

MSc in Computer Science
at University of Milan

Emulatore CHIP-8 su STM32
Relazione per il progetto di PROS,
corso tenuto da **Danilo Bruschi**

federico.bruzzone@studenti.unimi.it

lorenzo.ferrante1@studenti.unimi.it

andrea.longoni3@studenti.unimi.it

Federico Bruzzone

Lorenzo Ferrante

Andrea Longoni

Anno accademico 2022/2023

Contents

1	Introduzione	3
2	Stato dell'arte	3
3	Hardware	4
3.1	Componenti utilizzati e relativi costi	5
3.2	Schema di collegamento	5
4	Software	7
4.1	Interprete CHIP-8	7
4.1.1	Gestione del timing	8
4.1.2	Ottimizzazioni	8
4.1.3	Comportamenti ambigui	9
4.2	Porting su STM32	9
4.2.1	Architettura software	10
4.2.2	Interfaccia con la scheda microSD	11
4.2.3	Menù di selezione	11
4.2.4	Rendering del font	11
4.2.5	Funzionamento del keypad	12
4.2.6	Interfaccia con lo schermo	12
5	Assemblaggio	13
6	Analisi del consumo energetico	14
7	Considerazioni finali	14
Riferimenti bibliografici		15
Appendix A Che aspetto ha un programma CHIP-8?		16

List of Figures

4	Schema dei collegamenti.	6
5	Esempio della mappatura di un pixel.	9
6	Class diagram.	10

7	Sequence diagram.	10
8	Esempio della mappatura di un carattere del font.	11
9	Assemblaggio finale.	13

List of Tables

1	Materiali utilizzati per la realizzazione del progetto.	5
---	-----------------------------------------------------------------	---

Abbiamo deciso di creare un'organizzazione GitHub per rendere accessibile il codice sorgente del nostro progetto ad altri programmati open-source coinvolti con lo sviluppo di emulatori e con la community online di CHIP-8.

Link all'organizzazione:

- *Macchina virtuale: <https://github.com/CHIP-8-Org/Core>*
- *Port su STM32: <https://github.com/CHIP-8-Org/STM32-Port>*
- *Documentazione: <https://github.com/CHIP-8-Org/Docs>*

1 Introduzione

CHIP-8 è un linguaggio di programmazione creato a metà degli anni '70 da Joseph Weisbecker per semplificare lo sviluppo di videogiochi per microcomputer a 8 bit. I programmi CHIP-8 vengono interpretati da una macchina virtuale che è stata estesa parecchie volte nel corso degli anni, tra le versioni più adottate citiamo S-CHIP e la più recente XO-CHIP.

La semplicità dell'interprete in aggiunta alla sua lunga storia e popolarità hanno fatto sì che emulatori e programmi CHIP-8 vengano realizzati ancora oggi. Nel corso degli anni molti videogiochi storici sono stati riscritti in CHIP-8 tra cui Pong, Space Invaders e Tetris.

Lo scopo del progetto è quello di realizzare un emulatore CHIP-8 e S-CHIP in grado di funzionare su un microcontrollore STM32.

In questo documento ci riferiremo alla macchina virtuale che interpreta programmi CHIP-8 con "interprete". Mentre utilizzeremo "emulatore" per indicare l'interprete assieme ad una sua implementazione (o "port"), ovvero un programma che gestisce l'audio, il video, l'input da tastiera e interagisce con l'API della macchina virtuale.

2 Stato dell'arte

Al giorno d'oggi risulta difficile ottenere un numero esatto di utenti che utilizzano CHIP-8, un buon indicatore può essere il topic "chip8" di GitHub che raggruppa quasi un migliaio di repository.

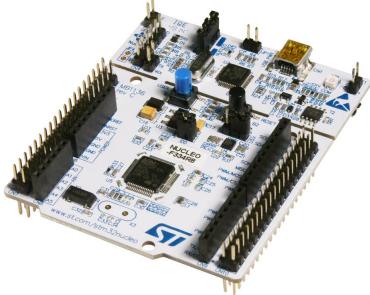
Tra queste la più popolare è Octo, un'implementazione scritta in JavaScript capace di eseguire la versione base di CHIP-8, S-CHIP e XO-CHIP nel browser. La repository è mantenuta da John Ernest, l'inventore di XO-CHIP che nel 2014 ha riportato in vita CHIP-8 modernizzandolo e aggiungendo nuove funzionalità.

Inoltre ogni anno viene organizzata la Octojam, una game jam dove ogni partecipante prova a sviluppare un videogioco per CHIP-8 (o per le sue estensioni) partendo da zero.

Grazie al suo instruction set ridotto e alla sua limitata richiesta di risorse hardware è stato portato su un elevato numero di piattaforme, tra cui il Game Boy Color, calcolatrici grafiche serie HP 48 e Emacs (il famoso editor di testo).

