



Relazione Cardsorter

Laboratorio di Programmazione
di Sistemi Embedded e Mobile

Membri gruppo

de Benedictis Riccardo, matr. 587702
Musicco Nicola, matr. 591640
Cialdella Andrea, matr. 587777

Docente

Marco Fiore, PhD

Introduzione

Le carte di Magic: The Gathering sono una passione che accomuna numerosi giocatori in tutto il mondo tuttavia, con l'aumento del numero di espansioni nel corso degli anni, diventa sempre più difficile riuscire ad organizzare efficacemente la propria collezione. Qui entra in gioco la nostra idea di voler automatizzare, o meglio, semi-automatizzare, il processo di riordinamento delle amate figurine. Il sistema progettato, chiamato CardSorter-9000 basa la propria filosofia su due punti fondamentali: l'economicità e la facile riproducibilità. Per questo, pensiero cardinale del progetto è la possibile apertura ad una versione open source, volta al DIY e ad una partecipazione attiva dell'utente finale, sia esso un esperto che un neofita del mondo del fai da te.

Obiettivi del progetto / Motivazione

Il progetto CardSorter-9000 nasce proprio per risolvere il problema del “sorting”, ovvero dello smistamento automatico delle carte collezionabili, cercando di farlo in maniera veloce ed efficace e al tempo stesso trovare un modo di rendere sempre disponibile l'accesso alla propria collezione attraverso l'ausilio di un app mobile.

Per questo l'obiettivo principale di questo progetto è quello di realizzare un sistema che gestisca il riconoscimento automatico del colore di una carta di MTG e lo spostamento automatizzato della stessa nel contenitore corretto. Inoltre si vuole integrare nel sistema un processo semi-automatico di memorizzazione della carta, mediante app mobile.

Modellazione piattaforma

Il sistema è stato progettato partendo dal modello FURPS+ che evidenzia i requisiti chiave. In seguito si è realizzato il diagramma UML per i casi d'uso facendo uso del servizio online plantuml.com per la realizzazione dello schema

Requisiti (modello FURPS+)

- **Functionality**
 - **F1:**Il sistema deve riconoscere automaticamente il colore della carta (blu,nero,bianco,verde,rosso o altro)
 - **F2:**Il sistema deve smistare le carte in base al colore
 - **F3:**L'app riceve per ogni carta una notifica con il colore della carta
 - **F4:**L'utente può inserire il nome della carta per inserirla nella collezione digitale
- **Usability**



- **U1:** Il sistema fornisce un feedback visivo attraverso l'accensione di un led e un feedback sonoro con un buzzer quando rileva una carta
- **U2:** L'app deve presentare un'interfaccia semplice in cui le carte devono presentare le statistiche e una loro immagine.
- **Reliability**
 - **R1:** Il sistema deve riconoscere correttamente il colore della carta nel 90% dei casi
 - **R2:** Nel caso di interruzione collegamento l'app deve avvertire l'utente per poter effettuare nuovamente la connessione
- **Performance**
 - **P1:** Il sistema deve riconoscere il colore e inviare la notifica in meno di 2 secondi
 - **P2:** La connessione tra app e sistema deve avvenire in meno di 2 secondi
- **Supportability**
 - **S1:** Il codice deve essere strutturato in modo da poter essere modificato con la possibilità di riconoscere automaticamente il nome della carta
 - **S2:** L'app deve funzionare su dispositivi Android
- **+Altri requisiti**
 - **A1:** Il collegamento deve avvenire tramite Bluetooth
 - **A2:** L'app deve accedere alle informazioni della carta attraverso l'API di Scryfall
 - **A3:** Il sistema deve funzionare con alimentazione 5V e 3.3V
 - **A4:** Utilizzo della scheda ESP32 per la gestione degli attuatori e dei sensori

Attori e casi d'uso

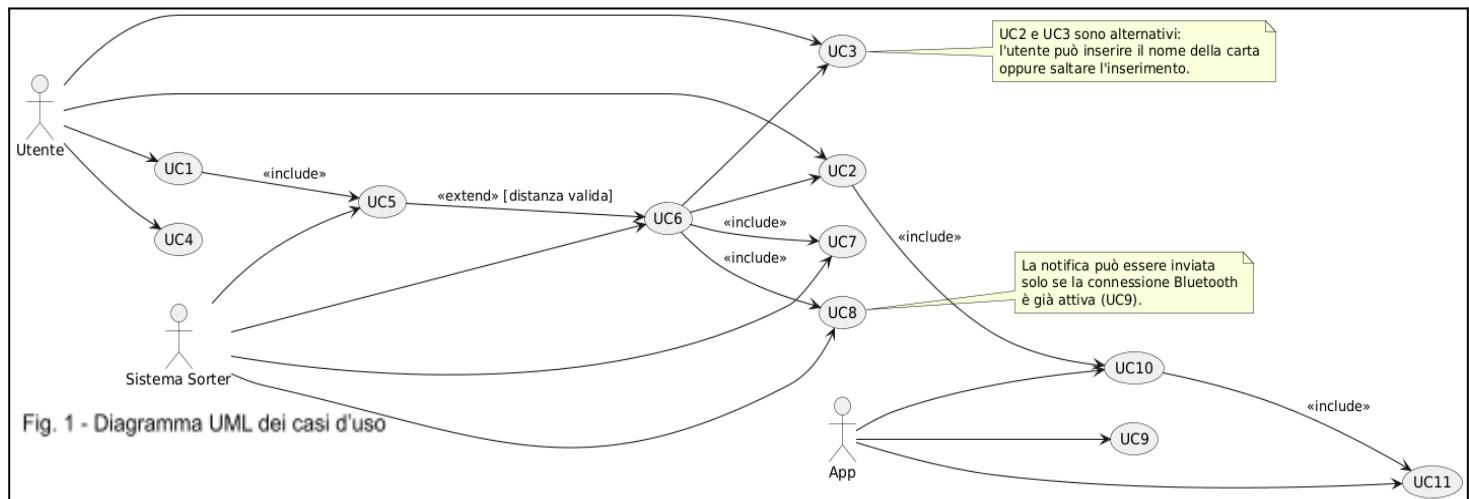
Gli attori principali sono:

- **sistema sorter:** hardware e software che riconosce il colore e smista le carte
- **Utente:** operatore che interagisce con l'app
- **App:** che comunica con l'api e aggiorna la collezione

I casi d'uso principali sono:

- **UC1:** Inserisce carta
- **UC2:** Inserisce nome carta
- **UC3:** Pressione pulsante bypass
- **UC4:** Modifica collezione
- **UC5:** Misura distanza
- **UC6:** Riconosce colore carta
- **UC7:** Smista carta
- **UC8:** Invia notifica carta
- **UC9:** Connessione Bluetooth

- UC10: Inserimento carta collezione
- UC11: Comunica con l'Api di Scryfall



Diagrammi UML (es. classi, sequenza)

