

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	3
3.1	Componenti utilizzati e relativi costi	4
3.2	Schema di collegamento	5
3.2.1	Schema di collegamento	5
4	Software	6
4.1	Interprete CHIP-8	6
4.1.1	Gestione del timing	7
4.1.2	Ottimizzazioni	8
4.1.3	Comportamenti ambigui	9
4.2	Porting su STM32	9
4.2.1	Architettura software	9
4.2.2	Interfaccia con la scheda microSD	10
4.2.3	Menù di selezione	10
4.2.4	Funzionamento del keypad	11
4.2.5	Interfaccia con lo schermo	11
4.2.6	Rendering del font	13
5	Assemblaggio	13
6	Analisi del consumo energetico	14
7	Considerazioni finali	14
	Riferimenti bibliografici	15

List of Figures

4	Schema del progetto.	6
5	Esempio della mappatura di un pixel.	8
6	Class diagram.	9
7	Sequence diagram.	10

8	Esempio della mappatura di un carattere del font	13
9	Assemblaggio finale.	14

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).

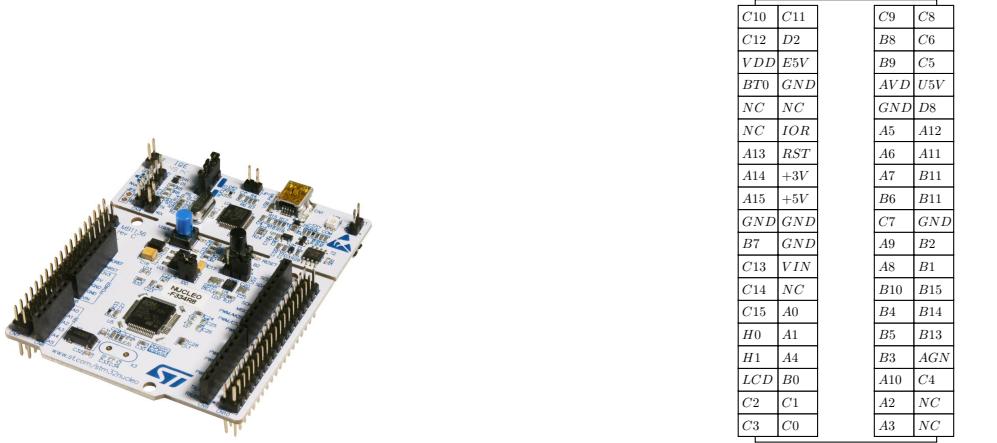
Sebbene CHIP-8 e S-CHIP siano stati tradizionalmente implementati tramite software esistono anche implementazioni hardware. Ne citiamo una in particolare scritta nel linguaggio Verilog per schede FPGA.

3 Hardware

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 offriva una quantità di SRAM troppo limitata (solo 8 Kb).

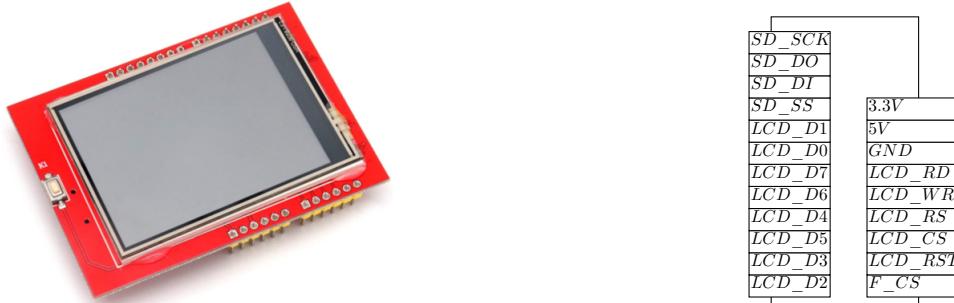
In particolare le ROM dei giochi CHIP-8 possono arrivare a pesare fino a 3.5 Kb per questo motivo è stato necessario utilizzare una scheda leggermente più potente.



