UnMutex

Voice-piloted Controller for Mission-Critical Tasks

Università degli Studi di Modena e Reggio Emilia

Corso di Studi:
Artificial Intelligence Engineering

Insegnamento:
Real Time Embedded Systems

Anno accademico:
2024/2025

Beatrice Calderara

Sommario

Questo report presenta la progettazione e implementazione di UnMutex, interfaccia vocale embedded per il controllo e per la gestione di task critici in tempo reale. La piattaforma hardware centrale è costituita dal microcontrollore Raspberry Pi Pico, che acquisisce segnali audio da un microfono analogico amplificato e li trasmette in formato digitale a un sistema host per l'elaborazione vocale. Sul lato software, il firmware sviluppato in C con FreeRTOS gestisce l'acquisizione dati, la sincronizzazione tramite mutex e la comunicazione USB seriale. Sul PC host, un modello di intelligenza artificiale implementato in Python effettua il keyword spotting in tempo reale, riconoscendo quattro comandi vocali fondamentali. La struttura distribuita del sistema ottimizza le risorse computazionali e garantisce elevata efficacia e precisione nel riconoscimento vocale. Il progetto si distingue per l'estensibilità e la modularità, con potenziali applicazioni in domotica, sicurezza industriale, e sistemi di controllo hands-free, dimostrando l'integrazione sinergica di tecnologie embedded e intelligenza artificiale per applicazioni realtime critiche.

Indice

1 Introduzione **2**

2 Requisiti e specifiche **3**
2.1 Requisiti funzionali 3
2.2 Requisiti non funzionali 4

3 Architettura del sistema **6**
3.1 Descrizione generale 6
3.2 Schema a blocchi 6

4 Progettazione hardware **8**

5 Progettazione software **10**
5.1 Architettura software e descrizione dei task principali 10
5.2 Gestione della sincronizzazione e comunicazione inter-task . . 12
5.3 Implementazione della priorità real-time 13
5.4 Formato dei pacchetti audio 14
5.5 Modello di Keyword Spotting 15

6 Implementazione **17**
6.1 C per il firmware embedded 17
6.2 Python per il modello di intelligenza artificiale 17
6.3 Ambienti di sviluppo 18

7 Conclusioni **19**

A Codice sorgente **20**

Capitolo 1

Introduzione

L'ispirazione per questo progetto nasce da una domanda fondamentale: come integrare efficacemente un sistema embedded in un'applicazione della vita reale, garantendo al contempo controlli affidabili e interazioni naturali? Da questa riflessione prende forma *UnMutex*.

UnMutex è un controller vocale progettato per gestire task critici in ambienti embedded dove la gestione concorrente e la sincronizzazione sono essenziali. Utilizzando segnali vocali come input, il sistema permette di pilotare dispositivi hardware in modo intuitivo e immediato, migliorando l'esperienza utente e ampliando le capacità interattive tipiche dei controller tradizionali.

Il progetto combina così tecnologie di acquisizione e gestione hardware basate su FreeRTOS, con sofisticati algoritmi di riconoscimento vocale implementati tramite modelli di intelligenza artificiale eseguiti in tempo reale su un sistema host. Questa architettura distribuita consente di mantenere un elevato livello di affidabilità e reattività, indispensabili per il controllo di processi critici.

Il presente documento illustra il processo di progettazione, sviluppo e implementazione di UnMutex, con particolare attenzione all'architettura hardware e software, all'integrazione dei modelli di keyword spotting e alla scelta dei linguaggi e degli ambienti di sviluppo, mostrando come l'innovazione tecnologica possa tradursi in sistemi embedded efficienti e funzionali applicabili nella realtà quotidiana.

Capitolo 2

Requisiti e specifiche

2.1 Requisiti funzionali

Il sistema deve acquisire il segnale audio tramite il microfono analogico MAX 4466, il quale è collegato al canale ADC del Raspberry Pi Pico (pin ADC26). L'acquisizione avviene con campionamento a 16 kHz, scelta che garantisce un adeguato compromesso tra fedeltà audio e vincoli computazionali del microcontrollore. Il segnale analogico viene digitalizzato dall'ADC a 12 bit e organizzato in buffer circolari di dimensione definita (BUFFERSIZE = 256 campioni), implementati con una tecnica ping-pong per assicurare continuità e prevenire perdite di dati.