Sebbene CHIP-8 e S-CHIP siano stati tradizionalmente implementati tramite software esistono anche implementazioni hardware. Ne citiamo una in particolare scritta nel linguaggio Verilog per schede FPGA.

3 Hardware



(a) Il microcontrollore **STM32F334R8T6**.

C10	C11	C9	C8
C12	D2	B8	C6
VDD	E5V	B9	C5
BT0	GND	AVD	U5V
NC	NC	GND	D8
NC	IOR	A5	A12
A13	RST	A6	A11
A14	+3V	A7	B11
A15	+5V	B6	B11
GND	GND	C7	GND
B7	GND	A9	B2
C13	VIN	A8	B1
C14	NC	B10	B15
C15	A0	B4	B14
H0	A1	B5	B13
H1	A4	B3	AGN
LCD	B0	A10	C4
C2	C1	A2	NC
C3	C0	A3	NC

(b) Pinout del microcontrollore **STM32F334R8T6**.

Il componente hardware principale è il microcontrollore **STM32F334R8T6** (Fig. 1a), equipaggiato con un processore ARM Cortex-M4 da 72 MHz, 16 Kb di SRAM e 64 Kb di memoria flash. Abbiamo deciso di utilizzare questa scheda anziché quella fornita ci durante il corso (**STM32L053R8T6**) perché quest'ultima offre una quantità di SRAM troppo limitata (solo 8 Kb).

In particolare le ROM dei giochi CHIP-8 possono arrivare a pesare fino a 3.5 Kb per questo motivo è stato necessario utilizzare una scheda leggermente più potente.



(a) Lo schermo **ILI9341**.

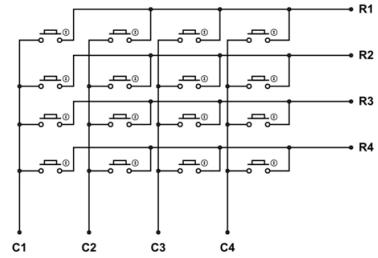
SD_SCK	
SD_DO	
SD_DI	
SD_SS	3.3V
LCD_D1	5V
LCD_D0	GND
LCD_D7	LCD_RD
LCD_D6	LCD_WR
LCD_D4	LCD_RS
LCD_D5	LCD_CS
LCD_D3	LCD_RST
LCD_D2	F_CS

(b) Pinout dello schermo **ILI9341**.

Per quanto riguarda lo schermo dell'emulatore abbiamo optato per un display TFT LCD a colori retroilluminato (Fig. 2a) basato sul controller **ILI9341**. Il display ha una dimensione di 2.4 pollici, una risoluzione di 320×240 px e al suo interno è presente un lettore di schede microSD integrato.



(a) Il tastierino matriciale 4×4 .



(b) Struttura del tastierino matriciale 4×4 .

Per poter interagire con l'emulatore abbiamo utilizzato un tastierino matriciale 4×4 (Fig. 3a) analogo alla tastiera originale del COSMAC VIP [2], un microcomputer creato da Joseph Weisbecker appositamente per CHIP-8.

Infine, per supportare gli effetti sonori riprodotti dai videogiochi CHIP-8 abbiamo utilizzato un beeper passivo monotono, che anche essendo un dispositivo molto semplice adempie correttamente ai suoi compiti dato che il suono prodotto da un emulatore CHIP-8 deve avere solo un tono.

3.1 Componenti utilizzati e relativi costi

Descrizione	Modello	Costo unitario	Unità	Costo
Microcontrollore	STM32 F334R8T6	14.99	1	14.99
Schermo	ILI9341 2.4"	6.50	1	6.50
Tastierino	Matrix keypad 4×4	3.99	1	3.99
Beeper		0.99	1	0.99
Cablaggio		4.99	1	4.99
Interruttore e altri materiali		4.99	1	4.99
Scocca	GW42002	9.99	1	9.99
Totale				46.50€

Table 1: Materiali utilizzati per la realizzazione del progetto.

In Tabella 1 è possibile vedere i componenti utilizzati per la realizzazione del progetto. I costi indicati provengono da negozi online come Amazon e eBay.

3.2 Schema di collegamento

Legenda dei colori in Figura 4

- **Verde:** Schermo
- **Blu:** Scheda microSD
- **Magenta:** Tastierino
- **Arancione:** Beeper
- **Ciano:** Reset
- Nero: GND
- **Rosso:** Alimentazione

