

**MSc in Computer Science**  
at University of Milan

**CHIP-8 STM32**  
Proposta per il Progetto di PROS,  
corso tenuto da **Danilo Bruschi**

Email:  
[federico.bruzzone@studenti.unimi.it](mailto:federico.bruzzone@studenti.unimi.it)  
[lorenzo.ferrante1@studenti.unimi.it](mailto:lorenzo.ferrante1@studenti.unimi.it)  
[andrea.longoni3@studenti.unimi.it](mailto:andrea.longoni3@studenti.unimi.it)

Creato da:  
**Federico Bruzzone**  
**Lorenzo Ferrante**  
**Andrea Longoni**

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Stato dell'arte</b>	<b>3</b>
<b>3</b>	<b>Hardware</b>	<b>4</b>
3.1	Schema di collegamento . . . . .	5
3.1.1	Descrizione del collegamento dei componenti . . . . .	5
3.2	Componenti utilizzati e relativi costi . . . . .	6
<b>4</b>	<b>Software</b>	<b>7</b>
4.1	Interprete/Emulatore 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 . . . . .	10
4.2.1	Architettura software . . . . .	10
4.2.2	Interfaccia con la scheda microSD . . . . .	10
4.2.3	Menu di selezione . . . . .	12
4.2.4	Funzionamento del keypad . . . . .	12
4.2.5	Interfaccia con lo schermo . . . . .	13
4.2.6	Rendering del font . . . . .	14
<b>5</b>	<b>Assemblaggio</b>	<b>15</b>
<b>6</b>	<b>Analisi del consumo energetico</b>	<b>16</b>
<b>7</b>	<b>Considerazioni finali e sviluppi futuri</b>	<b>16</b>
	<b>Riferimenti bibliografici</b>	<b>17</b>

## List of Figures

4	Schematic del progetto. . . . .	6
5	Esempio della mappatura di un pixel. . . . .	9
6	Class diagram dell'architettura software. . . . .	10
7	Sequence diagram dell'architettura software. . . . .	11

8	Esempio della mappatura di un carattere del font . . . . .	15
9	Assemblaggio finale. . . . .	16

## List of Tables

1	Materiali utilizzati per la costruzione del progetto. . . . .	6
2	Tempi previsti ed effettivi per la realizzazione dei componenti software . . . . .	17

*Per divulgazione abbiamo deciso di rendere pubblico il nostro progetto. Questo documento è stato scritto con l'intento di essere comprensibile a chiunque abbia una conoscenza di sistemi operativi e sistemi embedded. Inoltre, abbiamo deciso creare un'organizzazione GitHub chiamata **Chip-8-Org**<sup>a</sup> per rendere pubblico il codice sorgente della virtual machine<sup>b</sup>, il porting su STM32 con il codice relativo alle interfacce hardware<sup>c</sup> e i latex relativi a questa relazione e alla proposta<sup>d</sup>. L'obiettivo di questa azione è quello di promuovere la conoscenza di CHIP-8 e S-CHIP e di rendere disponibile un progetto open-source per chiunque sia interessato a sviluppare un emulatore per CHIP-8 o S-CHIP.*

*GitHub:*

- *Organizzazione: <https://github.com/CHIP-8-Org>*
- *Federico Bruzzone: <https://github.com/FedericoBruzzone>*
- *Lorenzo Ferrante: <https://github.com/0xHaru>*
- *Andrea Longoni: <https://github.com/Andreal2000>*

<sup>a</sup><https://github.com/CHIP-8-Org>

<sup>b</sup><https://github.com/CHIP-8-Org/Core>

<sup>c</sup><https://github.com/CHIP-8-Org/CHIP-8-STM32>

<sup>d</sup><https://github.com/CHIP-8-Org/Latex>

## 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 costruire 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 puo' 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 Earnest, 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).

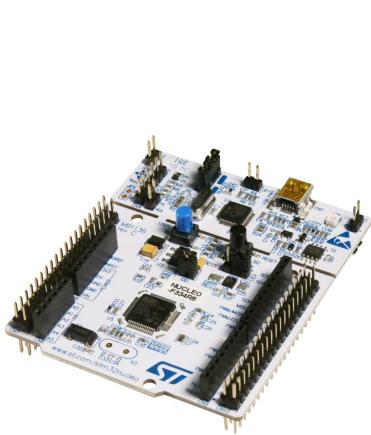
Sebbene CHIP-8 e S-CHIP siano stati tradizionalmente implementati tramite software esistono anche

implementazioni hardware. Ne citiamo una in particolare scritta nel linguaggio Verilog per schede FPGA.

### 3 Hardware

Le componenti principali utilizzate per la relizzazione del progetto sono 5: una scheda ST **STM32F334R8T6** (Fig. 1a), uno schermo TFT LCD **ILI9341** (Fig. 2a), e un lettore di schede microSD integrato nello schermo, un tastierino matriciale  $4 \times 4$  e un beeper.

Il componente principale é il microcontrollore **STM32F334R8T6** basato su architettura ARM con processore Cortex-M4 da 72 MHz, 64 Kb di memoria flash e 16 Kb di SRAM. Abbiamo deciso di utilizzare questa scheda perché le ROM dei giochi CHIP-8 e S-CHIP hanno dimensione massima di 4 Kb. Considerando questo e il fatto che la macchina virtuale necessita 4 Kb per poter funzionare non è stato potuto utilizzare la scheda **STM32L053R8T6** fornita ci durante il corso a causa della sua quantità limita di SRAM, precisamente 8 Kb.



(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 secondo componente che abbiamo utilizzato é un display TFT LCD (thin-film-transistor liquid-crystal display) a colori retroilluminato (Fig. 2a), per poterci interfacciare con l'emulatore. Questo display, da 2.4 pollici, é basato sul controller **ILI9341** e ha una risoluzione di  $320 \times 240$  px.



(a) Lo schermo **ILI9341**.