(a) Il microcontrollore **STM32F334R8T6**.

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

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) Lo schermo **ILI9341**.

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

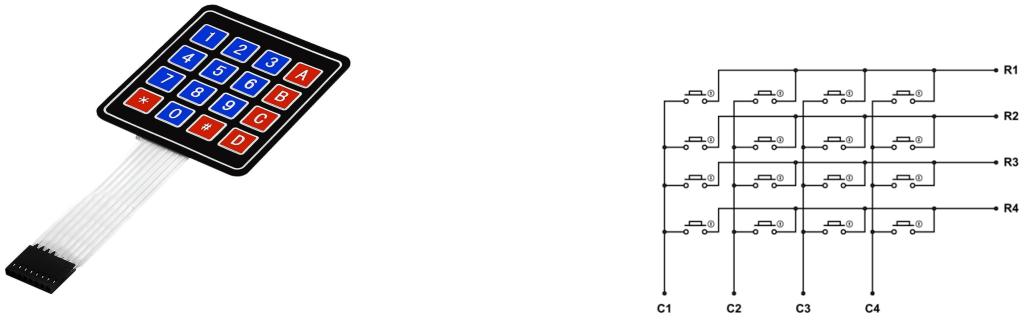
Per poter interagire con l'emulatore abbiamo utilizzato un tastierino matriciale 4×4 (Fig. 3a) analogo alla tastiera originale del COSMAC VIP¹, un microcomputer creato da Joseph Weisbecker appositamente per CHIP-8.

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

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.

¹https://en.wikipedia.org/wiki/COSMAC_VIP



(a) Il tastierino matriciale 4×4 .

(b) Struttura del tastierino matriciale 4×4 .

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.

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

3.2.1 Schema di collegamento

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).

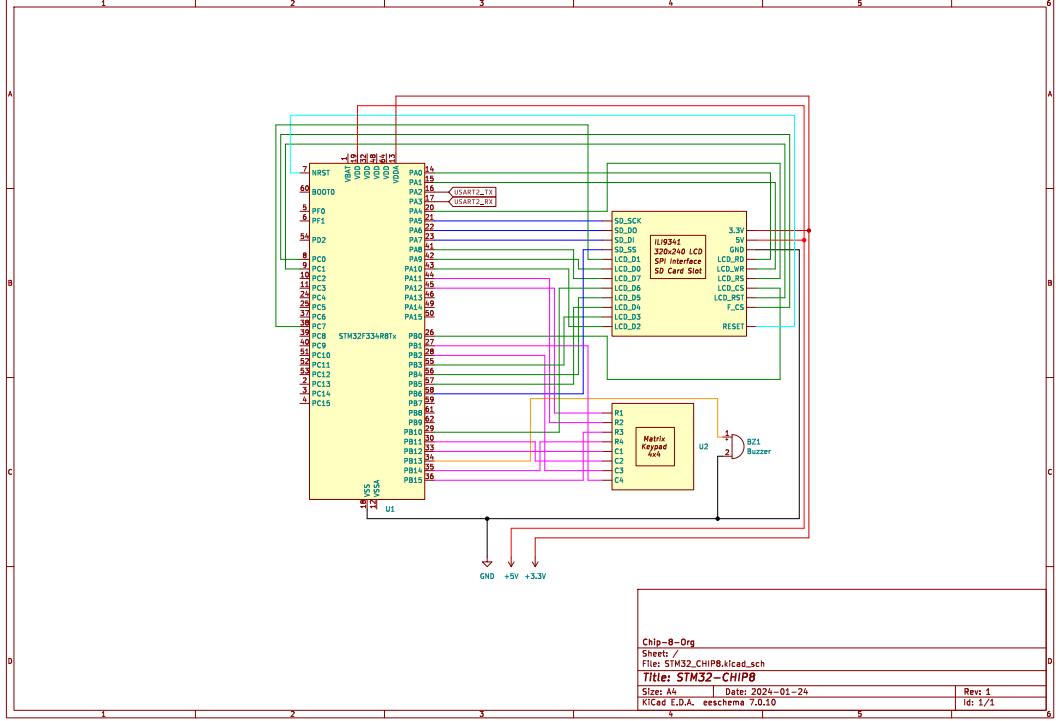


Figure 4: Schema del progetto.