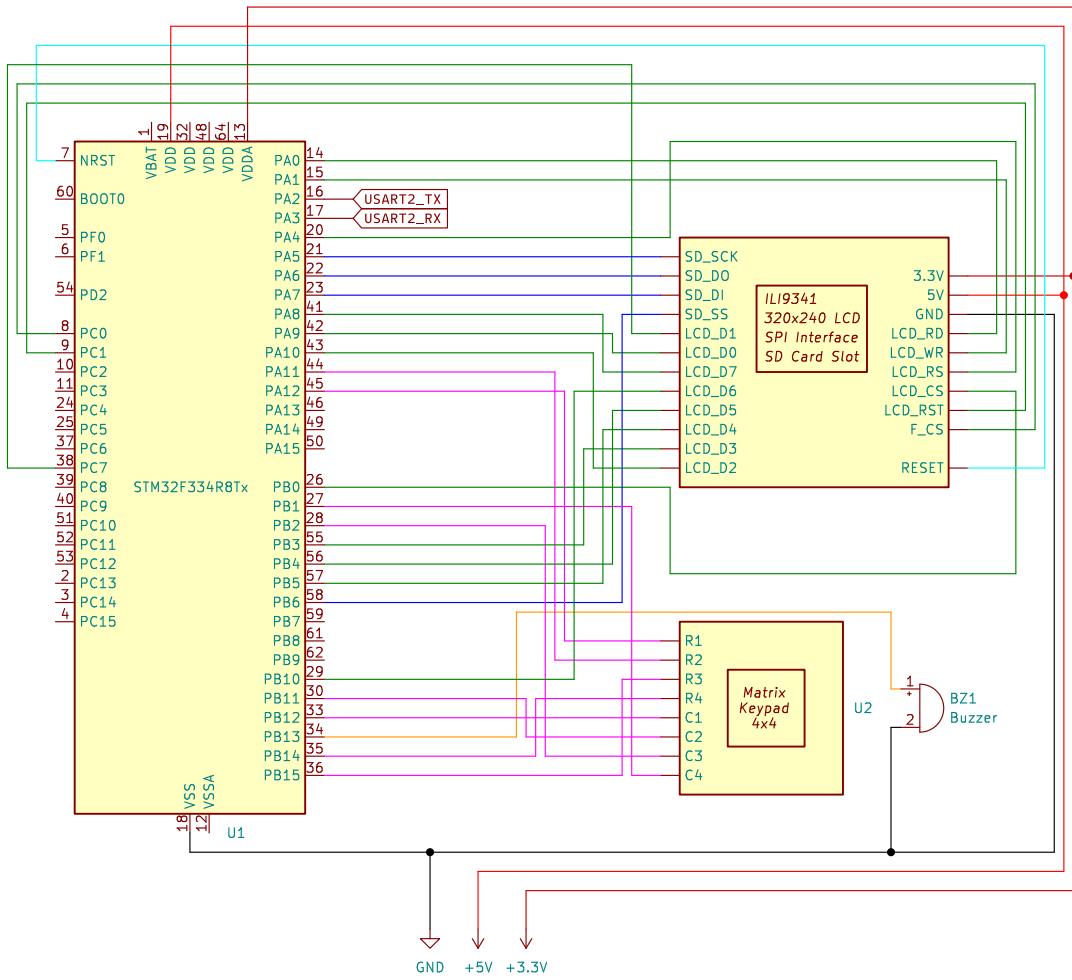


Figure 4: Schema dei collegamenti.

In Figura 4 è possibile vedere lo schema di collegamento dei componenti hardware realizzato utilizzando il software KiCad. Il display e il lettore microSD integrato sono stati collegati al microcontrollore attraverso il pinout standard degli Shields di Arduino, un'interfaccia hardware che permette di collegare una scheda Arduino ad un modulo esterno.

Il tastierino matriciale 4×4 è stato collegato al microcontrollore tramite 8 pin GPIO. In particolare i 4 pin relativi alle righe (R1, R2, R3, R4) sono stati impostati in modalità `GPIO_MODE_IT_RISING` (interrupt rising edge).

Quando un pin GPIO viene configurato come sorgente di interrupt sul fronte di salita il segnale di interrupt è generato quando il pin passa da basso (0) ad alto (1), risultando particolarmente utile per intercettare il cambiamento dello stato di un pulsante quando viene premuto.

Invece, i 4 pin relativi alle colonne (C1, C2, C3, C4) sono stati impostati in modalità `GPIO_MODE_OUTPUT_PP` (push-pull output) per consentire l'invio dell'interrupt alla pressione di un tasto, dato che chiudendo il circuito si permette alla corrente proveniente dal pin GPIO di fluire verso i pin di interrupt. Successivamente viene identificato il tasto premuto, questo meccanismo verrà elaborato nella sezione 4.2.5.

Infine, il beeper è stato collegato al microcontrollore tramite un pin GPIO e GND. Quest'ultimo è stato impostato in modalità `GPIO_MODE_OUTPUT_PP` (push-pull output) per permettere l'invio di un segnale al beeper.

4 Software

Il nostro software si divide in due componenti principali: l'interprete CHIP-8 e l'infrastruttura necessaria per "portarlo" sul microcontrollore STM32, ovvero l'interfaccia con lo schermo e i gestori per la scheda microSD, per il keypad e per il beeper.