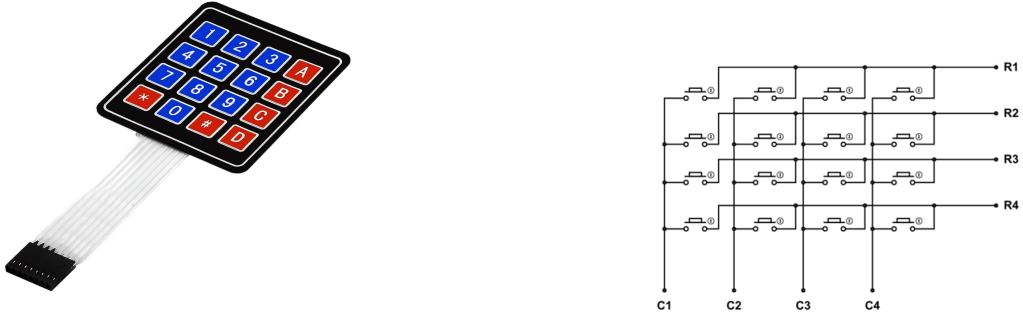
SD_SCK	
SD_DO	
SD_DI	
SD_SS	3.3V
LCD_D1	5V
LCD_D0	GND
LCD_D7	LCD_RD
LCD_D6	LCD_WR
LCD_D4	LCD_RS
LCD_D5	LCD_CS
LCD_D3	LCD_RST
LCD_D2	F_CS

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

Il terzo componente é un lettore di schede microSD integrato nello schermo (Fig. 2a). Questo componente é basato sul controller **ILI9341** che permette di interfacciarsi con una microSD formattata in FAT32, sulla quale é possibile salvare file di dimensione massima 4 Gb e per un totale di 2 Tb di dati.

Per interagire con l'emulatore abbamo utilizzato una tastiera matriciale  $4 \times 4$  corrispondente alla tastiera

esadecimale originale del CHIP-8 (Fig. 3a).



(a) Il tastierino matriciale  $4 \times 4$ .

(b) Struttura del tastierino matriciale  $4 \times 4$ .

Come ultimo componenete, per riprodurre gli effetti sonori generati dal gioco un beeper passivo monotono é stato collegato al microcontrollore tramite un GPIO output e GND. In CHIP-8 e S-CHIP vi é un solo frequenza che può essere riprodotta durante tutta l'esecuzione del gioco.

Inoltre, abbiamo deciso di aggiungere uno slot per l'alimentazione tramite due batterie AA.

### 3.1 Schema di collegamento

#### Legenda dei colori in Figura 4

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

#### 3.1.1 Descrizione del collegamento dei componenti

In Figura 4 é possibile vedere lo schema di collegamento delle componenti utilizzate per la realizzazione del progetto. Per realizzare lo schema é stato utilizzato il software KiCad. E' importante notare, che lo schermo **ILI9341** e l'integrato lettore di schede microSD, utilizza lo pinout standard degli Shields di Arduino, ovvero un'interfaccia hardware che permette di collegare una scheda Arduino ad un modulo esterno. Quindi, é stato collegato al nostro microcontrollore STM32 utilizzando lo stesso pinout.

Il tastierino matriciale  $4 \times 4$  é 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 si configura un pin GPIO come sorgente di interrupt su un fronte di salita, significa che l'interrupt verrà generato quando il livello logico del pin passa da basso (0) a alto (1). Questo è utile, ad esempio, quando si desidera intercettare un cambiamento di stato su un pulsante quando viene premuto. Invece, i 4 pin relativi alle colonne (**C1**, **C2**, **C3**, **C4**) sono stati impostati in modalità

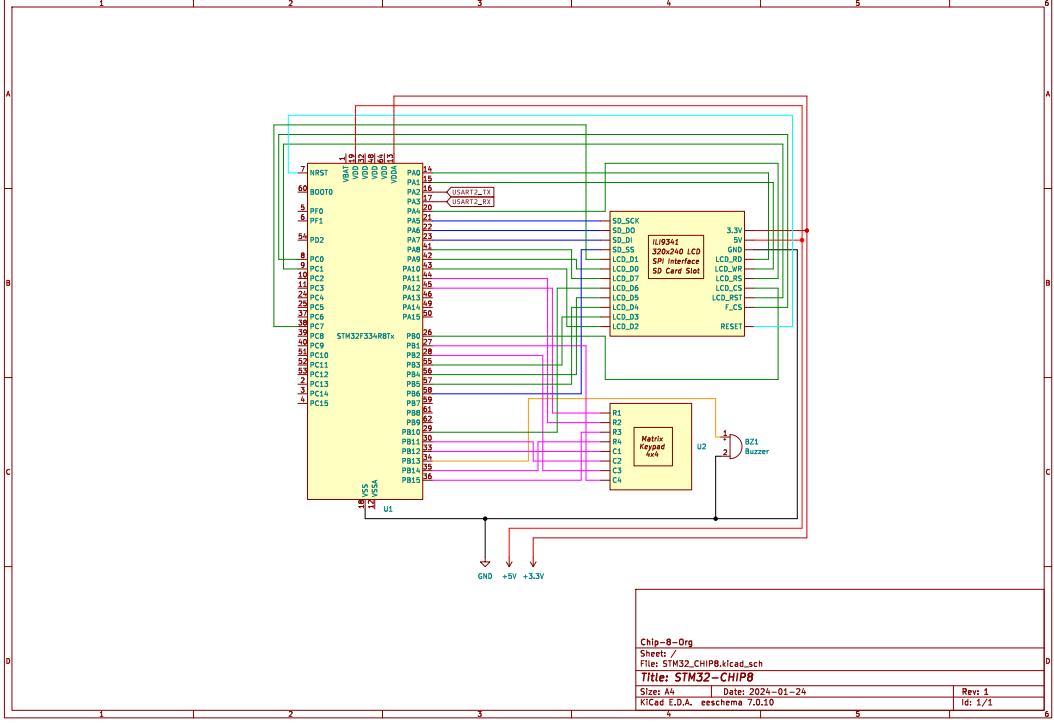


Figure 4: Schematic del progetto.