Il flusso dati audio digitalizzato è trasmesso in tempo reale al PC attraverso l'interfaccia USB, sfruttando la comunicazione seriale. La trasmissione è gestita con un task dedicato ad alta priorità in FreeRTOS, che preleva i buffer dall'ADC e li impacchetta con marker di inizio e fine frame (MARKERSTART, MARKEREND), al fine di consentire una corretta decodifica lato host e la sincronizzazione dei dati.

Sul PC, un modello di intelligenza artificiale basato su tecniche di keyword spotting elabora i pacchetti audio in ingresso per riconoscere quattro comandi vocali: `on`, `off`, `up` e `down`. Il riconoscimento è svolto in tempo reale, con latenze ridotte per garantire un'interazione fluida. Ogni parola chiave riconosciuta viene tradotta in un comando seriale, che viene rispedito via USB al Raspberry Pi Pico.

Il microcontrollore, all'arrivo del comando, attiva task specifici per l'esecuzione delle azioni corrispondenti:

- `on`: attivazione del LED onboard (pin GPIO 25) mantenendolo acceso;
- `off`: spegnimento del LED;

- `up`: attivazione del lampeggio del LED con frequenza moderata, implementata tramite timer hardware;
- `down`: priorità massima, che comporta l'accensione del buzzer (pin GPIO 15, gestito tramite PWM), accompagnata da lampeggio rapido del LED.

La gestione concorrente di questi task utilizza FreeRTOS, con l'implementazione di mutex per assicurare l'accesso esclusivo alle risorse hardware condivise quali il LED e il buzzer. Questo evita race conditions e garantisce un comportamento deterministico del sistema.

Le priorità dei task sono state accuratamente definite per rispettare la logica di precedenza: il task `down` ha la priorità più alta, necessario per garantire la risposta immediata dell'allarme; seguono `up`, `off` e infine `on`. La comunicazione tra i task e la sincronizzazione avvengono tramite code e mutex FreeRTOS per mantenere un flusso di dati e comandi coerente e privo di deadlock.

In conclusione, il sistema garantisce un ottenimento e una trasmissione efficiente e tempestiva del segnale audio, un efficace riconoscimento vocale tramite AI sul PC, e il controllo affidabile e prioritario delle periferiche embedded secondo le specifiche funzioni implementate nel firmware su Raspberry Pi Pico.

2.2 Requisiti non funzionali

Il progetto impone vincoli stringenti in termini di gestione temporale e qualità del servizio, fondamentali per garantire la correttezza e la tempestività delle operazioni in un contesto real-time. La definizione delle priorità dei task nel sistema operativo real-time FreeRTOS segue una logica rigorosa, dove il comando `down` detiene la massima priorità, imponendo la sospensione immediata di task a priorità inferiore per assicurare il pronto intervento dell'allarme.

La sincronizzazione delle risorse hardware critiche, quali il LED e il buzzer, è garantita mediante mutex per evitare race conditions e garantire l'atomicità delle operazioni di controllo. I timeout configurati per l'acquisizione delle risorse prevengono deadlock, salvaguardando la robustezza operativa del sistema in presenza di contese o anomalie.

La comunicazione tra Raspberry Pi Pico e host PC, basata su USB seriale, deve rispettare requisiti di bassa latenza e alta affidabilità, minimizzando jitter e perdite di pacchetti. A tal fine, la progettazione prevede un meccanismo di buffering e code FIFO che assicurano il passaggio ordinato e tempestivo

delle informazioni tra interrupt di acquisizione ADC e task di trasmissione seriale.

Dal punto di vista software, l'architettura modulare e separata in task isolati facilita la manutenzione, l'estensibilità e la verifica unitaria, elementi imprescindibili per un sistema embedded complesso. L'organizzazione concorrenziale deve inoltre garantire la prevedibilità temporale negli scenari di carico variabile, indispensabile nei sistemi realtime.

Infine, il sistema deve mantenere alta affidabilità e stabilità durante operazioni prolungate mantenendo la conformità ai vincoli di consumo energetico e risorse limitate del microcontrollore, evitando stati di stallo e garantendo una gestione efficiente delle risorse di sistema.