- Diagramma UML delle classi [EMBEDDED]

Il diagramma delle classi (Fig. 2) mette in evidenza la costituzione del codice a livello di costruzione logica: il corpo principale del programma fa uso di funzioni personalizzate, importate nel codice attraverso una libreria realizzata ad-hoc per il progetto, che a loro volta sono costruite su classi e procedure definite in librerie open source disponibili nell'IDE di sviluppo Arduino. Va fatto notare che lo schema non è completo: le classi rappresentate in basso vengono utilizzate anche in maniera diretta nel corpo principale, ma per mantenere ordine nella figura, sono stati omessi i collegamenti diretti. Entrando più nel dettaglio:

- MainSketch: corpo principale che contiene le due procedure di inizializzazione, adibita al setup delle connessioni seriali, dei due sensori, di buzzer e pulsanti e del collegamento bluetooth con l'app mobile; e di esecuzione ciclica, che contiene il vero e proprio processo di riconoscimento della carta e smistamento;
- CARD_SORTER_9000_lib: libreria personalizzata che astrae le principali funzioni utili - inizializzazione dei dispositivi, lettura e normalizzazione dei dati, riconoscimento del colore e controllo dei servomotori;
- VL53L0X e Adafruit_TCS34725: classi che permettono la definizione dei sensori come oggetti, costruendo opportune procedure per l'inizializzazione e la lettura dei dati;
- Servo: classe che costruisce gli oggetti rappresentanti i servomotori. Comprende le procedure di inizializzazione, di descrizione dei pin di collegamento e di controllo del movimento;



- TwoWire: classe che descrive i collegamenti seriali. Costituisce la base per qualsiasi tipo di collegamento seriale presente nel progetto;
- BluetoothSerial: basata su TwoWire, costruisce gli oggetti relativi alla connessione tramite Bluetooth Classic;
- Bounce: classe di metodi avanzati volti alla minimizzazione degli effetti di rimbalzo sui pulsanti.

- **Diagramma UML di sequenza [EMBEDDED]**

Describe (Fig. 3) il flusso temporale delle operazioni durante un ciclo di smistamento, mettendo in evidenza l'ordine di chiamata delle funzioni e delle procedure e lo scambio di dati nei vari scope. Nel dettaglio:

- User → MainSketch: avvio del ciclo di controllo;
- *Lettura Distanza*:
 - MainSketch → CARD_SORTER_9000_lib: chiamata della funzione getDist(). Ottenimento della distanza media. Passaggio intermedio per la classe VL53 e chiamata della procedura di lettura della distanza istantanea;
- *Lettura Colore*:
 - MainSketch → TCS: chiamata alla procedura per la lettura dei colori rawData();
 - MainSketch → CARD_SORTER_9000_lib: chiamata della procedura di normalizzazione dei dati;
 - MainSketch → CARD_SORTER_9000_lib: chiamata della funzione di riconoscimento del colore;
- *Smistamento*
 - MainSketch → CARD_SORTER_9000_lib: ricorso alla procedura di messa in moto dei motori. La chiamata presenta un passaggio intermedio nella classe di definizione dei servomotori;
- *Bluetooth*:
 - MainSketch → Bluetooth: reset del flag di proseguimento. Inizio attesa risposta dal client bluetooth;
 - User → MainSketch: la ricezione positiva della flag fa ricominciare il loop.

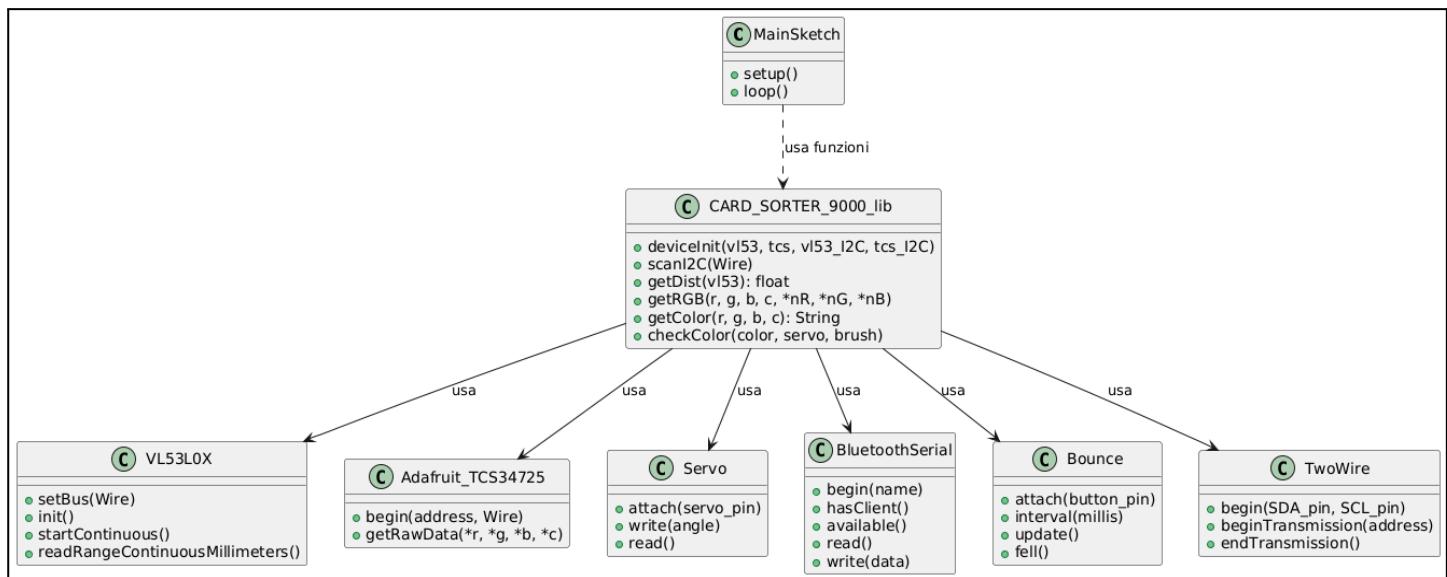


Fig. 2 - Diagramma UML delle classi [EMBEDDED]