`GPIO_MODE_OUTPUT_PP` (push-pull output) per permettere l'invio dell'interrupt alla pressione di un tasto, dato che chiudendo il circuito permettiamo alla corrente proveniente dal pin GPIO di fluire verso i pin di interrupt. Successivamente, viene identificato il tasto premuto come verra' spiegato nella sezione 4.2.4.

L'ultimo componente é il beeper, che é stato collegato al microcontrollore tramite un pin GPIO e GND. Il pin GPIO é stato impostato in modalitá `GPIO_MODE_OUTPUT_PP` (push-pull output) per permettere l'invio di un segnale al beeper.

### 3.2 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 4x4	3.99	1	3.99
Beeper		0.99	1	0.99
Cablaggio		4.99	1	4.99
Bottone e altri materiali		4.99	1	4.99
Scocca	GW42002	9.99	1	9.99
<b>Totalle</b>				<b>46.50€</b>

Table 1: Materiali utilizzati per la costruzione del progetto.

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

## 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 come già anticipato nella sezione 3.

### 4.1 Interprete/Emulatore CHIP-8

Un emulatore è un software progettato per replicare il funzionamento di un processore e di altre componenti di un sistema informatico, consentendo così l'esecuzione di programmi scritti per un'architettura specifica su un'altra architettura. Lo sviluppo di un emulatore richiede una dettagliata comprensione dell'architettura da emulare, il che è al di là dello scopo di questo corso.

Nonostante questo, 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 [2] e S-CHIP [6].

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, come si può vedere nel listato 1.

```
1 #define RAM_SIZE 4096
2 #define SCREEN_SIZE 1024           // 128x64 pixels = 8192 bits = 1024 bytes
3 #define KEYPAD_SIZE 16
4
5 typedef struct {
6     uint8_t RAM[RAM_SIZE];
7     uint16_t I;                  // Index register
8     uint16_t PC;                // Program counter
9     uint16_t stack[16];
10    uint8_t SP;                 // Stack pointer
11    uint8_t V[16];              // Variable registers
12    uint8_t hp48_flags[8];       // HP-48's "RPL user flag" registers (S-CHIP)
13    uint8_t screen[SCREEN_SIZE];
14    uint8_t keypad[KEYPAD_SIZE];
15    uint8_t wait_for_key;
16    uint8_t DT;                 // Delay timer
17    uint8_t ST;                 // Sound timer
18    uint16_t opcode;            // Current opcode
19    uint64_t rng;               // PRNG state
20    int IPF;                  // No. instructions executed each frame
21    bool hi_res;               // Enable 128x64 hi-res mode (S-CHIP)
22    bool screen_updated;        // Was the screen updated?
23    Platform platform;         // CHIP-8, CHIP-48/S-CHIP 1.0 or S-CHIP 1.1 behavior?
24 } Chip8;
```

Listing 1: Struttura dell'emulatore Chip8

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 [4], 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, in particolare abbiamo re-implementato le funzioni `memset` e `memcpy`. Inoltre, abbiamo trovato un modo alternativo per implementare le asserzioni e includere una funzione ad hoc per la generazione di numeri pseudo-casuali come si puo' vedere nel listato 2.