4.1 Interprete CHIP-8

```
1 typedef struct {
2     uint8_t RAM[4096];
3     uint16_t I;           // Index register
4     uint16_t PC;          // Program counter
5     uint16_t stack[16];
6     uint8_t SP;          // Stack pointer
7     uint8_t V[16];        // Variable registers
8     uint8_t screen[1024];   // 128x64 pixels = 8192 bits = 1024 bytes
9     uint8_t keypad[16];
10    uint8_t DT;          // Delay timer
11    uint8_t ST;          // Sound timer
12    uint16_t opcode;      // Current opcode
13 } Chip8;
```

Listing 1: La struct che rappresenta lo stato della macchina virtuale.

Abbiamo deciso di scrivere l'interprete da zero e per farlo è stato necessario consultare le specifiche (de facto standard) che definiscono il comportamento di un interprete CHIP-8 [3] e S-CHIP [10].

L'interprete ha un'architettura basata su registri e possiede 4 Kb di memoria, 16 registri general purpose, un registro per gli indirizzi di memoria, un registro per il delay timer, un registro per il sound timer, uno stack per gestire le chiamate a subroutine, uno stack pointer e un program counter.

Il delay timer viene utilizzato come cronometro mentre il sound timer è utilizzato per gestire gli effetti sonori, quando il suo valore è diverso da zero, l'emulatore attiva il beeper.

Ad ogni ciclo di esecuzione l'interprete effettua il fetch dell'istruzione puntata dal program counter in memoria, la decodifica e la esegue.

Sono supportate 45 istruzioni diverse, ciascuna delle quali è rappresentata da uno specifico opcode in cui al suo interno sono passati anche eventuali parametri.

Il programma è scritto in C99, non ha I/O ed è freestanding [6], ovvero non dipende dalla libreria standard del C (libc). Tutto questo è mirato a rendere l'interprete altamente portabile.

Per rimuovere la dipendenza da libc è stato necessario includere alcune funzioni direttamente da libgcc, trovare un modo alternativo per implementare le asserzioni e includere una funzione ad hoc per la generazione di numeri casuali.

Le asserzioni sono gestite tramite una macro che verifica una condizione. Se la condizione è falsa, viene dereferenziato un puntatore a NULL, terminando così il programma. Si è utilizzato `volatile` per evitare possibili ottimizzazioni del compilatore. Invece per quanto riguarda la generazione di numeri casuali è stato adottato un LCG troncato [7], un generatore di numeri pseudo-casuali molto comune.

```
#define ASSERT(expr)           |
    if (!(expr)) {           |
        *(volatile int *) 0 = 0; |
    }
```

```
static uint8_t rand_byte(uint64_t *s) {
    *s = *s * 0x3243f6a8885a308d + 1;
    return *s >> 56;
}
```

Listing 2: Implementazioni di `ASSERT` e `rand_byte`.

Infine per testare più comodamente l'interprete abbiamo sviluppato un semplice emulatore su desktop utilizzando SDL2 [9], una libreria scritta in C che consente di gestire audio, video e input da tastiera. In seguito l'interprete è stato sottoposto ad un'apposita test suite [1] che mira a verificare il comportamento corretto di ciascun opcode.

4.1.1 Gestione del timing

Uno dei problemi principali durante lo sviluppo di un emulatore è la gestione del timing, in particolare è necessario limitare la "velocità" dell'emulatore bloccando temporaneamente la sua esecuzione.

Inoltre abbiamo dovuto disaccoppiare la frequenza dell'interprete (regolabile dal giocatore) dalla frequenza del delay timer e del sound timer (costante a 60 Hz). Dove con frequenza dell'interprete ci riferiamo al numero di istruzioni che esegue ogni frame.

Inizialmente abbiamo optato per la gestione di una singola istruzione per ciclo di esecuzione, di conseguenza il ritardo del game loop risultava variabile e dipendeva dalla frequenza selezionata dal giocatore. Per assicurare una frequenza \mathcal{F} di 60 Hz i timer venivano decrementati ogni n -esima iterazione del game loop, dove $n = \frac{\mathcal{F}}{60}$. Ad esempio se $\mathcal{F} = 540$, i timer venivano decrementati ogni 9° ciclo.

Purtroppo però questo approccio presenta un problema non trascurabile, ovvero effettua una chiamata ad una funzione simil-sleep per un periodo molto breve dopo ogni istruzione. Ad esempio se $\mathcal{F} = 540$, il ritardo di una sleep sarebbe solo di 1.85 ms, e questo genere di funzione non offre una precisione simile. Per questo motivo abbiamo optato per una soluzione differente.

Abbiamo fissato il ritardo del game loop a 16.666 ms, un valore sufficientemente alto da non avere problemi di granularità. Inoltre in questo modo otteniamo un frame rate di 60 FPS esatti. Avendo reso il ritardo costante abbiamo dovuto rendere variabile il numero di istruzioni gestite durante un ciclo di esecuzione. In particolare vengono gestite n istruzioni per ciclo, dove $n = \frac{\mathcal{F}}{60}$. Ad esempio se $\mathcal{F} = 540$, vengono gestite 9 istruzioni per ciclo. A questo punto dato che il game loop viene ripetuto con una frequenza di 60 Hz risulta banale gestire la frequenza dei timer.