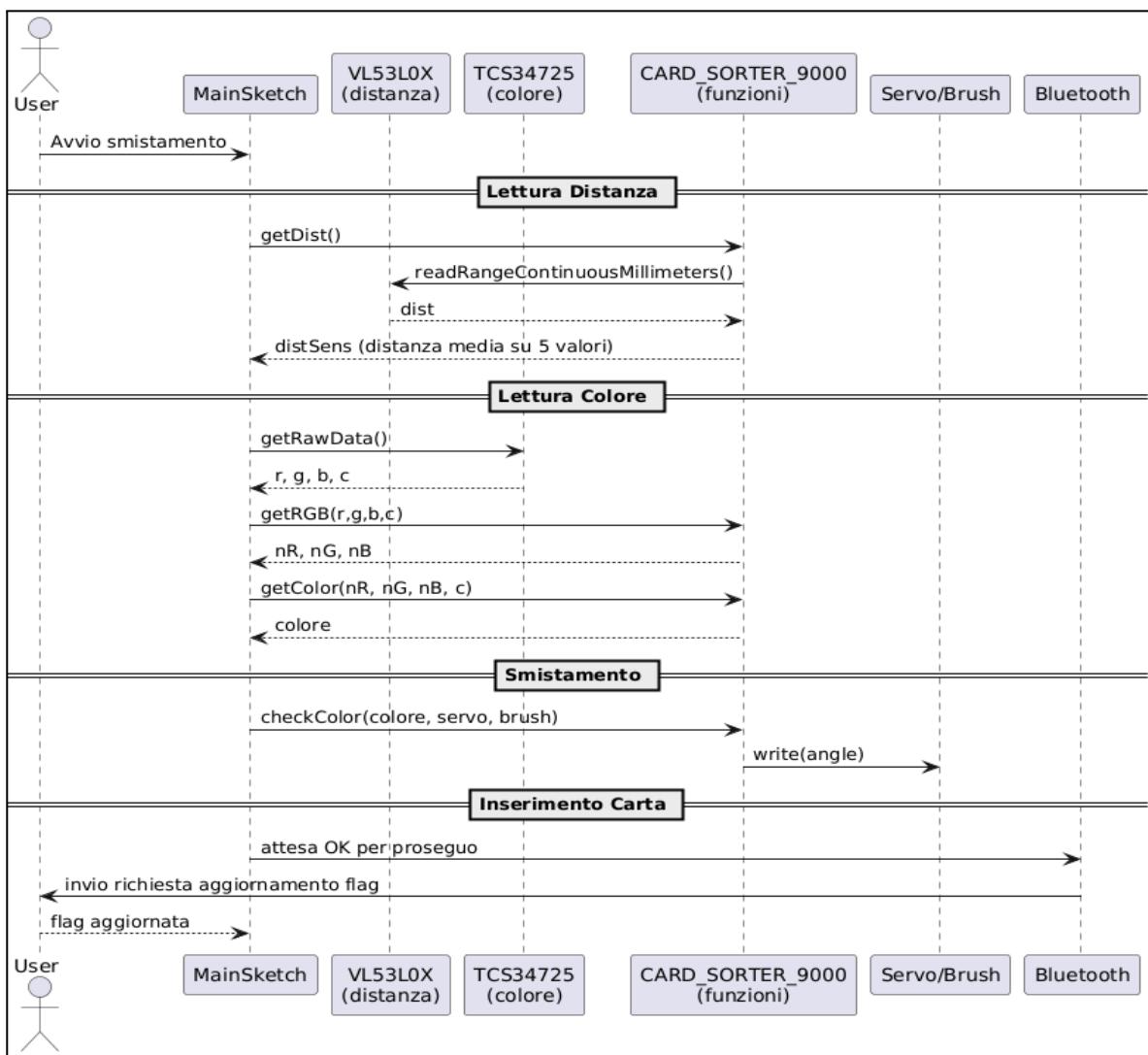


Fig. 3 - Diagramma UML della sequenza [EMBEDDED]

- Diagramma UML delle classi [MOBILE]

Il lavoro svolto per la parte embedded del progetto viene riproposto per la parte mobile. Le classi analizzate sono:

- Homepage e HomePageState: la parte principale dell'app dove vengono inseriti i widget e viene impostata la connessione Bluetooth del dispositivo. Qui c'è la logica principale dell'applicazione. Vengono anche salvate le variabili principali. Inoltre sono presenti funzioni di visualizzazione tramite filtri e ricerca.
- MagicCardApi: gestisce la chiamata all'api Scryfall.
- CardListWidget: gestisce la parte di visualizzazione e impostazione dell'immagine della carta nella finestra.
- CardWidget: si occupa di inserire correttamente l'immagine della carta.
- GameCard: classe che rappresenta la carta come oggetto.
- CardDialogs: si occupa della gestione delle visualizzazioni dei form per l'inserimento, la modifica e la cancellazione delle carte.

- Diagramma UML di sequenza [MOBILE]

Viene descritto il flusso di operazioni per la continuazione dello smistamento e dell'inserimento delle carte.

Trasmissione Colore

SistemaBluetooth -> HomePage: L'ESP32 invia il colore rilevato via Bluetooth all'app tramite il listener onDataReceived.

Gestione Stato

HomePage -> HomePage: L'app aggiorna lo stato con pendingBluetoothColor per gestire la carta in sospeso.

Notifica Utente

HomePage -> Utente: Viene mostrato un alert nella UI che segnala la presenza di una carta rilevata.

Azione Utente

Utente -> HomePage: L'utente preme il pulsante "Inserisci" per procedere con l'aggiunta.

Apertura Dialogo

HomePage -> CardDialogs: Viene visualizzata la finestra di dialogo specifica per le carte Bluetooth.

Inserimento Dati

Utente -> CardDialogs: L'utente inserisce il nome della carta e conferma.



Recupero Dati Online

CardDialogs -> MagicCardApi: Richiesta all'API Scryfall per ottenere i dettagli della carta.

Comunicazione con Scryfall

MagicCardApi -> ScryfallAPI: Chiamata HTTP all'endpoint pubblico delle carte Magic.

Restituzione Dati

ScryfallAPI -> MagicCardApi: L'API restituisce tipo, attacco, difesa e URL immagine.

Creazione Carta

CardDialogs -> CardDialogs: Viene istanziato un nuovo oggetto GameCard con i dati ottenuti.

Aggiornamento Inventario

CardDialogs -> HomePage: La carta viene aggiunta alla lista principale allCards.

Salvataggio Locale

HomePage -> ArchiviazioneLocale: I dati vengono persistiti nel file locale assets.json.

Applicazione Filtri

HomePage -> HomePage: Vengono riapplicati i filtri per aggiornare la lista visualizzata

Conferma Hardware

CardDialogs -> SistemaBluetooth: Viene inviato il byte 0x59 come segnale di successo.

Chiusura Procedura

CardDialogs -> Utente: La finestra di dialogo si chiude, completando l'operazione.