#### TODO SPIEGARE IL CODICE

```
#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 [5], 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.

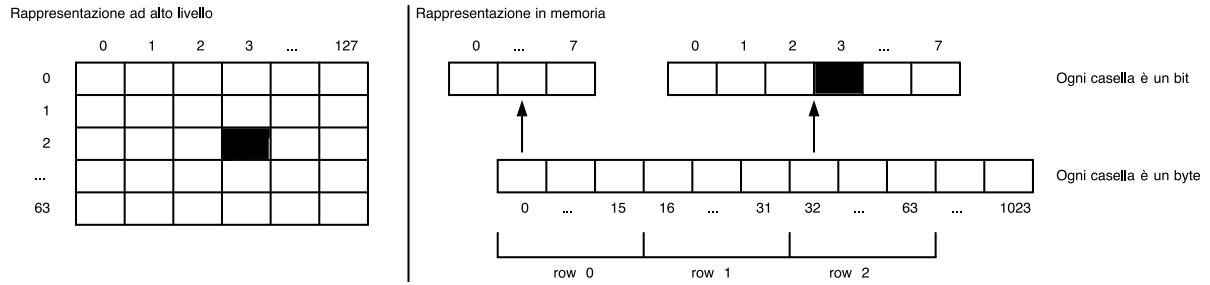


Figure 5: Esempio della mappatura di un pixel.

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 è mappata ad una coordinata "in memoria", dove la prima componente è l'indice del byte nell'array unidimensionale e la seconda componenete è il bit all'interno del byte. La figura [5] mostra un esempio di mappatura di un pixel. Più precisamente la funzione  $F$  è definita come segue:

$$F : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$$

$$F(x, y) = \left( \left\lfloor \frac{128y + x}{8} \right\rfloor, 7 - (x \bmod 8) \right)$$

where  $x \in [0, 127]$   $y \in [0, 63]$

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. Avendo notato che il numero di opcode che modificano lo schermo è molto limitato, precisamente sono solo 7 su 45, questa funzione consente di ridurre di 6.42 volte il numero di volte in cui lo schermo viene aggiornato.

#### 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 HP48 presentano un comportamento leggermente diverso durante l'esecuzione delle istruzioni di SHIFT.

Questi comportamenti ambigui si propagano fino ai programmatori 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 sviluppato l'emulatore, abbiamo sviluppato un'infrastruttura per poterlo eseguire sul microcontrollore STM32. Come già anticipato nella sezione 3, 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.

E' utile ricordare che l'emulatore che abbiamo sviluppato è stato progettato per essere eseguito su qualunque architettura hardware in per cui è compilabile un file C, quindi non è stato necessario apportare modifiche all'interprete per poterlo eseguire sul microcontrollore STM32.

Quindi, il processo di adattamento di questo emulatore per il microcontrollore è stato relativamente semplice e non ha richiesto modifiche significative. Tuttavia, in quanto l'emulazione avviene su una architettura con potenza di calcolo limitata rispetto a quella di un comune calcolatore, le prestazioni e la fluidità dell'emulatore sono inferiori rispetto a quelle che avevamo usando SDL2 [5].

Normalmente, software sviluppati per calcolatori odierni non richiedono di compilare il codice in *release* per notare una sostanziale differenza dalla comune compilazione, per esempio quella in *debug*. Mentre, per quanto concerne il microcontrollore, la compilazione in *release* permette di ottenere un incremento di prestazioni significativo anche se in alcune situazioni è stato necessario eseguire in modalità *debug* per poter usufruire delle feature del *debugger*, in particolare *breakpoint* e *watchpoint*.

### 4.2.1 Architettura software

In Figura 6 è possibile vedere il “class” diagram riferito al porting dell'emulatore su STM32. L'architettura è composta da 5 componenti principali: Keypad, Screen, SD, Keypad e Beeper poi utilizzati in figura 7 per mostrare come questi interagiscono tra di loro.

Il `menu`, a differenza dello studio di fattibilità, è stato rimosso dalle figure dato che tutte le operazioni che lo coinvolgono sono state spostate all'interno del `main`.

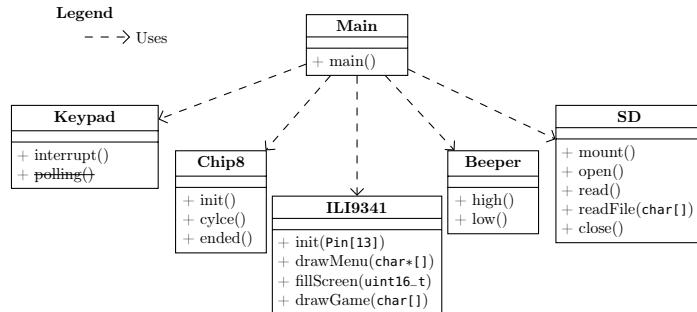


Figure 6: Class diagram dell'architettura software.

Il flusso di esecuzione dell'emulatore è mostrato in Figura 7, come si puo' notare e' diviso in due parti. In particolare, la prima inizia con la lettura dei file presenti sulla scheda microSD, successivamente il giocatore seleziona il gioco da eseguire e altre informazioni come la velocità di esecuzione e la modalità di esecuzione. Successivamente, l'emulatore inizia l'esecuzione del gioco selezionato. Mentre la seconda parte mostra il flusso di esecuzione dell'emulatore durante l'esecuzione del gioco.

### 4.2.2 Interfaccia con la scheda microSD

Inizialmente, la scheda è stata formattata in FAT32 e sono stati caricati i file dei giochi. Successivamente, la scheda è stata inserita nello slot della scheda microSD integrato nello schermo.

La scheda microSD è stata gestita tramite la libreria `FatFs` [3], una libreria open source che consente

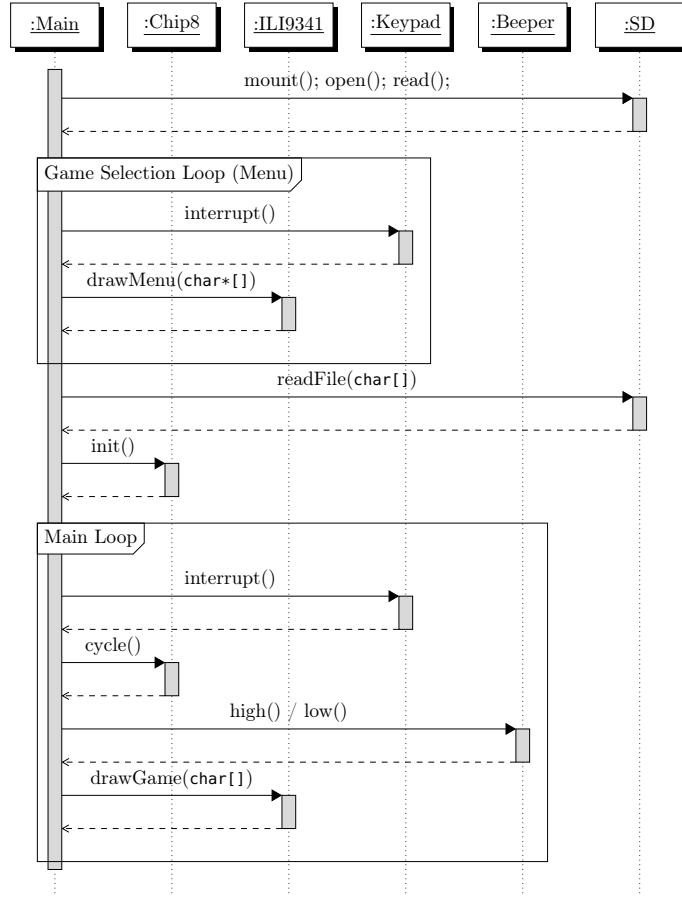


Figure 7: Sequence diagram dell’architettura software.

