimentazione di un fattore 2 (9V/2=4.5V), ma con un intervallo di variazione di circa **[3,7]V**. Questa operazione di posizionamento dello zero a Vcc/2, viene effettuata tramite il partitore di tensione formato da **R4** ed **R5** per poter dare al segnale amplificato un intervallo di variazione più grande possibile, evitando così di introdurre clipping: fenomeno che provocherebbe distorsione, alterando il segnale originale.

Successivamente, lo scopo del condensatore **C3**, sarà lo stesso di C1, in quanto abbiamo bisogno nuovamente di ricondurre lo zero del segnale a 0, e quindi di disaccoppiarlo, eliminando nuovamente la componente continua del segnale.

Fatto questo, il nostro segnale varierà in un intervallo di circa **[-2,2]V**, che, per essere completamente apprezzato dal nostro ADC, dovrà essere riportato in un intervallo da **[0,Vout]**: in breve, abbiamo bisogno di normalizzare il segnale. Questo è lo scopo delle due resistenze **R6** ed **R7** che, insieme ad alcune feature di sicurezza offerte da **D1** e **Z1** (limitazione del voltaggio massimo e minimo del segnale prodotto), trasformano il nostro segnale nell'intervallo finale **[0,3.3]V**, pronto per essere acquisito digitalmente dal nostro ADC.

2.2. Descrizione del circuito digitale

Il segnale, per essere digitalizzato, deve essere acquisito digitalmente tramite un processo di **digitalizzazione**; a questo scopo, per l'implementazione, si è deciso di utilizzare un microcontrollore **NUCLEO-F401RE** di STMicroelectronics, scheda di sviluppo programmabile con features avanzate ed a prestazioni adatte per il compito che dovrà svolgere.

Lo sviluppo del firmware di questa board avviene utilizzando il linguaggio C e la comunicazione che questa board implementa con le sue stesse periferiche (come l'ADC interno, ad esempio), viene implementata a basso livello tramite semplici operazioni bitwise, in parte implementate dal Prof. **Corrado Santoro** nella sua libreria **<stm32_unict_lib.h>** ed in parte implementate manualmente da me.

Il principale lavoro svolto dal firmware della scheda, può essere riassunto nelle classiche operazioni di digitalizzazione: **campionamento**, **quantizzazione** e **trasmissione** dati.

2.3. Il firmware

Il firmware scritto, come detto prima, esegue operazioni molto semplici, ma non tutte sono state prevista dalla libreria **<stm32_unict_lib.h>**. Per poter permettere tali operazioni, la libreria è stata da me estesa al fine di poter supportare anche queste ultime.

Il funzionamento del codice inizia da una prima parte di inizializzazione di un **timer** del microcontrollore: questo ci serve per poter dire all'**ADC** ogni quanto deve prelevare un campione dal segnale.

Di conseguenza, le tempistiche di questo timer dovranno essere calcolate in modo da poter notificare l'ADC al momento giusto (alla fine di ogni periodo di campionamento).

2.3.1. Configurazione Timer

Di seguito, è riportata la parte di codice responsabile per l'inizializzazione generale del **TIM2** della Nucleo. Per far ciò, abbiamo bisogno di definire il nostro tasso di campionamento: personalmente, per ottenere dei valori interi dalle varie divisioni presenti nei calcoli ed una buona qualità audio, ho scelto come tasso di campionamento **16kHz**, ma, per motivi di funzionamento che probabilmente riguardano la bufferizzazione dei campioni da parte dell'UART, il dispositivo funziona correttamente al doppio della frequenza prevista; quindi tutti i calcoli saranno eseguiti con un tasso di campionamento di **32kHz**.

In questa configurazione, il timer funziona in OC (**Output Compare Mode**), una modalità di funzionamento che permette la generazione di un certo segnale con dei parametri dipendenti dalle impostazioni specifiche del timer, per scopi esterni al timer stesso.

In questo caso, il segnale generato esegue un toggle del suo stato ogni volta che è trascorso un periodo del nostro tasso di campionamento: questo evento è direttamente collegato alla fonte di trigger del nostro **ADC1**, configurato in modo tale da iniziare un'acquisizione ogni volta che questo segnale cambia il suo stato da HIGH a LOW e viceversa.