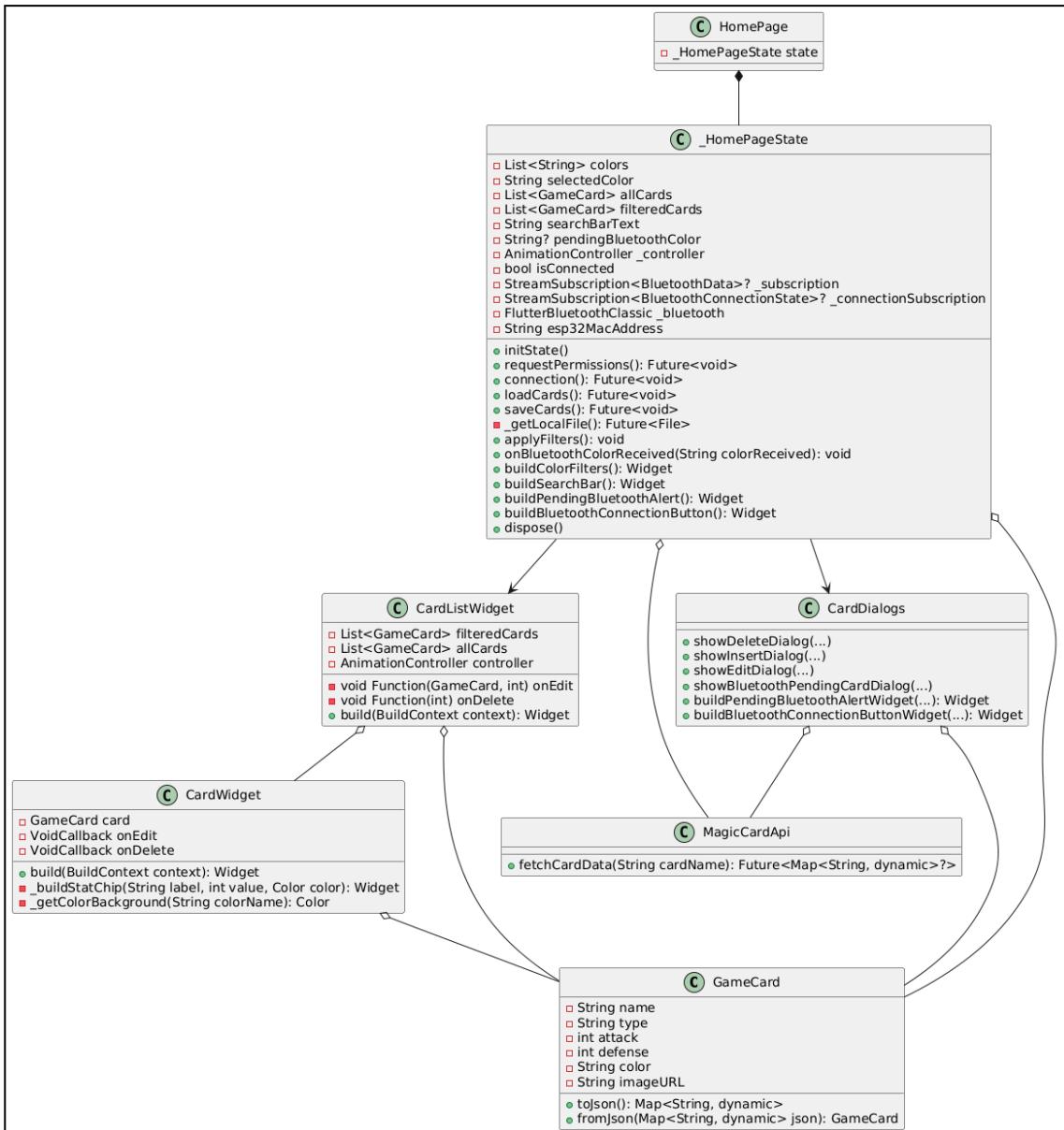


Fig. 4 - Diagramma UML delle classi [MOBILE]

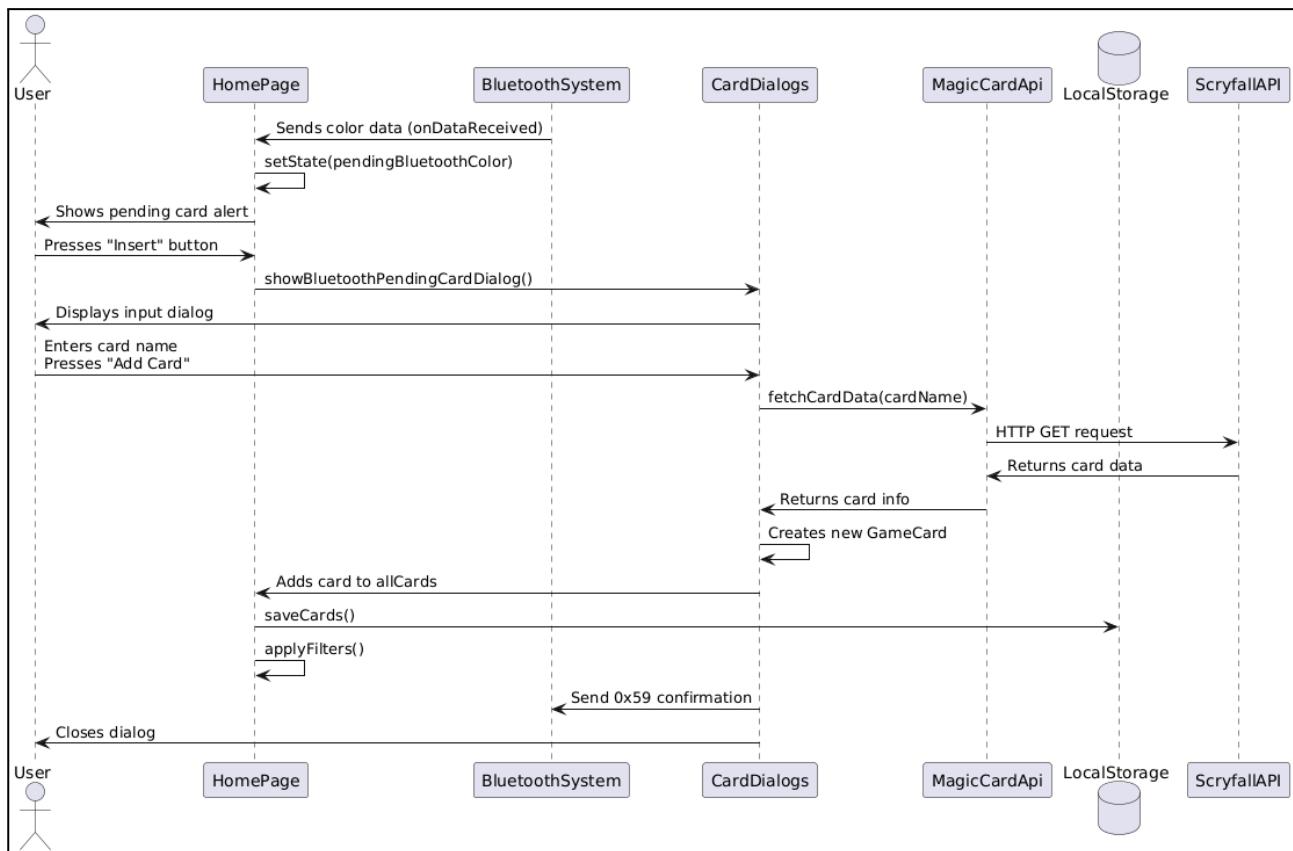


Fig. 5 - Diagramma UML della sequenza [MOBILE]

Realizzazione prototipo

Il prototipo è stato realizzato su una breadboard in modo da poter effettuare in maniera facile e immediata eventuali modifiche.