Capitolo 3

Architettura del sistema

3.1 Descrizione generale

L'architettura del sistema si basa su una separazione funzionale tra l'acquisizione e l'elaborazione del segnale audio, rispettivamente svolte dall'unità embedded Raspberry Pi Pico e dal PC host. Il Pico si occupa della conversione e trasmissione dati, mentre il PC esegue il modello di intelligenza artificiale per il riconoscimento delle parole chiave.

I comandi derivanti dall'elaborazione AI sul PC vengono ricevuti dal Pico, che delega l'attivazione delle periferiche hardware (LED e buzzer) a task distinti, gestiti mediante un sistema operativo real-time. La concorrenza, la sincronizzazione e la gestione delle priorità assicurano la corretta e tempestiva esecuzione delle azioni richieste.

L'architettura si configura come un sistema distribuito con comunicazione sincrona e controllo concorrente, progettato per mantenere integrità, reattività e affidabilità nella gestione delle interazioni vocali e dei segnali hardware.

3.2 Schema a blocchi

Il segnale vocale viene acquisito da un microfono MAX4466, amplificato e campionato dall'ADC del microcontrollore Raspberry Pi Pico alla frequenza di 16 kHz, organizzando i campioni in due buffer ping-pong gestiti da FreeRTOS. Quando un buffer è pieno, il relativo puntatore viene messo in coda e un task dedicato invia il contenuto via USB verso il PC, incapsulando i campioni in pacchetti con marker di inizio e fine per garantire una ricezione robusta.

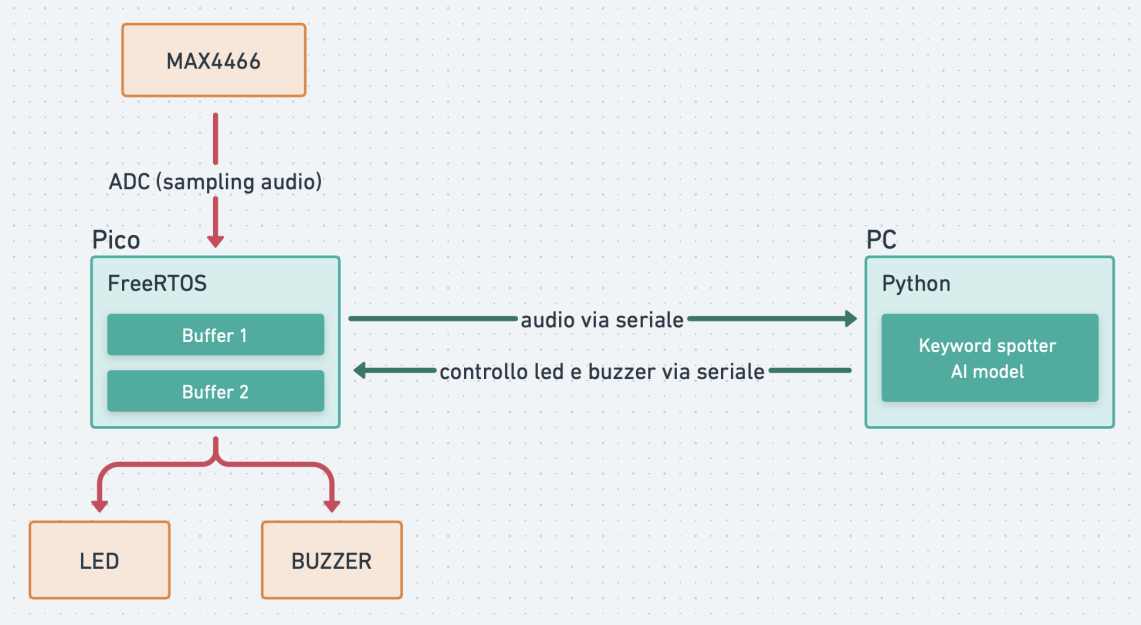

Figura 3.1: Schema a blocchi dell'architettura del sistema

Sul PC, un programma Python riceve i pacchetti audio, li ricompone in una finestra temporale e li passa a un modello di keyword spotting basato su Wav2Vec2, che esegue l'inferenza e decide se è stata pronunciata una parola chiave. In caso di rilevamento con confidenza sufficiente, il software Python invia un comando testuale sulla stessa connessione seriale verso il Pico, seguendo un semplice protocollo a stringhe.