Sono state considerate anche eventuali problematiche che sarebbero potute sorgere con questo approccio. In particolare non tutte le istruzioni impiegano lo stesso tempo per essere eseguite, ma fortunatamente anche l'istruzione più lenta richiede una quantità trascurabile di tempo. Ciò significa che possiamo comportarci come se tutte le istruzioni richiedessero il medesimo tempo.

4.1.2 Ottimizzazioni

È stato necessario introdurre delle ottimizzazioni all'interno dell'interprete per poterlo far girare su un microcontrollore.

L'ottimizzazione principale è legata alla rappresentazione dello schermo in memoria. Ad alto livello lo schermo può essere visto come una matrice di 128x64 pixel monocromi. Una rappresentazione simile occuperebbe 8192 byte, dato che ciascun pixel verrebbe rappresentato da un byte.

Purtroppo il nostro microcontrollore ha a disposizione solamente 16 Kb di SRAM, di conseguenza una soluzione simile non è praticabile.

Per questo motivo abbiamo deciso di rappresentare lo schermo come un array unidimensionale di 1024 byte, dove ciascun pixel viene rappresentato da un singolo bit. In questo modo otteniamo un risparmio di spazio pari a ben l'87.5%.

Questa decisione ha aggiunto però un livello di indirezione dato che una coordinata ad alto livello sulla matrice 128x64 deve essere mappata ad una coordinata "in memoria".

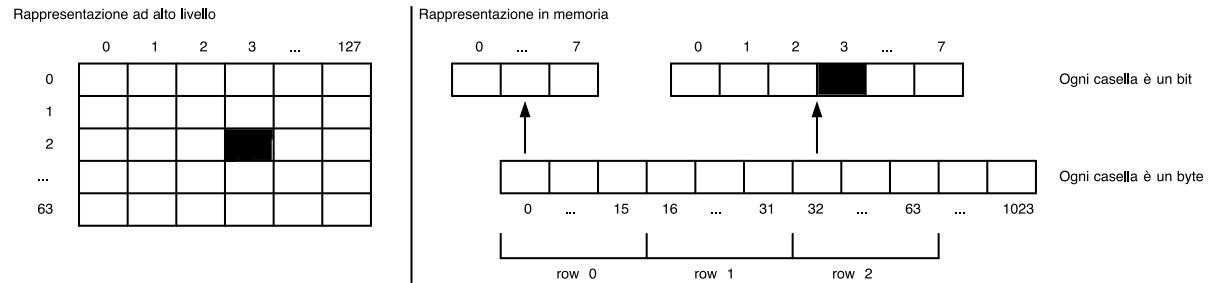


Figure 5: Esempio della mappatura di un pixel.

Un’ulteriore ottimizzazione viene resa disponibile attraverso l’API dell’interprete sotto forma di una funzione che consente al chiamante di controllare se l’array che rappresenta lo schermo è stato modificato nell’ultimo ciclo di esecuzione. In questo modo la grafica viene renderizzata dal chiamante solo quando è effettivamente necessario.

4.1.3 Comportamenti ambigui

Gli interpreti CHIP-8 e S-CHIP hanno sviluppato molteplici comportamenti ambigui nel corso degli anni. Questi cosiddetti "quirk" variano in base alle piattaforme per cui è stato sviluppato l’interprete. Ad esempio gli interpreti per calcolatrici HP 48 presentano un comportamento leggermente diverso durante l’esecuzione delle istruzioni di SHIFT.

Questi comportamenti ambigui si propagano fino ai programmati CHIP-8 che si appoggiano a quest’ultimi e scrivono videogiochi che non sono del tutto compatibili con interpreti più vecchi. Per evitare questa frammentazione è necessario supportare le piattaforme principali e i loro quirk.

Il nostro interprete supporta CHIP-8, CHIP-48, S-CHIP 1.0 e S-CHIP 1.1, in questo modo è in grado di eseguire la stragrande maggioranza dei videogiochi reperibili in rete.

4.2 Porting su STM32

Dopo aver terminato l’interprete abbiamo sviluppato un’infrastruttura per poterlo eseguire sul microcontrollore STM32.

Come già anticipato, l’infrastruttura si compone di un’interfaccia con lo schermo, un gestore per la scheda microSD, un gestore per il tastierino e un gestore per il beeper.

È importante ricordare che l’interprete è stato progettato per essere altamente portabile e indipendente dalla piattaforma su cui verrà eseguito, per questo motivo non è stato necessario apportare modifiche significative.

Tuttavia, a causa della potenza limitata della scheda la fluidità del gameplay è inferiore rispetto alla port su SDL2 che viene eseguita su normale computer.