Schema circuitale

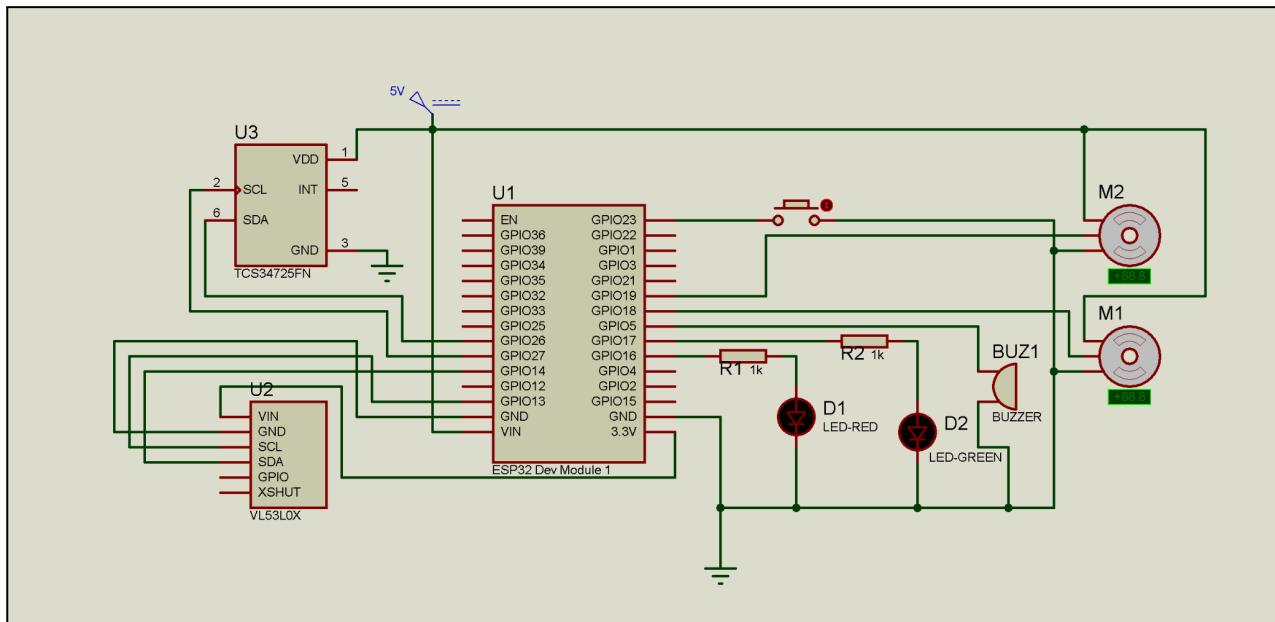


Fig. 6 - Schema circuitale

Per quel che concerne lo schema circuitale, l'unica scelta di nota effettuata è quella di non utilizzare i due pin classici predisposti da Espressif per il collegamento seriale, ovvero GPIO21 e GPIO22. L'obiettivo è quello di facilitare la costruzione fisica del sistema, evitando di affollare la stessa sezione di pin con troppi collegamenti. In tal modo, i sensori sono posizionati sullo stesso lato, creando anche ordine a livello di collocazione dei componenti.

Scelte progettuali



Fig. 7 - Sensore VL53L0X

Per il progetto si è scelto il sensore Time of Flight VL53L0X per rilevare la presenza della carta. La scelta è ricaduta su un sensore al laser anziché su un modello IR poiché tali sensori non sono influenzati dalla luce ambientale e dalla riflettività della carta che potrebbero fornire dei falsi positivi. Inoltre rispetto ai sensori ad ultrasuoni, il modello scelto gode di maggiore sensibilità e minor consumo energetico.



Il principio di funzionamento di questo sensore si basa sulla misurazione del tempo impiegato da degli impulsi di luce infrarossa temporizzati inviati dal fotodiode presente sul PCB e riflessi dall'oggetto presentato al sensore.

Poiché la velocità della luce è costante allora è facilmente calcolabile la distanza dell'ostacolo dal sensore come:

$$d = \frac{c\Delta t}{2}$$



Fig. 8 - Sensore TCS34725

Per la valutazione del colore invece si è optati per il sensore TCS34725 poiché rispetto ad altri sensori di colore, come il TCS3200, presenta una migliore accuratezza e gode di un filtro IR interno che riduce l'interferenza della luce infrarossa. Il funzionamento del sensore si basa sulla misura della corrente attraversante una matrice di quattro fotodiodi, ognuno dei quali è coperto da un materiale che assorbe i colori complementari a quello che vuole captare. In tal modo, i diodi sono in grado di registrare rispettivamente rosso, verde e blu. Il quarto diodo si occupa di misurare il livello di luce totale che impatta il sensore stesso.



Fig. 9 - ESP32 Dev Module I

Poiché entrambi i sensori richiedono un controllo mediante protocollo di comunicazione seriale I2C si è scelto come microcontrollore la scheda ESP32 Dev Module 1. Le caratteristiche di interesse per questa scheda sono la presenza di un modulo bluetooth, il supporto di due bus I2C fisici indipendenti, un processore centrale che opera a velocità maggiore di quella di Arduino Uno, l'elevato numero di pin I/O e la possibilità di sfruttare l'ambiente di sviluppo di Arduino per la programmazione. A questo proposito, nonostante questa scheda non sia nativamente presente tra quelle programmabili all'interno di 'Arduino IDE', è possibile indicare nella sezione "preferences" l'URL del file .json, disponibile sul GitHub ufficiale di [Espressif](#), che permette di aggiungere l'ESP32 tra le schede utilizzabili.



Fig. 10 - Servo Motore MG90S

Lo smistamento è eseguito con due servo motori MG90S, con rotazione massima di 180° controllati attraverso un segnale PWM. In particolare il motore M1 si occupa della rotazione dei contenitori effettuando spostamenti di 60° o 120°, mentre M2 comanda una spazzola che spinge la carta presentata al sensore di colore alla sua destra o alla sua sinistra, dividendo la base circolare in due

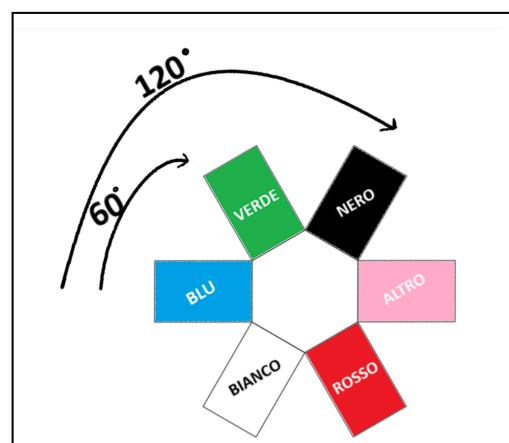


Fig. 11 - Schema Rotazione raccoglitrice

quadranti comprendenti tre contenitori ciascuno. In tal modo è possibile sfruttare intelligentemente i 180° disponibili per M1 compiendo mezze rotazioni anziché rotazioni intere.

