Music to DMX

Matteo Di Bartolomeo
241469@studenti.unimore.it

Federico Silvestri
243938@studenti.unimore.it

Abstract

Il progetto si pone come obiettivo il controllo di apparecchiature luminose ed effetti scenici per eventi in base al ritmo della musica tramite microcontrollore.
Spesso installazioni piccole (come un dj-set) non prevedono una vera e propria regia, ma è lo stesso musicista a dover occuparsi della gestione delle luci e degli effetti scenici come macchine del fumo o delle fiamme. Per questo motivo si è cercato di realizzare un sistema che, tramite il riconoscimento del beat della canzone in riproduzione, si occupasse in modo autonomo della gestione delle luci creando un effetto gradevole alla vista e permettesse anche di integrare altri effetti scenici tramite una pulsantiera.

1. Protocollo DMX512

Le luci e i macchinari per effetti utilizzati nel mondo dello spettacolo utilizzano uno standard di comunicazione digitale chiamato DMX512, il quale permette di controllare i dispositivi dalla regia.
Questo standard si basa sul modello fisico RS-485 e prevede che il collegamento avvenga tramite cavi XLR a 5 poli, anche se nella pratica si utilizzano cavi a 3 poli.
I dati vengono trasmessi in modo seriale a 250 kbit/s e sono raggruppati in pacchetti di dimensione massima di 513 byte.

2. Progettazione

2.1. Hardware

La schedina che si occupa della gestione e dell'esecuzione di tutto il progetto è un microcontrollore appartenente alla famiglia STM32, più precisamente il modello B-L475E-IOT1A.
Per l'acquisizione dell'audio avremmo potuto utilizzare il microfono integrato presente sulla schedina ma abbiamo preferito utilizzarne uno esterno(modello MAX9814) che ci ha consentito di avere un migliore risultato, grazie anche ad una migliore gestione del guadagno (gain). Infatti, utilizzando il microfono in ambienti molto rumorosi si rischia che l'acquisizione dell'input audio venga disturbata da rumori molto forti (es. musica ad alto volume, suoni esterni ecc ecc) i quali portano il microfono a lavorare oltre i suoi limiti determinando il fenomeno del *clipping* della forma d'onda che descrive il segnale audio in entrata.
Per poter comunicare con le luci tramite lo standard DMX si ha avuto necessità anche di un modulo che permettesse la conversione del'output inviato dalla scheda in modo da adattatarla al protocollo fisico RS-485.
Dunque, a questo modulo è stato saldato un connettore XLR che ha permesso poi il collegamento e l'interfacciamento di tutta l'attrezzatura.
Infine, visto che tra i dispositivi da coordinare sono presenti anche delle macchine delle fiamme, si è pensato di aggiungere ad ognuna un sensore di distanza ad ultrasuoni (modello HC SR04) con lo scopo di evitare la loro attivazione qualora qualcuno dovesse essere troppo vicino ad esse aumentando la safety del sistema.

2.2. Software

Per la realizzazione del software ci si è basati sul sistema operativo FreeRTOS. Il codice è scritto in linguaggio C (si è cercato di seguire i criteri di sicurezza MISRA-C 2004).
Fondamentale è stato l'utilizzo di GitHub che ci ha permesso di lavorare anche durante quest'ultimo periodo di pandemia in cui non ci è stato possibile lavorare fisicamente insieme al progetto. Infatti, tramite esso è stato possibile condividere tutti i file relativi al progetto e tramite l'utilizzo di branch separati è stato possibile lavorare in autonomia, facendo anche delle prove di configurazioni differenti, ad aspetti differenti del progetto per poi riunirli insieme effettuando il merge dei branch creati con il main branch.
Inoltre sfruttando e osservando i commit eseguiti da ognuno è sempre stato possibile vedere le modifiche che sono state fatte al progetto e i commenti lasciati dall'autore del commit stesso.