```

1  FRESULT fres;           // File result
2  FIL fil;               // File handler
3  UINT bytesRead;        // Bytes read from file
4  unsigned char *file_data = NULL; // Pointer to file data
5
6  fres = f_open(&fil, games[game_selected], FA_READ);
7  data = malloc(f_size(&fil));
8  f_read(&fil, file_data, (UINT)f_size(&fil), &bytesRead);
9  f_close(&fil);
10 memcpy(&vm.RAM[PC_OFFSET], data, f_size(&fil));
11 free(data);

```

Listing 3: Caricamento di un gioco dalla scheda microSD.

di interfacciarsi con filesystem FAT12, FAT16 e FAT32. Questa libreria é stata sviluppata da ChaN, un *mediocre<sup>1</sup>* *embedded system engineer* giapponese che ha sviluppato anche la libreria *Petit FatFs* per microcontrollore con risorse limitate. Per la comunicazione tra il microcontrollore e la scheda microSD é stato utilizzato il protocollo *Serial Periferal Interface* (SPI), in particolare il microcontrollore é stato configurato come master e la scheda microSD come slave. L’interfaccia SPI é nota per non utilizzare indirizzi ma rimane un protocollo valido per la comunicazione su bus.

A livello software abbiamo letto tutti i file presenti sulla scheda microSD e li abbiamo memorizzati in un vettore. Questo per avere la possibilità di selezione e successivamente l’aperura del file selezionato.

Nel listato 3 é possibile vedere il codice che consente di caricare un gioco dalla scheda microSD. E’

<sup>1</sup>[http://elm-chan.org/profile\\_e.html](http://elm-chan.org/profile_e.html)

importante notare che a riga 12 il file viene spostato nella virtual machine, precisamente viene settato il campo `RAM` della struct `Chip8` partendo da `PC_OFFSET` che di default è settato a `0x200` come mostrato nel listato 1. Successivamente, deallocando lo spazio utilizzato temporaneamente per memorizzare il file riusciamo a manterenere il livello di memoria utilizzata pari a quella della struttura dell'emulatore, vale a dire costante.

**Ottimizzazione** Una piccola ottimizzazione, ma non da sottovalutare, è stata quella di non abilitare il Long File Names (LFN) sul filesystem FAT32 della scheda microSD. Questo implica il fatto che tutti i file devono avere un nome di al massimo 13 byte (caratteri ASCII). Così facendo abbiamo risparmiato in termini complessità di computazione, memoria flash e SRAM.

#### 4.2.3 Menu di selezione

La selezione del gioco avviene tramite un menù di selezione. Il menu è composto da una lista di giochi presenti sulla scheda microSD suddivisi in varie pagine e da una sezione per la selezione selezione per la velocità di esecuzione e la modalità di esecuzione. Un aspetto importante è che l'aggiunta di un gioco non comporta la necessità di modificare il codice, in quanto il menù è generato dinamicamente in base ai file presenti sulla scheda microSD. Inoltre, basterà aggiungere un file scritto in linguaggio CHIP-8 o S-CHIP e riavviare il microcontrollore per essere pronti a giocare.

Si può notare nel listato 4 che la selezione del gioco avviene tramite il tastierino, in particolare i tasti 7 e 15 permettono di navigare tra le pagine. Il tasto 0 permette di selezionare la modalità di esecuzione ciclando tra le 3 modalità disponibili: CHIP8, SCHIP1.0 e SCHIP1.1; invece, il tasto 8 permette di selezionare la velocità di esecuzione ciclando tra le 4 preimpostate 600, 1200, 1800 e 2400. Gli altri tasti sono riservati per avviare il gioco selezionato.

```

1 while (game == -1) {
2     WriteMenu(&lcd, x, y, &font, font_h, font_w, g_size, &games, pages, modes[i_modes], freqs[i_freqs]);
3
4     for (int i = 0; i < 16; i++) {
5         if (vm.keypad[i]) {
6             switch (i) {
7                 case 0: i_modes = (i_modes + 1) % 3; break;
8                 case 7: if (pages > 0) { pages--; FillScreen(&lcd, RgbTo565(255, 0, 0)); } break;
9                 case 8: i_freqs = (i_freqs + 1) % 4; break;
10                case 15: if (pages < g_size / 12) { pages++; FillScreen(&lcd, RgbTo565(255, 0, 0)); } break;
11                default:
12                    if (i >= 1 && i <= 6) { game = i - 1 + pages * 12; }
13                    else { game = i - 9 + 6 + pages * 12; }
14                    break;
15            }
16            releaseKey(&vm, i);
17        }
18    }
19 }
```

Listing 4: Menu di selezione.

Per ovviare al fatto che cambiando schermata potrebbero rimanere degli artefatti grafici, abbiamo deciso di pulire lo schermo ad ogni cambio di schermata. Questo è stato possibile grazie alla funzione `FillScreen`, la quale utilizza metodi dell'*Hardware Abstract Layer* (HAL) per pulire lo schermo producendo anche un effetto di transizione tra una scheda e l'altra.

#### 4.2.4 Funzionamento del keypad

Inizialmente il keypad è stato gestito tramite polling, ovvero il microcontrollore controllava ad ogni aggiornamento dello schermo ( $60 \text{ FPS} \approx 16.666 \text{ ms}$ ) lo stato dei pin di riga e di colonna per capire quale tasto fosse stato premuto. Questo approccio è stato scartato in favore dell'approccio basato su interrupt,

in quanto il polling consumava molte risorse del microcontrollore e non garantiva un framerate stabile in quanto in  $\sim 16$  ms non riusciva a garantire la lettura di tutti i pin ed eseguire le restanti operazioni.

La procedura di gestione degli interrupt su microcontrollori STM32 è standard. Bisogna implementare una funzione che gestisce l'interrupt e bisogna configurare il pin GPIO come sorgente di interrupt. Nel nostra situazione, come accennato in sezione 3.1.1, i pin di riga sono stati impostati in modalità `GPIO_MODE_IT_RISING` (interrupt rising edge) e i pin di colonna in modalità `GPIO_MODE_OUTPUT_PP` (push-pull output) per permettere l'invio dell'interrupt alla pressione di un tasto.

Successivamente, viene capito quale tasto è stato premuto, leggendo il valore dei pin di riga e di colonna. Per fare ciò, i pin di riga sono stati impostati in modalità `GPIO_MODE_INPUT`. Come si puo' vedere nel listato 5 a riga 1, nella routine di interrupt viene abilitato il pin di output della prima colonna e in seguito per ogni riga viene letto il valore del pin di input per conoscere la seconda coordinata del tasto premuto, e in caso sia premuto viene salvato nella struttura dati apposita nella virtual machine. Questa operazione viene ripetuta per ognuna delle quattro colonne. Al termine della routine di interrupt (`void Interrupt_handle_keypad(uint16_t GPIO_Pin)`), i pin di riga vengono re-impostati in modalità `GPIO_MODE_IT_RISING` per permettere l'invio dell'interrupt alla pressione di un nuovo tasto.