La comunicazione con l'utente avviene tramite App e attraverso feedback visivi mediante l'utilizzo di un led rosso e uno verde. Per una comunicazione più efficace ai led si associa un segnale acustico realizzato con un buzzer attivo.

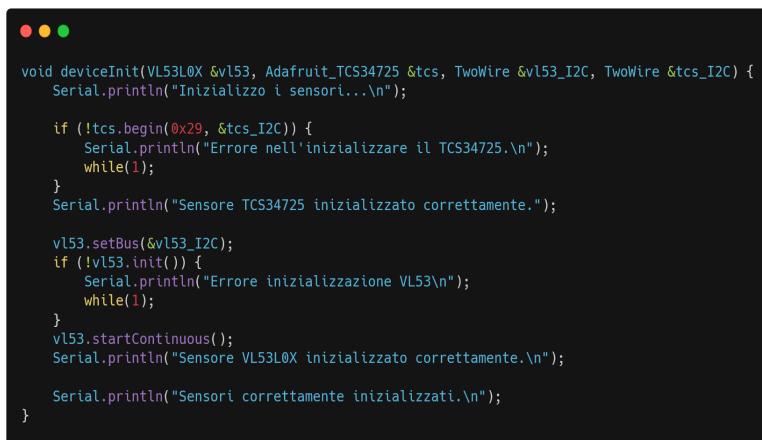
Per quanto riguarda la realizzazione dell'app si è scelto di utilizzare il framework Flutter per un possibile scaling multipiattaforma futuro.

Parti rilevanti del codice

- [EMBEDDED]

Il codice di controllo è stato strutturato cercando la miglior leggibilità e chiarezza possibile. A tal fine è nata una libreria personalizzata completa di funzioni e procedure pensate ad-hoc per semplificare il sezionamento del codice. In questo senso, funzioni e procedure utilizzate sono state realizzate per compiere compiti specifici nel tentativo di scandire le tempistiche di azione di sensori ed attuatori e snellire la lettura del codice. L'enfasi è data alla semplicità. Riassumendo:

- ❖ **Librerie di base:** la libreria è costruita anzitutto



```

void deviceInit(VL53L0X &vl53, Adafruit_TCS34725 &tcs, TwoWire &vl53_I2C, TwoWire &tcs_I2C) {
    Serial.println("Inizializzo i sensori...\n");

    if (!tcs.begin(0x29, &tcs_I2C)) {
        Serial.println("Errore nell'inizializzare il TCS34725.\n");
        while(1);
    }
    Serial.println("Sensore TCS34725 inizializzato correttamente.");

    vl53.setBus(&vl53_I2C);
    if (!vl53.init()) {
        Serial.println("Errore inizializzazione VL53\n");
        while(1);
    }
    vl53.startContinuous();
    Serial.println("Sensore VL53L0X inizializzato correttamente.\n");

    Serial.println("Sensori correttamente inizializzati.\n");
}

```

Fig 13. - Funzione Inizializzazione



```

# include<Wire.h>
# include<VL53L0X.h>
# include<Adafruit_TCS34725.h>
# include<Bounce2.h>
# include<ESP32Servo.h>
# include<BluetoothSerial.h>

```

Fig 12. - librerie incluse

per essere un wrapper per tutte le altre librerie utilizzate nel programma;

- ❖ **deviceInit():** si tratta di una procedura a cui sono passati i due sensori come oggetti e le rispettive classi di rappresentazione dei collegamenti seriali. Nel corpo della procedura i sensori vengono inizializzati, indicando esplicitamente i due bus di connessione con l'ESP32. Volendo entrare nel dettaglio, le due procedure di



inizializzazione sono utilizzate come flag (comportandosi come variabili booleane) per indicare all'utente se la connessione è avvenuta correttamente o meno.

- ❖ **scanI2C()**: si tratta di una procedura pensata per valutare la correttezza delle connessioni I2C inizializzate nello script principale mediante la funzione precedente.

```
void scanI2C(TwoWire &Wire) {
    Serial.println("Scansionando il bus I2C...");
    byte error, address;
    int nDevices = 0;

    for (address = 1; address < 127; address++) {
        Wire.beginTransmission(address);
        error = Wire.endTransmission();

        if (error == 0) {
            Serial.print("Dispositivo trovato con indirizzo: 0x");
            if (address<16) Serial.print("0");
            Serial.println(address, HEX);
            Serial.println(" ");
            nDevices++;
        }
    }

    if (nDevices == 0) Serial.println("Nessun dispositivo trovato\n");
}
```

Fig 14. - Funzione ScanI2C

Nel corpo della procedura vengono mandati pacchetti dati, della grandezza di un byte, su ciascuna linea del bus I2C passato come argomento, dunque 127 in totale. Una flag (data dalla variabile "error") tiene conto della mancata ricezione del pacchetto e identifica l'indirizzo dell'eventuale dispositivo

```
float getDist(VL53L0X &vl53) {
    int i;
    float dist, sum, med;

    for (i=0; i<5; i++) {
        dist = float(vl53.readRangeContinuousMillimeters());
        sum += dist;
    }
    med = sum/5.0;

    return(med);
}
```

Fig 15. - Funzione getDist

collegato.

- ❖ **getDist()**: semplice funzione che, attraverso il sensore VL53L0X, legge cinque valori di distanza in tempo reale e ne calcola la media aritmetica.
- ❖ **getRGB()**: procedura che normalizza i valori del colore letti dal sensore TCS34725. La scelta di operare tale normalizzazione sul posto, mediante puntatori è dovuta



```
void getRGB(uint16_t r, uint16_t g, uint16_t b, uint16_t c, float* nR, float* nG, float* nB) {  
    *nR = (float)r/c;  
    *nG = (float)g/c;  
    *nB = (float)b/c;  
}
```