2.2.1 Task

Il programma è composto da 5 task fondamentali:

1. **Input recording**: si occupa dell'acquisizione dell'audio salvandolo temporaneamente in un buffer di 170 int. Tale dimensione è stata stabilita successivamente ad un processo di *fine tuning* grazie al quale si

è notato che una dimensione minore avrebbe influito negativamente sulle performance del sistema, mentre una dimensione maggiore avrebbe comportato uno spreco di memoria;

2. **Input processing**, caratterizzato da due fasi:
   (a) Preprocessing dell'input, dove vengono applicati dei filtri (2 per la precisione) al segnale presente nel buffer per migliorarne la qualità ed eliminare le frequenze che non sono di nostro interesse;
   (b) Elaborazione del segnale filtrato, che ha reso possibile il riconoscimento del beat.
3. **Send output**, il quale invia il segnale che pilota i vari dispositivi (luci, macchine del fumo e delle fiamme);
4. **Fog**, che si occupa di intercettare l'interrupt generato dal bottone che permetterà di azionare la macchina del fumo;
5. **Fire**, che intercetta l'interrupt del bottone che permetterà di attivare le macchine delle fiamme. Quest'ultime, come già anticipato in precedenza, saranno attivate solamente nel caso in cui non ci sia nessuno nelle loro immediate vicinanze..

Il task di registrazione è periodico (periodo 250 ms, 4 Hz) in modo da cercare di stimare meglio la velocità con il quale si susseguono i picchi all'interno del segnale di input. In particolare, si è scelto un periodo di 250 ms in quanto, così, è possibile gestire tutte le canzoni al di sotto dei 240 bpm (si consideri che di norma le canzoni vanno dai 85 - 95 bpm dell'Hip Hop ai 170 - 174 bpm dell' Hardstyle e della Drum and Bass). La frequenza di campionamento del task (1000 Hz) è stata stabilita sfruttando il teorema di Nyquist in modo da riuscire a campionare le nostre frequenze d'interesse, quelle dei bassi, senza la neccessità di salvare inutili informazioni riguardo frequenze maggiori.
Il task di processamento dei dati è sincronizzato con il task di acquisizione tramite un semaforo e si occupa dell'individuazione dei picchi all'interno dello spettro di frequenze campionato. In particolare, si è fatto in modo che il sistema fosse in grado di adattarsi alle varie parti delle canzoni (intro, strofa, ritornello, bridge ...) resettando i valori del picco massimo e minimo registrati ogni 60 acquisizioni (circa ogni 15 secondi).
L'invio dei dati di output è anch'esso periodico e prevede l'invio di un risultato ogni 41 ms (24Hz). La periodicità del task deriva dal fatto che, in mancanza della ricezione di un segnale, le luci si spengono, dunque è necessario fare il refresh dell'input, nel caso in cui non sia stato rilevato un picco all'interno della canzone, o inviare un nuovo segnale nel caso opposto. Per determinare il valore del periodo invece ci si è basati sulle frequenze di aggiornamento dei dispositivi in commercio che si aggirano intorno ai 24 Hz.
Come anticipato in precedenza, per migliorare la qualità del segnale in ingresso al sistema sono stati applicati due filtri:

1. Il primo si occupa di escludere le frequenze molto basse che sono caratterizzate da rumore e disturbi (0Hz-50Hz);
2. Il secondo taglia le frequenze alte (al di sopra dei 300Hz).

Per scegliere le frequenze corrette abbiamo analizzato le risposte in frequenza della maggior parte degli speaker in commercio e abbiamo notato che la banda di frequenze d'interesse è proprio quella delle basse frequenze (50 - 300 Hz).
Si è cercato di ottimizzare il più possibile l'occupazione di memoria evitando il sovradimensionamento delle variabili utilizzate, nonché il sovrannumero delle stesse, e modificando i file di configurazione di FreeRTOS in modo da diminuire le strutture caricate.

3. Testing

Il testing è stato fondamentale per cercare di eseguire il fine-tuning dei parametri necessari affinchè il gioco di luci finale risultasse il più possibile fluido e dinamico e, soprattutto, rispettasse i tempi dettati dalla musica che era in riproduzione.
Durante questa fase, infatti, il lavoro si è concentrato sull'ampliamento o la riduzione della banda di frequenze utilizzate per l'analisi del segnale d'ingresso e sulla modifica delle soglie utili al cambiamenot del colore e del movimento delle luci.