NB: In questo frammento di codice manca l'istruzione "TIM_on(TIM2);", in quanto quest'ultima è interna alle routine di interrupt dei quattro tasti della shield Unict per la scheda Nucleo; difatti, per avviare l'acquisizione, bisognerà premere uno di questi 4 pulsanti, abilitando così il timer.

```
//Campionamento a 32000Hz
//(84MHz/2625)*1=32kHz
//PSC=2625, ARR=1
TIM_config_timebase(TIM2,2625,1);

//Configurazione dell'evento di campionamento ad ogni tick di TIM2
TIM_set_CCR_val(TIM2,2,0);

//Ad ogni tick, inverti il segnale di trigger per l'ADC
TIM_config_OC(TIM2,2,OC_TOGGLE);
TIM_set_OC_polarity(TIM2,2,OC_ACTIVE_HIGH);

//TIM2 pronto
TIM_enable_OC(TIM2,2);
```

Una volta inizializzato il timer con i valori desiderati, possiamo andare a configurare il nostro **ADC1**:

2.3.2. Configurazione ADC

```
//Collegamento del pin PA0 sull'ingresso ADC1_IN0
ADC_channel_config(ADC1,GPIOA,0,0);

//Selezione di ADC1_IN0 (PA0)
ADC_sample_channel(ADC1,0);

//Abilito l'interrupt software di fine campionamento
ADC_enable_irq(ADC1);

//Avvia un'acquisizione ogni inversione del segnale di trigger
ADC_EXT_trig_source(ADC1,TRIG_BOTH_EDGES);

//Il segnale di trigger proviene da TIM2
ADC_EXT_trig_mode(ADC1,TIM2_CC2);

//Attiva l'ADC1
ADC_on(ADC1);
```

Queste operazioni di inizializzazione, oltre alla selezione del canale, della sorgente del segnale e dell'allineamento dei bit del campione, specificano anche che alla fine di ogni operazione di campionamento deve essere scatenato un **interrupt** software per il salvataggio e la trasmissione del campione stesso. Ogni campione deve essere catturato quando specificato dal segnale proveniente da **TIM2** che, appunto, prima abbiamo configurato per emettere una frequenza pari al tasso di campionamento.

2.3.3. Configurazione della UART

La configurazione della porta seriale o **UART** (Universal Asynchronous Receiver-Transmitter o più semplicemente, porta seriale) è quasi interamente prevista nella libreria **<stm32_unict_lib.h>**. Infatti, per poter utilizzare la UART integrata tramite USB, basta semplicemente specificare la velocità di comunicazione tra la Nucleo ed il dispositivo connesso a quest'ultima tramite la semplice istruzione "CONSOLE_init(B1M);", dove "B1M" è una costante di libreria che specifica una velocità di **1Mbps**.

La scelta della velocità dipende dal nostro **bitrate**: con 32kHz di tasso di campionamento, 2 Byte di risoluzione ed un solo canale, abbiamo bisogno di una larghezza di banda di almeno *32k x 16 =* ***512kbps***.

2.3.4. Acquisizione e salvataggio campioni

Una volta stabilite le tempistiche e configurato l'hardware di conseguenza, vale la pena di dare un'occhiata alla codifica ed alla trasmissione dei campioni acquisiti:

```
//Il bit del segno, se negativo, indica la parte alta del numero a 16 bit.
void write_12(uint16_t data){
  __io_putchar((uint8_t)((data>>6) | (1<<7)));
  __io_putchar(((uint8_t)data) & ((1<<6)-1));
}

void ADC_IRQHandler(){
  if(ADC_isset(ADC1)){
    n=ADC_read(ADC1);
    write_12(n);
    ADC_clear(ADC1);
  }
}
```

Per prima cosa, viene letto il valore campionato dall'ADC, poi il valore letto viene inviato tramite l'UART al dispositivo ricevente tramite una **codifica** specificata nella funzione "**write_12**": per prima cosa i 12 bit del campione vengono divisi in 6 bit della parte alta e 6 bit della parte bassa. Successivamente, in questi 2 nuovi numeri ottenuti, si lavorerà su l'ultimo bit a sinistra: il bit del **segno**, che al livello di codifica servirà a distinguere la parte alta dalla parte bassa del numero in fase di decodifica. Questo sistema ci permetterà sempre di avere sempre il giusto **byteorder** in fase di decodifica.