Fig 16. - Funzione getRGB

alla volontà di semplificare l'argomento restituito dalla funzione: l'alternativa provata in fase di sviluppo e scartata relativamente subito era quella di utilizzare un vettore di lunghezza finita che contenesse le variabili normalizzate in modo da avere il singolo return necessario alla corretta definizione della funzione secondo il linguaggio C. Tale idea è stata scartata a causa della poca praticità della soluzione, che avrebbe richiesto la separazione delle variabili singole nel corpo principale del programma. Tale soluzione risulta più elegante e visivamente chiara.

```
String getColor(float r, float g, float b, uint16_t c) {  
    if (c > 1850) return "White";  
    else if (c < 850) return "Black";  
    else if (r > 0.54 && g < 0.32 && b < 0.22) return "Red";  
    else if (r > 0.3 && g > 0.36 && b > 0.27 && c > 1250) return "Blue";  
    else if (g > b && r > b && g > 0.35) return "Green";  
    return "Other";  
}
```

Fig 17. - Funzione getColor

❖ **getColor:** particolare attenzione è stata posta alla costruzione di questa funzione. Anzitutto va notato che essa restituisce una variabile di tipo String, normalmente inesistente in C, ma implementata dall'ambiente di sviluppo Arduino per semplificare la programmazione, probabilmente seguendo una costruzione simile a quella del più recente C++. Attraverso tale funzione avviene il vero e proprio riconoscimento del colore della carta presentata al sensore e la calibrazione dello stesso ritoccando i

```
void checkColor(String colore, Servo &servo, Servo &brush) {  
    if (colore == "Blue" || colore == "Other") {  
        servo.write(0);  
        delay(1000);  
        if (colore == "Blue" ) brush.write(0);  
        else brush.write(180);  
    }  
    else if (colore == "Green" || colore == "Red") {  
        servo.write(60);  
        delay(1000);  
        if (colore == "Green" ) brush.write(0);  
        else brush.write(180);  
    }  
    else if (colore == "Black" || colore == "White") {  
        servo.write(120);  
        delay(1000);  
        if (colore == "Black" ) brush.write(0);  
        else brush.write(180);  
    }  
}
```

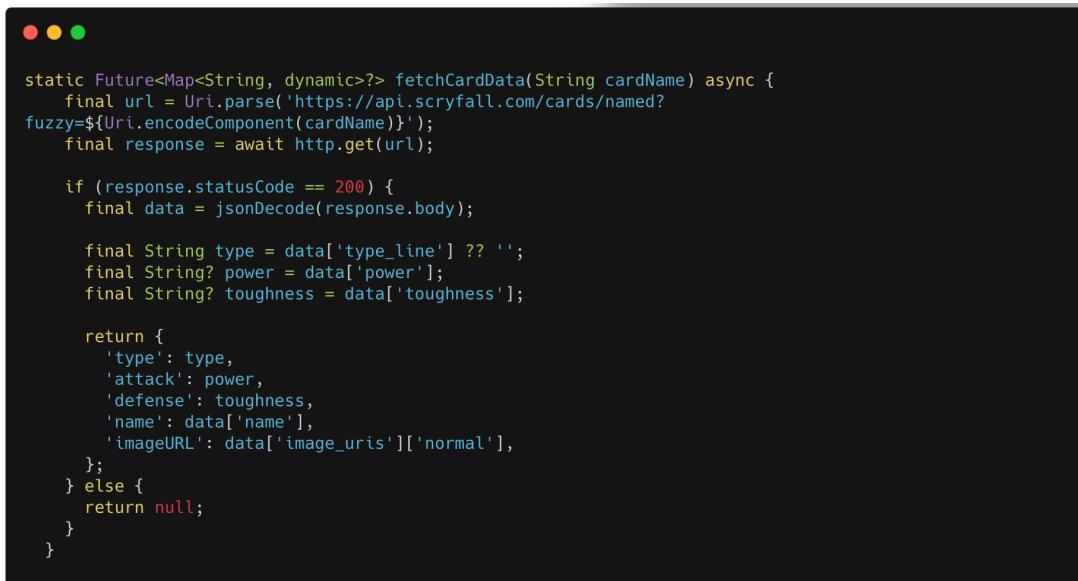
Fig 18. - Funzione checkColor

valori numerici delle soglie di riconoscimento.

❖ **checkColor:** seguendo l'idea di moto rotazionale della base, la procedura checkColor(), presa in input la stringa restituita dalla funzione precedente, aziona gli attuatori posizionando la base sul colore corretto e spingendo la carta appena analizzata nel contenitore appropriato.

Il programma principale può essere considerato a sua volta una implementazione delle funzioni descritte: esse vengono richiamate e completate con le azioni di controllo esterne, mediante il bluetooth o il pulsante manuale e con buzzer e led per una basilare comunicazione con l'utente.

- [MOBILE]



```

static Future<Map<String, dynamic>> fetchCardData(String cardName) async {
    final url = Uri.parse('https://api.scryfall.com/cards/named?fuzzy=${Uri.encodeComponent(cardName)}');
    final response = await http.get(url);

    if (response.statusCode == 200) {
        final data = jsonDecode(response.body);

        final String type = data['type_line'] ?? '';
        final String? power = data['power'];
        final String? toughness = data['toughness'];

        return {
            'type': type,
            'attack': power,
            'defense': toughness,
            'name': data['name'],
            'imageURL': data['image_uris']['normal'],
        };
    } else {
        return null;
    }
}

```

Fig 19. - Chiamata API

Per l'inserimento delle informazioni delle carte si è scelto un approccio ibrido. Dopo l'invio del nome della carta, l'app fa una chiamata all'API del database [Scryfall](#) e riceve le informazioni aggiuntive per la memorizzazione della carta. Di seguito viene riportato il codice:

Connessione con mobile

Il metodo scelto per la comunicazione fra l'app mobile e il sistema embedded è Bluetooth Classic per la sua semplicità, basando la struttura della propria codifica su quella di un semplice collegamento seriale.

Da lato embedded, il collegamento tramite Bluetooth si esaurisce in poche righe: nella fase di setup, dopo l'inizializzazione del collegamento, che per default identifica l'ESP32 come master, l'esecuzione è bloccata finché un client non si interfaccia con la scheda. Nel controllo ciclico, tramite bluetooth viene richiesto l'aggiornamento del flag di ricezione colore al client ed il codice è arrestato fino a tale momento.

Per quanto riguarda la parte mobile, in seguito alla richiesta dei permessi per l'utilizzo del Bluetooth avviene il primo tentativo di connessione che, se non andato a buon termine, creerà un widget per ritentare la connessione tramite azione manuale. In caso contrario, a connessione avvenuta, l'app è pronta a comunicare.

```
// parte di codice che gestisce la ricezione di dati
_subscription = _bluetooth.onDataReceived.listen((BluetoothData data) {
    onBluetoothColorReceived(data.asString());
});

void onBluetoothColorReceived(String colorReceived) {
    setState(() {
        pendingBluetoothColor = colorReceived;
    });
}

// parte di codice che gestisce il feedback
final List<int> response = [0x59];
try {
    await bluetooth.sendData(response);
} catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
            content: Text('Error: $e'),
            backgroundColor: Colors.red,
        ),
    );
}
```

Fig 20. - Ricezione Bluetooth

```
//--code--//

Serial.println("Inizializzo la connessione Bluetooth, controlla i dispositivi disponibili sul tuo cellulare!");

SerialBT.begin("Card_Sorter_9000");

while(!SerialBT.hasClient()) delay(100);

Serial.println("Cellulare connesso correttamente!");

//--code--//
```

Fig 21. - Setup Connessione

La gestione dell'inserimento delle carte avviene tramite la variabile "pendingColor" dove il sistema salva l'ultimo colore inviato dal dispositivo embedded. Tramite le funzioni di stato disponibili in Flutter, al momento della modifica del valore verrà aggiornata la pagina procedendo con la creazione di un widget che permetterà di inserire il nome della carta. Successivamente verrà inviato un feedback di avvenuto inserimento.