4.2.1 Architettura software

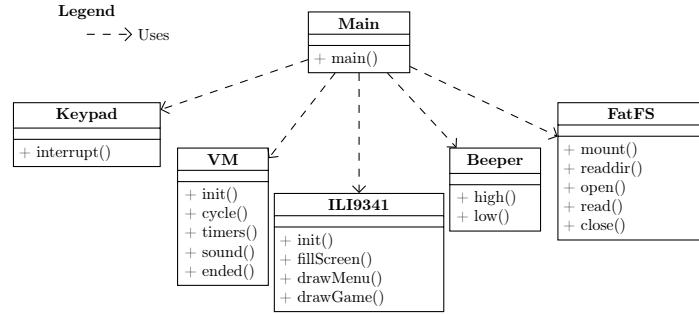


Figure 6: Class diagram.

In Figura è rappresentato il class diagram, il quale offre una panoramica dell’architettura del software. Una differenza da evidenziare rispetto alla proposta iniziale è che la classe `menù` non è più presente perché è stata integrata all’interno della classe `main`.

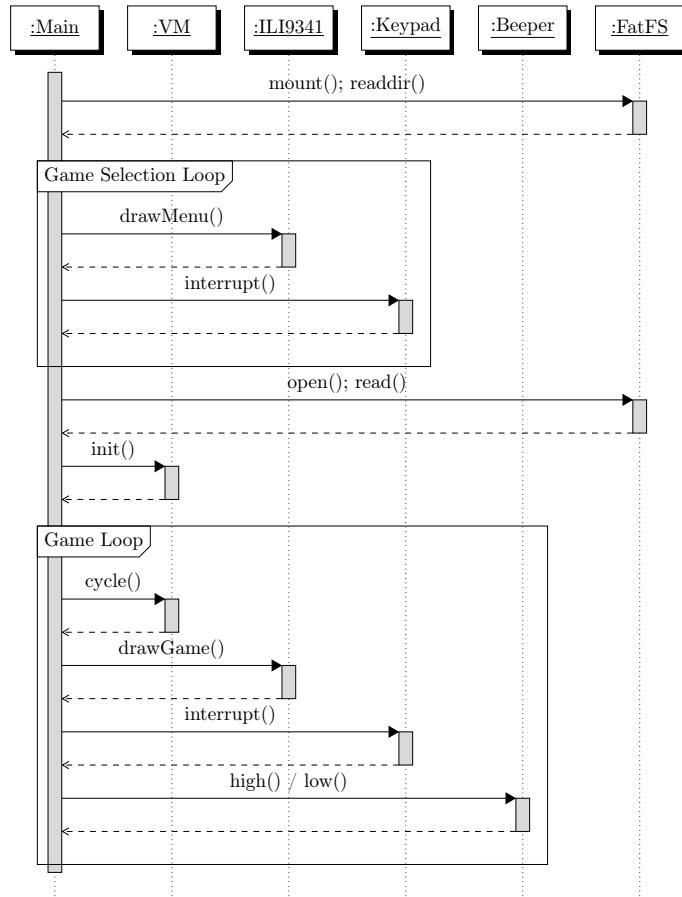


Figure 7: Sequence diagram.

Il flusso di esecuzione dell’emulatore è mostrato in Figura 7, per prima cosa vengono letti i nomi dei file presenti sulla scheda microSD, in seguito si entra nel game selection loop dove il giocatore potrà selezionare un gioco, la velocità dell’interprete e la piattaforma da emulare. Una volta selezionato il gioco si entrerà nel game loop dove inizia la fase di emulazione vera e propria.

4.2.2 Interfaccia con la scheda microSD

La scheda microSD è utilizzata come memoria di massa dell'emulatore contenente i file binari dei video-giochi CHIP-8. È stata formattata con FAT32 e viene gestita utilizzando FatFs [4], una libreria open source che permette di interfacciarsi con questa tipologia di filesystem.

La comunicazione tra il microcontrollore e la scheda microSD avviene tramite il protocollo Serial Peripheral Interface (SPI), dove il microcontrollore è configurato come master e la scheda microSD come slave.

4.2.3 Menù di selezione

La selezione del gioco avviene tramite un apposito menù che lista su più pagine tutti i giochi presenti sulla scheda microSD. Per aggiungere un gioco è sufficiente caricarlo sulla scheda microSD e riavviare il microcontrollore, poiché il menu viene generato dinamicamente. Inoltre, il menù permette anche di impostare la velocità e la modalità di esecuzione dell'interprete.

L'utente interagisce con il menù attraverso il tastierino, in particolare abbiamo predisposto dei tasti riservati per la navigazione tra le pagine, per la selezione della velocità e della modalità di esecuzione. I tasti rimanenti vengono utilizzati per avviare specifici giochi.

Infine, per risolvere il problema degli artefatti grafici al cambio di schermata, abbiamo implementato una funzione dedicata per "pulire" lo schermo e ottenere un effetto di transizione da una schermata all'altra.