2.4. Il software ricevente

Per poter interpretare, manipolare e salvare con una certa codifica i campioni acquisiti, abbiamo bisogno di un software di acquisizione che ci permetta di compiere tali manipolazioni: in questo caso, nella scelta del linguaggio di programmazione, si è deciso di utilizzare Java. Questa scelta deriva semplicemente dal fatto che, dovendo essere testato alternativamente su sistemi Windows e Linux, ho preferito utilizzare un linguaggio di facile portabilità. In questa breve descrizione non verrà citato il codice Java del software, che però sarà sempre reperibile per il download nell'area download dell'ultimo capitolo.

Il software descritto utilizza 2 librerie esterne:

1) **jSerialComm**: Per la gestione della porta seriale.
2) **Java Wav File IO**: Per la gestione di file wav.

Tramite l'utilizzo di queste due librerie esterne, le funzionalità di questo programma possono essere riassunte in queste semplici operazioni:

1) **Lettura** dei campioni trasmessi dalla porta seriale.
2) **Decodifica** e ricomposizione del campione a 12 bit (tramite la tecnica del bit di segno).
3) **Normalizzazione** del campione da **12 bit [0,4095]** a **16 bit [-32.768,32.767]**.
4) **Scrittura** di tutti i campioni normalizzati nel file specificato

Ovviamente, tramite opportuni tool o librerie, come ffmpeg ad esempio, sarebbe possibile applicare dei filtri software per migliorare la qualità audio finale, ottenendo così un suono di qualità superiore; ma questo esula dallo scopo principale del progetto.

Questo software è un semplice software in linea di comando: dopo l'esportazione come file .jar, va eseguito tramite shell con il comando *"java -jar Nucleo-Recorder.jar <nome_file.wav> <durata>"*.

Ad esempio, se volessimo registrare 5 secondi di audio in un file di nome "prova.wav", dovremo utilizzare il seguente formato: *"java -jar Nucleo-Recorder.jar prova.wav 0.05"*.

3. Risultati Ottenuti

L'obiettivo finale di questo progetto è di ottenere una qualità discreta dell'audio acquisito cercando allo stesso tempo di mantenere bassa la difficoltà computazionale dell'intero sistema, in quanto l'hardware utilizzato ha delle limitazioni e non è stato progettato specificatamente per il compito a cui lo si sta sottoponendo, ma è un hardware general-purpose.

Sotto questa premessa, il risultato finale ottenuto è stato inizialmente deludente qualitativamente parlando, in quanto vi era la presenza di varie problematiche a livello hardware e di bug a livello software. Successivamente, dopo svariate correzioni sia dal punto di vista elettronico, sia dal punto di vista del codice, posso affermare di essere riuscito ad ottimizzare il risultato finale, superando le iniziali aspettative che mi ero prefisso.

Successivamente, si potrebbe pensare ad espandere il progetto tramite il perfezionamento del firmware della scheda Nucleo per poterlo rendere, ad esempio, indipendente da un dispositivo ricevente ed anche in grado di riprodurre e memorizzare l'audio acquisito tramite appositi circuiti elettronici e memorie di massa.

D'altra parte, si potrebbe pensare di poter scrivere un driver apposito per poter utilizzare il dispositivo come una rudimentale scheda audio per l'acquisizione diretta dell'audio digitalizzato.

Queste per il momento sono solo idee, ma in un prossimo futuro verranno probabilmente implementate.

4. Riferimenti esterni

- **Sorgenti, circuito e PCB**
- **Corso di Laboratorio sistemi a microcontrollore**
- **NUCLEO-F401RE**
- **jSerialComm**
- **Java Wav File IO**
- **ffmpeg**
- IDE Java: **Eclipse**
- IDE Nucleo: **OpenStm32**
- Editor circuito elettrico e PCB: **Kicad**