Test della piattaforma

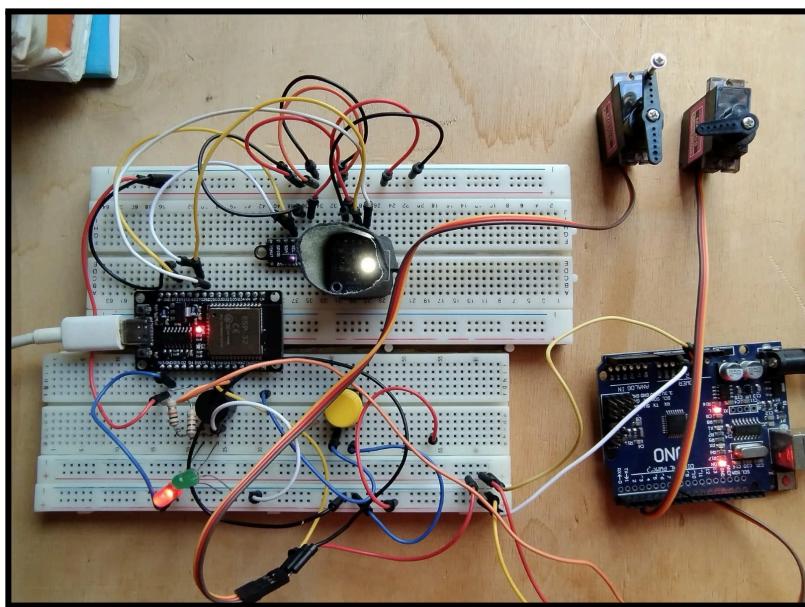


Fig 22. - Prototipo di Test

Durante la fase di test e calibrazione del sensore TCS34725, si è optato per l'utilizzo di cartoncini colorati uniformi al fine di simulare le carte Magic: The Gathering. Questa scelta è stata dettata dall'assenza di un sistema ottico (es. lente collimatrice) in grado di isolare con precisione una specifica area della carta, condizione necessaria per garantire misurazioni RGB affidabili e ripetibili.

Per una migliore rilevazione si è realizzato un cilindro di cartoncino nero in grado di isolare il sensore di colore dalla luce ambientale che genererebbe rumore nei dati rilevati.

Inoltre, in assenza di un generatore di tensione continua dalle caratteristiche adatte, si è utilizzato un Arduino UNO per alimentare i due servomotori, affidando il controllo all'esp32. Si è optati per questa modalità di test in quanto l'ESP32 può fornire al massimo 120 mA se si considera la somma delle correnti di tutte le uscite della scheda.

In fase di test i messaggi sono stati visualizzati sul monitor seriale dell'IDE di Arduino tuttavia sarebbe facilmente integrabile uno schermo OLDED (o LCD, generalmente più economico, ma limitato in funzionalità) per la comunicazione diretta con l'utente. La difficoltà incontrata durante la fase di prototipazione è stata l'assenza di hardware funzionante. L'implementazione tramite software si potrebbe realizzare utilizzando una libreria aggiuntiva, [SoftWire](#), disponibile direttamente nell'ambiente di sviluppo. Tramite tale libreria è possibile definire come oggetto una connessione virtuale che sfrutta il bit-banging per comunicare in parallelo alle interfacce seriali fisiche della scheda, sfruttando le connessioni generiche di input/output quali i pin GPIO. Con questa soluzione



si ha una velocità di trasmissione ridotta, senza però, perdite prestazionali nella visualizzazione del testo.

Discussione

Dai test effettuati è emerso che per ottenere delle valutazioni che rispettino i requisiti fissati è necessaria un'attenta calibrazione del sensore di colore all'interno della funzione checkColor().

Con un'opportuna calibrazione, la possibilità di errore si riduce notevolmente fino a considerarsi nulla.

Inoltre è emerso che i colori che vengono letti con maggiore volatilità sono il colore rosso ed il colore bianco. Il primo in quanto valori normalizzati della componente rossa troppo bassi porta l'algoritmo a confondere la lettura con il verde; il secondo, invece, in presenza di luce ambientale troppo forte, satura il valore di clear e l'algoritmo legge la carta come se fosse Blu. Entrambe queste problematiche possono essere risolte ricalibrando la funzione checkColor() in condizioni normalizzate e con l'utilizzo di opportuni modificatori come lenti o fibre ottiche.

Nel sistema è integrata una logica che, effettuata la prima connessione, permette al sistema di rimanere in comunicazione con il client Android a meno di reset forzato della scheda ESP32. Attraverso la stessa logica viene ritagliata una finestra temporale di 2 secondi che permette al client di ristabilire la connessione con la scheda in modo del tutto automatico. Oltre tale finestra, è richiesto intervento manuale da parte dell'utente. Tale implementazione permette al sistema integrato di aderire il più possibile alle specifiche FURPS+ delineate precedentemente.

Conclusioni e sviluppi futuri

Il progetto ha permesso di sviluppare un sistema integrato che dimostra in modo concreto l'interazione tra un dispositivo embedded e un'applicazione mobile, attraverso una comunicazione efficace e affidabile. Durante lo sviluppo sono state affrontate e superate diverse sfide, sia a livello di programmazione low-level su microcontrollore, sia nell'implementazione dell'interfaccia utente e della logica di comunicazione sull'app mobile. Il lavoro può essere ampliato per un utilizzo più concreto, realizzando, mediante stampanti 3D ad esempio, una struttura di smistamento fisica e migliorando il circuito con sensori più precisi e una migliore gestione della memorizzazione delle carte. Attualmente viene utilizzato un file json interno che potrebbe essere sostituito da un Database in cloud aggiungendo quindi la possibilità di avere più utenti memorizzati in app. Inoltre può essere migliorata la lettura di ciascuna carta attraverso una videocamera e un algoritmo di machine learning in grado di leggere automaticamente le caratteristiche di ciascuna carta.



Politecnico
di Bari

per migliorare la precisione e la rapidità della memorizzazione in app e la velocità di smistamento, semplificando ulteriormente il codice.