```
1 HAL_GPIO_WritePin(C1.port, C1.pin, 1);
2 HAL_GPIO_WritePin(C2.port, C2.pin, 0);
3 HAL_GPIO_WritePin(C3.port, C3.pin, 0);
4 HAL_GPIO_WritePin(C4.port, C4.pin, 0);
5
6 if      (GPIO_Pin == R1.pin && HAL_GPIO_ReadPin(R1.port, R1.pin)) { c8_press_key(vm, KEY_00); }
7 else if (GPIO_Pin == R2.pin && HAL_GPIO_ReadPin(R2.port, R2.pin)) { c8_press_key(vm, KEY_01); }
8 else if (GPIO_Pin == R3.pin && HAL_GPIO_ReadPin(R3.port, R3.pin)) { c8_press_key(vm, KEY_02); }
9 else if (GPIO_Pin == R4.pin && HAL_GPIO_ReadPin(R4.port, R4.pin)) { c8_press_key(vm, KEY_03); }
```

Listing 5: Gestione dell'interrupt del keypad.

#### 4.2.5 Interfaccia con lo schermo

Il driver per display a cristalli liquidi **ILI9341** consente la comunicazione tramite due diverse interfacce: SPI (spiegata in sezione 4.2.2) e 8 bit parallela. Mentre l'interfaccia SPI richiede meno pin, è più lenta a causa del protocollo seriale. Al contrario, l'interfaccia parallela è più veloce ma richiede più pin. Poiché dobbiamo visualizzare i videogiochi sullo schermo e aggiornare l'immagine a circa 60 volte al secondo, abbiamo optato per l'interfaccia parallela.

L'interfaccia parallela è un metodo di comunicazione che coinvolge l'invio simultaneo di più bit di dati attraverso una serie di linee di comunicazione parallele. Nel contesto del display LCD ILI9341, l'interfaccia parallela ad 8 bit coinvolge l'uso di otto linee di dati per trasmettere informazioni al display. Oltre alle linee di dati, ci possono essere anche altre linee di controllo, come quelle per il segnale di sincronizzazione e i segnali di controllo. Quando si utilizza l'interfaccia parallela, i dati vengono inviati al display in gruppi di 8 bit simultaneamente, consentendo un trasferimento più rapido rispetto all'interfaccia SPI, che invia i dati bit per bit in sequenza.

Per l'inizializzazione abbiamo utilizzato la funzione `int Init(struct ILI9341_t *ili, struct ILI9341_Pin_t {D7, D6, D5, D4, D3, D2, D1, D0, RST, CS, RS, WR, RD})`, prende in input un puntatore alla struttura dati relativa alla scheda e i pin utilizzati per la comunicazione parallela. Gli ultimi cinque pin elencati nella firma della funzione sono stati inizializzati in modalità `GPIO_MODE_OUTPUT_PP` grazie alle funzioni della HAL.

Quindi, prima di inviare la sequenza di inizializzazione al display, è necessario impostare uno stato iniziale per la scheda. Questo stato prevede che il pin `LCD_CS` sia impostato su LOW, mentre i pin `LCD_WR` e `LCD_RD` devono essere disabilitati. Il pin `LCD_RST` è utilizzato per resettare lo stato interno del display quando è attivo, conosciuto anche come *hardware reset*. Per garantire un avvio sicuro, attiviamo e disattiviamo questo pin per resettare la scheda a uno stato noto. Successivamente, il pin `LCD_RST` rimarrà disabilitato per tutta l'esecuzione.

Una fase molto importante e' riservata all'invio della *init sequence*. Di base lo schermo e' in modalità *sleep*, quindi dobbiamo mandare una sequenza di comandi per risvegliarlo. Questa sequenza di comandi è specifica per il display ILI9341 ed sono spiegati dettagliatamente nel datasheet [7]. Questi comandi sono stati inviati al display tramite la funzione `void WriteCommand(struct ILI9341_t *ili, uint8_t cmd)`. Oltre al comando di *wake up*, e' stato eseguito il *reset software*, l'impostazione del *power control* e la quantità di bit per il colore del pixel (16 bit). Tralasciando qualche comando, in fine abbiamo inviato il comando di *display on*.

```

1 void PrintScreen(struct ILI9341_t *ili, unsigned char screen[]) {
2     SetDrawingArea(ili, 0, SCREEN_WIDTH - 1, 0, SCREEN_HEIGHT - 1);
3     WriteCommand(ili, CMD_MEMORY_WRITE);
4
5     for (int i = 0; i < SCREEN_SIZE; i++) {
6         for (int j = 0; j < 8; j++) {
7             if (((screen[i] << j) & 0x80) == 1) {
8                 WriteData(ili, 0xFFFF >> 8); WriteData(ili, 0xFFFF);
9             } else {
10                WriteData(ili, 0x0000 >> 8); WriteData(ili, 0x0000);
11            }
12        }
13    }
14 }
```

Listing 6: Funzione per la stampa dello schermo.

Nel file relativo allo schermo, abbiamo implementato diverse funzioni per gestirlo. In particolare, la funzione nel listato 6 e' stata la prima implementazione capace di stampare lo schermo. Successivamente, abbiamo basato le altre implementazioni su quest'ultima aggiungendo delle *features*. Gli elementi comuni a tutte le funzioni di stampa sono la chiamata della funzione `SetDrawingArea` che definisce i vertici del rettangolo da disegnare e la chiamata della funzione `WriteCommand` con argomento `CMD_MEMORY_WRITE` per indicare al display che stiamo per inviare dei dati.

Tra le necessità che avevamo dopo questa implementazione, c'era quella di poter stampare un'immagine più grande. In questi termini abbiamo implementato una versione migliorata, la quale accetta in input anche un parametro di *scale*.

Inoltre, e' importante notare che entrambe queste implementazioni utilizzano la HAL, questo generava un overhead nelle richieste rallentando l'applicativo. Abbiamo risolto questa problematica implementato le rispettive due funzioni in *bare metal* le quali sfuggono come base le funzioni `PIN_LOW_METAL` e `PIN_HIGH_METAL` visibili nel listato 7.