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 (C_1, C_2, C_3, C_4) 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 4.2.4.

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

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.

```

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 struttura che rappresenta lo stato della macchina virtuale

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, trovare un modo alternativo per implementare le asserzioni e includere una funzione ad hoc per la generazione di numeri casuali.

```

#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.

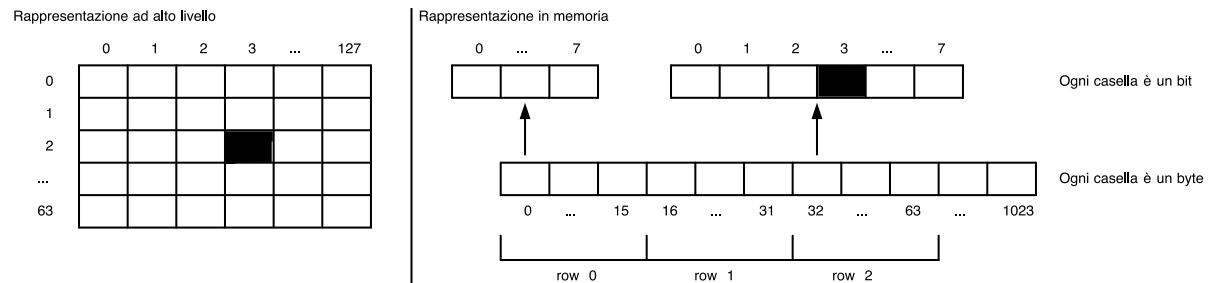


Figure 5: Esempio della mappatura di un pixel.

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 deve essere mappata ad una coordinata "in memoria".

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 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 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

In Figura 6 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.

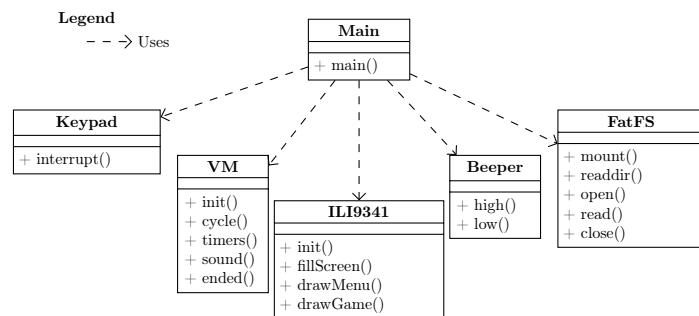


Figure 6: Class diagram.

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.