4. Studio di schedulabilità

Task name	Worst case	Ours	Period
InputRec	37.6 ms	37.6 ms	250 ms
InputProc	1 ms	1 ms	250 ms
SendOutput	**24 ms**	**3 ms**	41 ms
Fog	37 μs	37 μs	-
Fire	48 μs	48 μs	-

Per poter effettuare l'analisi di schedulabilità si è considerato ovviamente il WCET. In particolare, per quanto riguarda il task *SendOutput* si è considerato il caso in cui vengano utilizzati tutti e 512 canali che il protocollo DMX mette a disposizione (nel gergo tecnico si dice che si è connesso l'universo). Nel nostro caso, in base ai dispositivi in nostro possesso, abbiamo utilizzato solo 58 canali (2 teste mobili da 17 canali l'una, 3 luci da 6 canali ognuna, 1 macchina del fumo e 2 macchine delle fiamme da 2 canali caduna). Questa è la motivazione dietro la grande differenza fra il WCET e il nostro use-case per quanto riguarda il suddetto

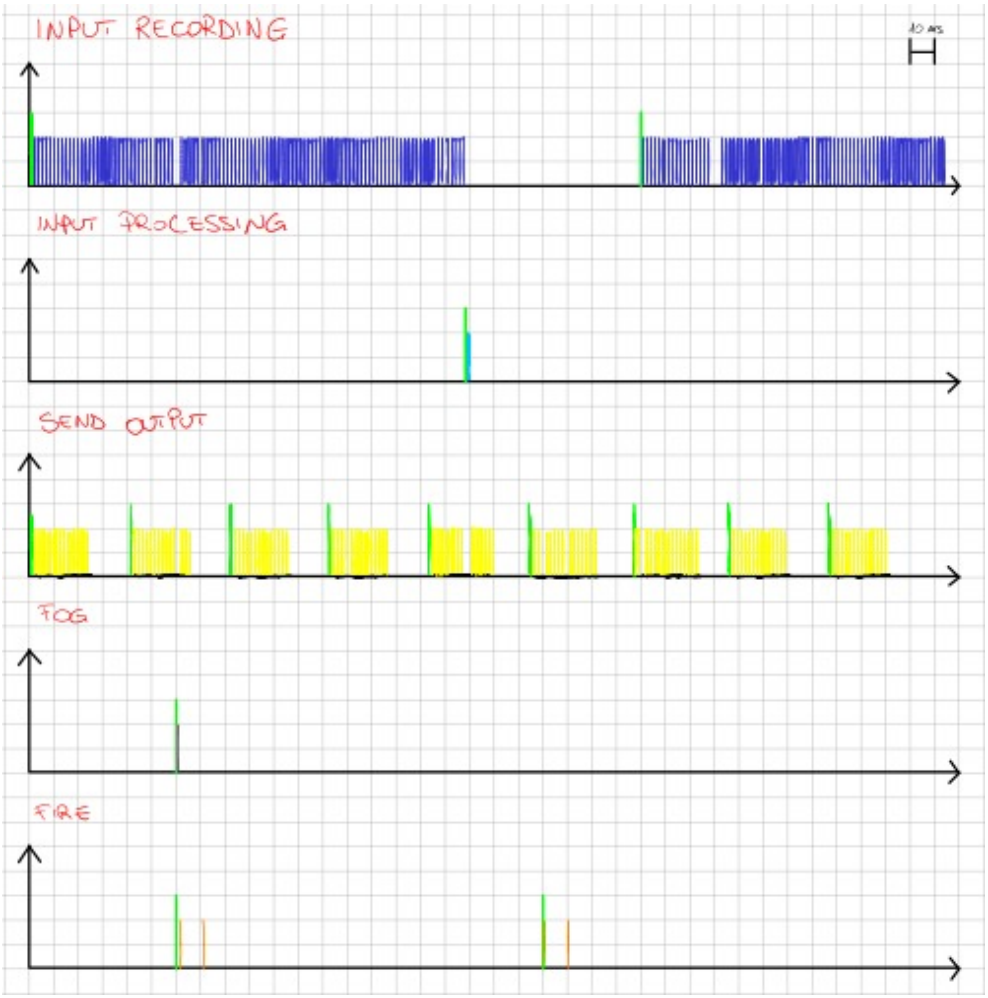

Figure 1. Studio di schedulabilità

task.
Il task che controlla la macchina del fumo e delle fiamme sono gestiti tramite interrupt e, poiché tali dispositivi vengono attivati poche volte e per un breve lasso di tempo, essi non sono presenti spesso all'interno dell'analisi in quanto aperiodici.
Il task di registrazione è periodico e la sua deadline è impostata in modo tale da riuscire a registrare ogni quarto di qualsiasi canzone gli venga sottoposta permettendo, così, una migliore analisi del beat e del ritmo.