```

void PIN_LOW_METAL(struct Pin_t p) {
    p.port->BSRR = (uint32_t)p.pin << 16U;
}

void PIN_HIGH_METAL(struct Pin_t p) {
    p.port->BSRR = p.pin;
}
```

Listing 7: Implementazioni bare metal di `pin_high` e `pin_low`

Entrambe queste funzioni impostano il pin GPIO che viene passato come argomento ad uno stato `LOW` o `HIGH` rispettivamente. Eseguiamo questa azione scrivendo nel registro BSRR (come mostrato nel capitolo 9.4.7 in [8]) (Bit Set/Reset Register). Nel caso della funzione `HIGH`, quando si scrive un 1 in un bit del registro BSRR, il corrispondente pin viene impostato su *high*, mentre se si scrive un 0, il pin viene lasciato invariato. Nel caso di quella `LOW`, si accede sempre al registro BSRR, ma questa volta si utilizzano i bit superiori (da 16 a 31) per impostare il pin a *low*.

#### 4.2.6 Rendering del font

E' noto che non esiste una funzione come la `printf` su microcontrollori, e di conseguenza neanche per stampare una stringa sullo schermo `ili9341` in quanto non vi e' la conoscenza ne di uno *standard output*

ne' di un font. Quindi, e' stato necessario trovare un font, e di preciso abbiamo utilizzato un font che rappresentasse ogni glifo (simbolo o lettera) come bitmap quindi in formato matriciale  $8 \times 10$  pixel descrivendoli come 10 numeri a 8 bits.

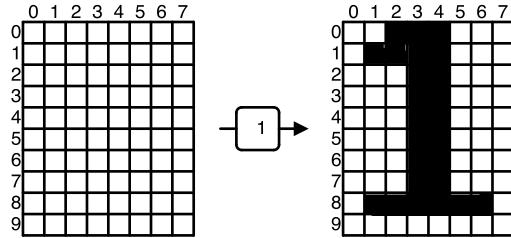


Figure 8: Esempio della mappatura di un carattere del font

In Figura 8 é possibile vedere come viene rappresentata graficamente la matrice che rappresenta un carattere. Lato implementativo ogni riga viene trattata in un `uint8_t` nel quale i bits settati a 1 sono quelli che sono quelli colorati di nero, per esempio la prima riga e' tradotta in `0b00111000` → `0d56` il quale sará il primo componente del vettore che rappresenta il carattere 1. Questa bitmap é passata alla funzione `void WriteChar(struct ILI9341_t *ili, int X, int Y, int FW, int FH, unsigned char (*font)[FH])` sottoforma di vettore (`font`) lungo 10 di `unsigned char`.

Inoltre, abbiamo implementato delle funzioni ausiliarie. In particolare, la funzione `void WriteString(struct ILI9341_t *ili, int X, int Y, int FW, int FH, int N_GLYPHS, unsigned char (*font)[N_GLYPHS][FH], char *str)` che consente di stampare una stringa sullo schermo utilizzando la funzione `WriteChar`. Questa funzione accetta in input la posizione in cui stampare la stringa, la dimensione del font, il numero di glifi presenti nel font e la stringa da stampare.

Abbiamo sfruttato queste implementazioni per stampare il menu di selezione, il quale é composto da una lista di giochi presenti sulla scheda microSD suddivisi in varie pagine e da una sezione per la selezione selezione per la velocitá di esecuzione e la modalitá di esecuzione. La funzione accetta in input un array di stringhe, un array di modalitá di esecuzione e un array di velocitá di esecuzione, e le restanti informazioni che servono a `WriteString`.

## 5 Assemblaggio

Una volta completato il software, come gi anticipato nella proposta di progetto, abbiamo deciso di assemblare le componenti hardware in un *case* in modo da nascondere i cavi e le componenti scoperte, lasciando accessibili le parti con cui il giocatore deve interagire.

Per la realizzazione, abbiamo utilizzato una base di supporto polifunzionale<sup>2</sup> di dimensione  $200 \times 150 \times 60$  correttamente rifinita per consentire la fuoriuscita del *display* e del *keypad* nella parte superiore e lateralemente sono stati applicati dei fori di dimensione variabile per permettere al grazioso suono del *beeper* di raggiungere le orecchie dell'ascoltatore e avere un interruttore per l'accensione e spegnimento della scheda.

Internamente alla base di supporto, i collegamenti sono rimasti quelli descritti in sezione 3.1.1 visibili in figura 4. Ma abbiamo aggiunto delle batterie esterne come alimentazione per la scheda evitando di doverla collegare ad una presa di corrente. Come si pu notare in figura 9 lo schermo é esterno alla scatola per permettere l'incastro con la scheda attraverso i pins dello *shield* in modo da non utilizzare colla o nastri adesivi. Per il *keypad* abbiamo utilizzato un nastro biadesivo per attaccarlo alla base di supporto, in quanto le lettere del *keypad* non rispecchiavano le originali del Cosmac Vip<sup>3</sup> abbiamo

<sup>2</sup><https://www.gewiss.com/al/it/prodotti/product.1000002.1000090.GW42002>

<sup>3</sup>[https://en.wikipedia.org/wiki/COSMAC\\_VIP](https://en.wikipedia.org/wiki/COSMAC_VIP)



Figure 9: Assemblaggio finale.

deciso di ricoprire i tasti incorretti con un adesivo indicante la lettera originale.

Il risultato finale dell’assemblaggio punta ad essere a scopo dimostrativo. Se si puntasse ad un prodotto commerciale, sarebbe necessaria una PCB ad-hoc al posto della *development board* di STM32 per ridurre la dimensione del prodotto, eventualmente un *case 3D printed* piú piccolo per aumentarne la portabilitá e in caso di miglior feedback tattile un *keypad* di qualitá superiore.