4.2.4 Rendering del font

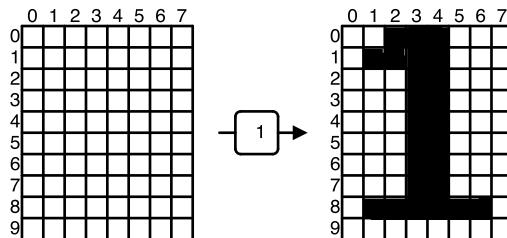


Figure 8: Esempio della mappatura di un carattere del font.

Per poter mostrare sullo schermo i nomi dei giochi caricati sulla scheda microSD è necessario l'utilizzo di un font. Abbiamo scelto un font che rappresenta ciascun carattere come una bitmap, ossia una matrice 8×10 pixel rappresentata da 10 interi ad 8 bit.

Nella Figura 8, è mostrata una rappresentazione grafica di questa matrice. Ogni riga è un intero a 8 bit, dove i bit a 1 corrispondono ai pixel neri. Ad esempio, la prima riga è rappresentata dal numero 56, che in binario è **00111000**.

4.2.5 Funzionamento del keypad

Inizialmente il keypad è stato gestito tramite polling, dove il microcontrollore verificava continuamente lo stato dei pin per identificare il tasto premuto. Tuttavia, questo approccio è stato abbandonato in favore di un approccio basato su interrupt. Il polling richiedeva molte risorse e non garantiva un framerate stabile, poiché non riusciva a leggere tutti i pin e completare le operazioni rimanenti entro il limite di 16 ms imposto dal frame rate dell'emulatore (60 FPS).

La procedura standard per gestire gli interrupt su microcontrollori STM32 coinvolge la configurazione di un pin GPIO come sorgente di interrupt e l'implementazione di una funzione per la gestione di un interrupt. Per poter generare un interrupt quando un tasto viene premuto abbiamo impostato i pin di riga in modalità `GPIO_MODE_IT_RISING` e i pin di colonna in modalità `GPIO_MODE_OUTPUT_PP`, come già accennato nella sezione 3.2.

4.2.6 Interfaccia con lo schermo

La comunicazione con il controller ILI9341 può avvenire tramite due interfacce diverse: SPI oppure parallela ad 8 bit. L'interfaccia SPI richiede meno pin ma è più lenta a causa del suo protocollo seriale, mentre l'interfaccia parallela è più veloce ma richiede più pin. La decisione di quale interfaccia utilizzare viene presa direttamente dal produttore che espone solamente alcuni dei pin del circuito integrato. Poiché il display dovrà essere aggiornato 60 volte al secondo abbiamo deciso di acquistare la versione dello schermo che supporta l'interfaccia parallela.

Durante la fase di inizializzazione i pin di controllo della scheda (`LCD_RD`, `LCD_WR`, `LCD_RS`, `LCD_CS`, `LCD_RST`) vengono impostati in modalità `GPIO_MODE_OUTPUT_PP` tramite le funzioni messe a disposizione dall'HAL.

Inoltre, prima di inviare la sequenza di inizializzazione al display, vengono disabilitati i pin `LCD_WR` e `LCD_RD` e viene impostato il pin `LCD_CS` a `LOW` (il controller `ILI9341` è attivo basso).

Il pin `LCD_RST` permette di resettare lo stato interno del display, per questo motivo viene attivato e disattivato all'avvio in modo tale da impostare la scheda in uno stato conosciuto. Questo pin rimarrà disabilitato per l'intera durata dell'esecuzione.

Nel momento in cui la scheda raggiunge uno stato conosciuto bisogna inviare una sequenza di comandi per attivare il display, questa sequenza di inizializzazione è spiegata dettagliatamente nel datasheet dello schermo [11].

Infine, per ottimizzare le prestazioni, abbiamo deciso di modificare il modo in cui stampiamo sullo schermo. Inizialmente la funzione che gestisce la stampa a schermo si appoggiava all'HAL, questo purtroppo generava un overhead non trascurabile che rallentava molto l'emulatore. Per risolvere questo problema abbiamo deciso di non appoggiarci più all'HAL e di scrivere la funzione in bare metal.

5 Assemblaggio



Figure 9: Assemblaggio finale.

Una volta terminato lo sviluppo del software abbiamo deciso di inserire i componenti hardware all'interno di un guscio protettivo, in modo da nascondere il cablaggio e i vari pin scoperti, lasciando accessibili solamente le parti con cui il giocatore interagisce direttamente.

Il guscio protettivo è stato realizzato con una base di supporto polifunzionale [5] a cui sono state apportate le modifiche appropriate per essere in grado di "montare" correttamente il display e il tastierino. Inoltre sono stati effettuati due fori sulla scocca per poter sentire più facilmente il beeper e per poter accedere ad un interruttore per l'accensione e spegnimento della scheda.

I collegamenti sono rimasti identici a quelli descritti nella sezione 3.2, è stata però aggiunta una batteria esterna.

Infine, come ultimo tocco puramente estetico, abbiamo applicato degli adesivi al tastierino per ricoprire i tasti che non corrispondevano a quelli originali del COSMAC VIP.