5. Utilizzazione

Nel calcolo dell'utilizzazione, come in precedenza, si è sempre considerato il caso peggiore. Avendo due task aperiodici, il caso maggiormente sfavorevole è quando essi vengono considerati come periodici con tempo interarrivo pari al minore fra quelli dei task sincroni (41 ms).

5.1. Utilizzazione del processore

L'utilizzazione del processore risulta essere del 74.18 % nel caso in cui si connetta l'universo, mentre nel nostro caso, del 22.96 % . La formula utilizzata per il calcolo è la seguente:

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1}$$

in cui C_i indica il tempo di esecuzione del task i-esimo e T_i il corrispettivo minimo tempo interarrivo.
Poichè l'utilizzazione del processore è minore di 1, il processore non è sovraccaricato.

5.2. Utilizzazione della memoria

Per quanto riguarda l'utilizzazione di memoria, il programma utilizza 41428 byte (3%) dello spazio disponibile per i programmi. Il massimo è 1048576 byte.
Le variabili globali usano 2936 byte (2%) di memoria di-

namica, lasciando altri 95368 byte liberi per le variabili locali. Il massimo è 98304 byte.

A. Misra-C 2004 rules

- Rule 1.2: No reliance shall be placed on undefined or unspecified behaviour.
- Rule 1.3: Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.
- Rule 2.2: Source code shall only use /*...*/ style comments
- Rule 2.3: The character sequence /* shall not be used within a comment.
- Rule 2.4: Sections of code should not be "commented out"
- Rule 3.2: The character set and the corresponding encoding shall be documented.
- Rule 3.6: All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.
- Rule 4.1: Only those escape sequences that are defined in the ISO C standard shall be used.
- Rule 4.2: Trigraphs shall not be used.
- Rule 5.1: Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
- Rule 5.2: Identifiers in an inner scope shall not use the same name as an identifiers in an outer scope, and therefore hide that identifier.
- Rule 5.3: A typedef name shall be a unique identifier.
- Rule 5.4: A tag name shall be a unique identifier.
- Rule 5.5: No object or function identifier with static storage duration should be reused.
- Rule 5.6: No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure member and union member names.
- Rule 6.1: The plain char type shall be used only for the storage and use of character values.
- Rule 6.2: signed and unsigned char type shall be used only for the storage and use of numeric values.
- Rule 6.3: typedefs that indicate size and signedness should be used in place of the basic numerical types.
- Rule 6.5: Bit fields of signed type shall be at least 2 bits long.
- Rule 7.1: Octal constants (other than zero) and octal escape sequences shall not be used.
- Rule 8.1: Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
- Rule 8.2: Whenever an object or function is declared or defined, its type shall be explicitly stated.
- Rule 8.3: For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.
- Rule 8.4: If objects or functions are declared more than once their types shall be compatible.
- Rule 8.5: There shall be no definitions of objects or functions in a header file.
- Rule 8.6: Functions shall be declared at file scope.
- Rule 8.7: Objects shall be defined at block scope if they are only accessed from within a single function.
- Rule 8.8: An external object or function shall be declared in one and only one file.
- Rule 8.9: An identifier with external linkage shall have exactly one external definition.
- Rule 8.10: All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
- Rule 9.1: All automatic variables shall have been assigned a value before being used.
- Rule 9.2: Braces shall be used to indicate and match the structure in the non-zero initialisation of arrays and structures
- Rule 10.1: The value of an expression of integer type shall not be implicitly converted to a different underlying type if: (a) it is not a conversion to a wider integer type of the same signedness, or (b) the expression is complex, or (c) the expression is not constant and is a function argument, or (d) the expression is not constant and is a return expression