## 6 Analisi del consumo energetico

La scheda **STM32F334R8T6** permette di essere alimentata tramite un cavo USB o una batteria esterna. Per motivi di portabilitá, nel nostro caso abbiamo deciso di utilizzare 4 batterie AA da 1.5V per un totale di 6V. Non é stato un lavoro particolarmente difficile, in quanto la scheda **STM32F334R8T6** ha un regolatore di tensione integrato che permette di essere alimentata da 3.3V a 5V, quindi abbiamo collegato le batterie in serie e abbiamo collegato il positivo e il negativo al pin **VDD** e **GND** della scheda.

Per avere una stima di quanto tempo il nostro progetto potesse rimanere acceso, abbiamo deciso di misurare il consumo energetico. Per fare ció abbiamo il software fornito da ST Microelectronics chiamato *STM32CubeIDE* il quale permette di misurare il consumo energetico della scheda.

Il nostro progetto consuma all’incirca 80 mAh, quindi con 4 batterie AA da 1.5V, che hanno una capacità di 2500 mAh, il nostro progetto può rimanere acceso per circa 35 ore.

Per calcolare il consumo energetico, abbiamo utilizzato la formula

$$\begin{aligned} \text{time (h)} &= \frac{\text{capacity (mA)}}{\text{consumption (mAh)}} \\ &= \frac{4 * 2500 \text{ mA}}{80 \text{ mAh}} \\ &= 40 \text{ h} \end{aligned}$$

## 7 Considerazioni finali e sviluppi futuri

Il progetto ha avuto successo, in quanto abbiamo raggiunto tutti gli obiettivi prefissati. Abbiamo sviluppato un emulatore di CHIP-8 e S-CHIP, lo abbiamo portato su un microcontrollore STM32 e abbiamo assemblato il tutto in un case. Abbiamo anche misurato il consumo energetico del progetto e abbiamo stimato che può rimanere acceso per circa 35 ore.

Il progetto ha avuto un impatto positivo sul nostro apprendimento, in quanto abbiamo imparato a sviluppare un emulatore e a lavorare con un microcontrollore, con tutte le sfide che questo comporta.

Per quanto riguarda i tempi di realizzazione, abbiamo rispettato i tempi previsti. Come si può vedere nella tabella sottostante, i tempi previsti e i tempi effettivi sono gli stessi ma sono stati distribuiti in modo diverso. I tempi stimati per l'ottimizzazione del software sono stati ridotti, in quanto abbiamo credevamo che sarebbe stato un lavoro più impegnativo. Mentre, i tempi stimati per la realizzazione del driver video sono stati aumentati, in quanto avevamo visto che usando la HAL era troppo lento e quindi abbiamo dovuto implementare le funzioni bare metal come spiegato in sezione 4.2.2.

Nome	Tempo di lavoro Stimato	Tempo di lavoro Effettivo
Macchina virtuale	3 settimane	3 settimane
Driver video	1 settimana	2 settimane
Driver microSD	3 giorni	3 giorni
Driver keypad	1 giorno	3 giorni
Driver audio	5 giorni	2 giorni
Menu di selezione	5 giorni	1 settimana
Ottimizzazione software	2 settimane	1 settimana
<b>Totale</b>	<b>2 mesi</b>	<b>2 mesi</b>

Table 2: Tempi previsti ed effettivi per la realizzazione dei componenti software

Come sviluppi futuri, abbiamo deciso di sviluppare un *assembler* che permette dato un file scritto in linguaggio CHIP-8 o S-CHIP di generare un file binario che può essere eseguito dall'emulatore. Inoltre, abbiamo deciso di sviluppare un *disassembler* che permette di convertire un file binario in un file scritto in linguaggio CHIP-8 o S-CHIP. Questi due strumenti permetterebbero di semplificare la creazione di nuovi giochi per CHIP-8 e S-CHIP. E li renderemo disponibili all'interno dell'organizzazione GitHub<sup>4</sup>.

Come miglioramento, abbiamo deciso di implementare un'ottimizzazione che permette di aggiornare solo la sprite che è stata modificata. Questo permetterebbe di ridurre il numero di volte in cui lo schermo viene aggiornato e di conseguenza ridurre il consumo energetico. Sarà un lavoro impegnativo, in quanto richiede di modificare l'emulatore e di implementare un'API che permette di ricevere le coordinate della sprite che è stata modificata.

## References

- [1] CHIP-8 test suite. <https://github.com/Timendus/chip8-test-suite>.
- [2] Cowgod's Chip-8 Technical Reference v1.0. <http://devernay.free.fr/hacks/chip8/C8TECH10.HTM>.
- [3] elm-chan.org - FatFs - Generic FAT Filesystem Module. [http://elm-chan.org/fsw/ff/00index\\_e.html](http://elm-chan.org/fsw/ff/00index_e.html).
- [4] ISO/IEC 9899:1999 - 4. Conformance. <https://port70.net/~nsz/c/c99/n1256.html#4p6>.
- [5] Simple DirectMedia Layer. <https://www.libsdl.org>.
- [6] SUPER-CHIP v1.1. <http://devernay.free.fr/hacks/chip8/schip.txt>.
- [7] Ilitek. Tft lcd single chip driver 240rgbx320 resolution and 262k color - specification. <https://cdn-shop.adafruit.com/datasheets/ILI9341.pdf>, 2011.

<sup>4</sup><https://github.com/epartedelnostroprogettoopen-source.com/CHIP-8-0rg>

- [8] ST. Stm32f334xx advanced arm-based 32-bit mcus - reference manual. [https://www.st.com/resource/en/reference\\_manual/rm0364-stm32f334xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf](https://www.st.com/resource/en/reference_manual/rm0364-stm32f334xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf), 2020.