Consideriamo il risultato finale accettabile come prototipo iniziale, ma se si volesse commercializzare un prodotto simile sarebbe imperativo utilizzare una PCB ad hoc più compatta, un guscio su misura e un tastierino di qualità superiore.

6 Analisi del consumo energetico

La scheda STM32F334R8T6 può essere alimentata sia tramite un cavo USB che mediante una batteria esterna; per rendere l'emulatore portatile abbiamo scelto quest'ultima soluzione utilizzando 4 batterie AA da 1.5 V ciascuna. Per passare dall'alimentazione USB a quella esterna abbiamo dovuto muovere il jumper JP5 dalla posizione 1-2 alla posizione 2-3 [12]. Successivamente, abbiamo collegato le batterie in serie e connesso il polo positivo al pin E5V e il polo negativo al pin GND della scheda.

Per ottenere una stima dell'autonomia dell'emulatore, abbiamo analizzato il suo consumo energetico utilizzando il software STM32CubeIDE. Il consumo energetico della scheda è di circa 30 mAh, mentre quello del display retroilluminato si aggira intorno ai 90 mAh. Quindi, complessivamente, l'emulatore consuma all'incirca 120 mAh.

Assumendo che le batterie abbiano una capacità di 2500 mAh, ci aspettiamo che l'emulatore possa rimanere acceso per circa 20 ore.

$$\text{tempo} = \frac{\text{capacità}}{\text{consumo}} = \frac{2500}{120} = 20$$

7 Considerazioni finali

Siamo riusciti a realizzare un emulatore CHIP-8 e S-CHIP in grado di funzionare su un microcontrollore STM32 raggiungendo così l'obiettivo che ci eravamo prefissati.

Alcuni videogiochi purtroppo non hanno un gameplay fluido a causa della potenza limitata della scheda, nonostante questo Tetris, Blitz, Tic-tac-toe e altri giochi analoghi hanno prestazioni più che accettabili.

Infine, per quanto riguarda possibili sviluppi futuri abbiamo ipotizzato che si potrebbe ulteriormente ottimizzare il software modificando l'interprete in modo tale da interagire direttamente con il display, perdendo però così la sua portabilità.

References

- [1] CHIP-8 test suite. <https://github.com/Timendus/chip8-test-suite>.
- [2] COSMAC VIP. https://en.wikipedia.org/wiki/COSMAC_VIP.
- [3] Cowgod's Chip-8 Technical Reference v1.0. <http://devernay.free.fr/hacks/chip8/C8TECH10.HTM>.
- [4] elm-chan.org - FatFs - Generic FAT Filesystem Module. http://elm-chan.org/fsw/ff/00index_e.html.
- [5] Gewiss - Base di supporto polifunzionale. <https://www.gewiss.com/al/it/prodotti/product.1000002.1000090.GW42002>.
- [6] ISO/IEC 9899:1999 - 4. Conformance. <https://port70.net/~nsz/c/c99/n1256.html#4p6>.
- [7] Linear congruential generator. https://en.wikipedia.org/wiki/Linear_congruential_generator.
- [8] Octo. <https://johnearnest.github.io/Octo>.
- [9] Simple DirectMedia Layer. <https://www.libsdl.org>.
- [10] SUPER-CHIP v1.1. <http://devernay.free.fr/hacks/chip8/schip.txt>.
- [11] Ilitek. Tft lcd single chip driver 240rgbx320 resolution and 262k color - specification. <https://cdn-shop.adafruit.com/datasheets/ILI9341.pdf>, 2011.
- [12] ST. Stm32 nucleo-64 boards - user manual. https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf, 2020.

A Che aspetto ha un programma CHIP-8?

I programmi CHIP-8 ricordano vagamente il linguaggio Assembly, ma le loro istruzioni non sono mappate 1:1 ad istruzioni macchina eseguibili da un processore reale. Invece, vengono mappate a bytecode interpretabile da una macchina virtuale (4.1).

Al giorno d'oggi la maggior parte dei programmatori CHIP-8 utilizza Octo [8], un IDE online scritto in JavaScript, che include un assemblatore per tradurre i programmi in bytecode e una macchina virtuale per eseguirli.

Di seguito è riportato un programma CHIP-8 d'esempio:

```
1 :alias px v1
2 :alias py v2
3
4 : main
5   px := random 0b0011111
6   py := random 0b0001111
7   i := person
8   sprite px py 8
9
10  loop
11    sprite px py 8
12    v0 := 5 if v0 key then py += -1
13    v0 := 8 if v0 key then py += 1
14    v0 := 7 if v0 key then px += -1
15    v0 := 9 if v0 key then px += 1
16    sprite px py 8
17
18  loop
19    vf := delay
20    if vf != 0 then
21      again
22      vf := 3
23      delay := vf
24    again
25
26 : person
27   0x70 0x70 0x20 0x70 0xA8 0x20 0x50 0x50
```