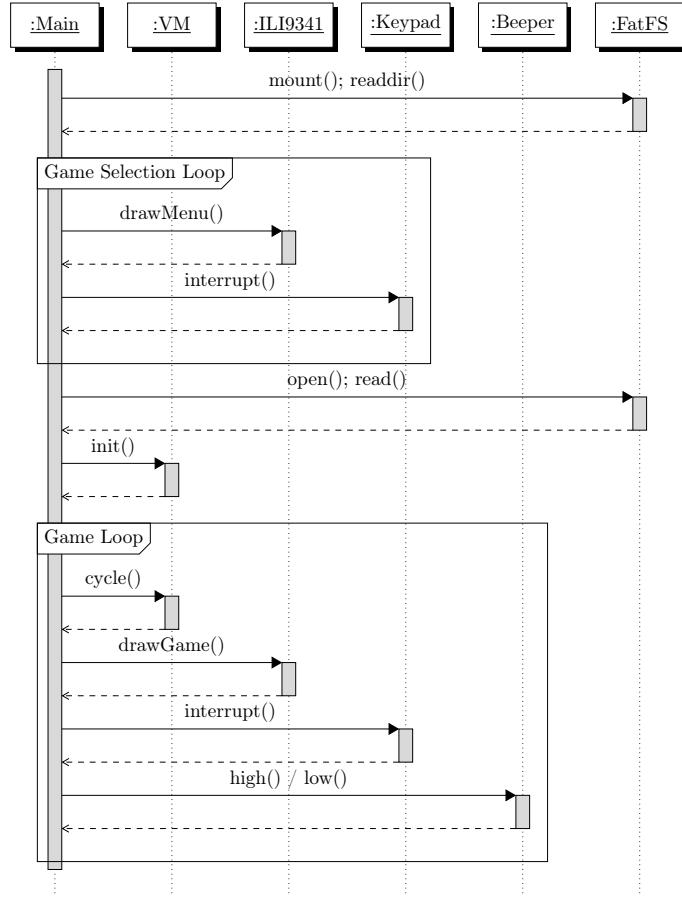


Figure 7: Sequence diagram.

4.2.2 Interfaccia con la scheda microSD

La scheda microSD è utilizzata come memoria di massa dell'emulatore contenente i file binari dei videogiochi CHIP-8. È stata formattata con FAT32 e viene gestita utilizzando **FatFs** [3], 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 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 $\sim 16 \text{ ms}$ 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 3.2.1, 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 può vedere nel listato 3 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 macchina virtuale. 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 3: 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 è riservata all'invio della *init sequence*. Di base lo schermo è 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*, è 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 4: Funzione per la stampa dello schermo.

Nel file relativo allo schermo, abbiamo implementato diverse funzioni per gestirlo. In particolare, la funzione nel listato 4 è 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, è 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 sfruttano come base le funzioni `PIN_LOW_METAL` e `PIN_HIGH_METAL` visibili nel listato 5.

```

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 5: 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

È 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 è la conoscenza ne di uno *standard output* né di un font. Quindi, è 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 8×10 pixel descrivendoli come 10 numeri a 8 bits.

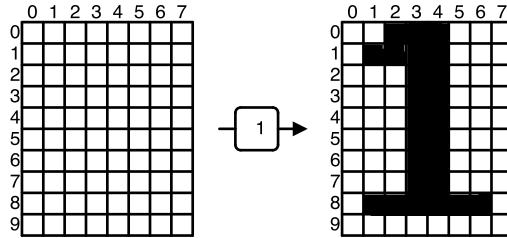


Figure 8: Esempio della mappatura di un carattere del font

In Figura 8 è possibile vedere come viene rappresentata graficamente la matrice che rappresenta un carattere. Lato implementativo ogni riga viene tradotta in un `uint8_t` nel quale i bits settati a 1 sono quelli che sono quelli colorati di nero, per esempio la prima riga è tradotta in $0b00111000 \rightarrow 0d56$ il quale sarà il primo componente del vettore che rappresenta il carattere 1. Questa bitmap è passata alla funzione `void WriteChar(struct ILI9341_t *ili, int X, int Y, int FW, int FH, unsigned char (*font)[FH])` sotto forma 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 menù di selezione, il quale 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. La funzione accetta in input un array di stringhe, un array di modalità di esecuzione e un array di velocità di esecuzione, e le restanti informazioni che servono a `WriteString`.

5 Assemblaggio

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 polifunzionale² a cui sono state apportate le modifiche appropriate per essere in grado di "montare" correttamente il display e il tastierino. Inoltre

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

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.



Figure 9: Assemblaggio finale.

I collegamenti sono rimasti identici a quelli descritti nella sezione 3.2.1, 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 1.5V ciascuna. Abbiamo collegato le batterie in serie e connesso il polo positivo al pin VDD 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] 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.
- [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.
- [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, 2020.