- Rule 10.2: The value of an expression of floating type shall not be implicitly converted to a different type if: (a) it is not a conversion to a wider floating type, or (b) the expression is complex, or (c) the expression is a function argument, or (d) the expression is a return expression
- Rule 11.1: Conversions shall not be performed between a pointer to a function and any type other than an integral type.
- Rule 11.2: Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.
- Rule 11.3: A cast should not be performed between a pointer type and an integral type.
- Rule 11.4: A cast should not be performed between a pointer to object type and a different pointer to object type.
- Rule 12.1: Limited dependence should be placed on C's operator precedence rules in expressions.
- Rule 12.2: The value of an expression shall be the same under any order of evaluation that the standard permits.
- Rule 12.3: The sizeof operator shall not be used on expressions that contain side effects.
- Rule 12.10: The comma operator shall not be used.
- Rule 12.13: The increment (++) and decrement (–) operators should not be mixed with other operators in an expression
- Rule 13.1: Assignment operators shall not be used in expressions that yield a Boolean value.
- : Rule 13.2: Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.
- Rule 13.4: The controlling expression of a for statement shall not contain any objects of floating type.
- Rule 13.5: The three expressions of a for statement shall be concerned only with loop control.
- Rule 13.6: Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.
- Rule 13.7: Boolean operations whose results are invariant shall not be permitted.
- Rule 14.1: There shall be no unreachable code.
- Rule 14.4: The goto statement shall not be used.
- Rule 14.5: The continue statement shall not be used.
- Rule 14.6: For any iteration statement there shall be at most one break statement used for loop termination.
- Rule 14.7: A function shall have a single point of exit at the end of the function.
- Rule 14.8: The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.
- Rule 14.9: An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement
- Rule 16.1: Functions shall not be defined with a variable number of arguments.
- Rule 16.2: Functions shall not call themselves, either directly or indirectly.
- Rule 16.3: Identifiers shall be given for all of the parameters in a function prototype declaration.
- Rule 16.4: The identifiers used in the declaration and definition of a function shall be identical
- Rule 16.5: Functions with no parameters shall be declared and defined with the parameter list void
- Rule 16.6: The number of arguments passed to a function shall match the number of parameters.
- Rule 16.8: All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
- Rule 17.1: Pointer arithmetic shall only be applied to pointers that address an array or array element.
- Rule 17.4: Array indexing shall be the only allowed form of pointer arithmetic.
- Rule 17.3: The declaration of objects should contain no more than 2 levels of pointer indirection.
- Rule 17.6: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.
- Rule 18.1: All structure and union types shall be complete at the end of a translation unit.

- Rule 18.2: An object shall not be assigned to an overlapping object.
- Rule 18.3: An area of memory shall not be reused for unrelated purposes.
- Rule 18.4: Unions shall not be used.
- Rule 19.1: #include statements in a file should only be preceded by other preprocessor directives or comments.
- Rule 19.2: Non-standard characters should not occur in header file names in #include directives.
- Rule 19.3: The #include directive shall be followed by either a ¡filename¿ or ”filename” sequence.
- Rule 19.4: C macros shall only expand to a braced initialiser, a constant, a string literal, a parenthesised expression, a type qualifier, a storage class specifier, or a do-while-zero construct.
- Rule 19.5: Macros shall not be #define'd or #undef'd within a block.
- Rule 19.6: #undef shall not be used.
- Rule 19.7: A function should be used in preference to a function-like macro.
- Rule 19.12: There shall be at most one occurrence of the # or ## operators in a single macro definition.
- Rule 19.14: The defined preprocessor operator shall only be used in one of the two standard forms.
- Rule 19.15: Precautions shall be taken in order to prevent the contents of a header file being included twice.
- Rule 20.1: Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.
- Rule 20.2: The names of standard library macros, objects and functions shall not be reused.
- Rule 20.4: Dynamic heap memory allocation shall not be used
- Rule 20.5: The error indicator errno shall not be used.
- Rule 20.7: The setjmp macro and the longjmp function shall not be used.
- Rule 20.6: The signal handling facilities of ¡signal.h¿ shall not be used.
- Rule 20.9: The input/output library ¡stdio.h¿ shall not be used in production code.
- Rule 20.10: The library functions atof, atoi and atol from library ¡stdlib.h¿ shall not be used.
- Rule 20.11: The library functions abort, exit, getenv and system from library ¡stdlib.h¿ shall not be used.
- Rule 20.12: The time handling functions of library ¡time.h¿ shall not be used.
- Rule 21.1: Minimisation of run-time failures shall be ensured by the use of at least one of (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.