Il Pico riceve questi comandi tramite un task di ricezione seriale FreeRTOS, che effettua il parsing della stringa e notifica i task dedicati al controllo delle uscite. I task per LED e buzzer, sincronizzati tramite mutex per l'accesso esclusivo alle periferiche, implementano le diverse modalità operative (accensione, spegnimento, lampeggio e allarme), chiudendo l'anello tra riconoscimento vocale sul PC e risposta tempo-reale sul dispositivo embedded.

Capitolo 4

Progettazione hardware

L'architettura hardware del sistema si fonda sull'impiego del microcontrollore **Raspberry Pi Pico**, scelto per la sua elevata versatilità e capacità computazionale, particolarmente adatto per applicazioni real-time grazie al processore dual-core ARM Cortex-M0+ operante a 133 MHz, alla presenza di un convertitore analogico-digitale (ADC) a 12 bit e al supporto per l'esecuzione del sistema operativo real-time FreeRTOS.

La scelta di Raspberry Pi Pico consente di disporre di numerose linee GPIO configurabili, periferiche PWM per il controllo del buzzer e un'interfaccia USB nativa per la comunicazione seriale con il PC, tutti elementi indispensabili nel contesto del progetto.

Il microfono impiegato è un modulo **MAX4466**, un microfono a elettrete con amplificatore integrato a basso rumore e consumo energetico minimo, dotato di potenziometro per la regolazione del guadagno. Questo modulo opera con tensioni di alimentazione comprese tra 2.4 e 5.5 V, ed è capace di fornire un segnale analogico ad elevata qualità, essenziale per un campionamento affidabile nella banda audio (20 Hz - 20 kHz). Le caratteristiche di soppressione dei disturbi elettromagnetici e la stabilità del segnale lo rendono adatto a progetti di acquisizione audio e riconoscimento vocale.

Il segnale in uscita dal MAX4466 è collegato direttamente al pin GP26 del Raspberry Pi Pico, corrispondente al canale ADC0, che campiona il segnale a 16 kHz con risoluzione 12 bit. L'uso della tecnica di buffering ping-pong con due buffer di dimensione 256 campioni complessivi permette di gestire l'acquisizione in modo continuo ed efficiente, minimizzando la perdita di dati in presenza di task concorrenti.

Le periferiche di output sono rappresentate da un **LED** collegato al pin GPIO 25, utilizzato per la segnalazione visuale dello stato del sistema, e un **buzzer** PWM connesso al pin GPIO 15, controllato tramite un segnale modulato in frequenza e duty cycle per generare allarmi sonori distinti.

Tutti i componenti hardware sono montati su una **breadboard** per consentire un cablaggio rapido e flessibile durante lo sviluppo e i test del prototipo, favorendo revisioni e modifiche di facile effettuazione.

Il design del sistema prevede un accordo accurato tra la frequenza di campionamento, la dimensione dei buffer e la capacità computazionale della CPU, per evitare perdite di dati durante il campionamento e la trasmissione seriale USB. Le connessioni di alimentazione e massa sono state realizzate con attenzione per minimizzare interferenze, utilizzando le caratteristiche fisiche del modulo microfonico come i bead SMD inseriti sull'alimentazione.

Questa combinazione hardware è risultata efficace nel soddisfare i requisiti funzionali e non funzionali del progetto, garantendo un'adeguata qualità del segnale acquisito, capacità di elaborazione concorrente e affidabilità nelle comunicazioni con il sistema esterno di riconoscimento vocale.

Capitolo 5

Progettazione software

5.1 Architettura software e descrizione dei task principali

Il sistema software sviluppato per Raspberry Pi Pico si basa sull'uso del sistema operativo real-time **FreeRTOS**, che consente la gestione concorrente e la sincronizzazione di più task con priorità differenziate. Questa scelta permette di rispettare i requisiti di real-time imposti dal progetto, grazie a meccanismi di preemption e mutex per la protezione delle risorse condivise.

L'architettura prevede la definizione di diversi task, ognuno con un compito specifico:

- **Task di acquisizione audio**: si occupa del campionamento del segnale analogico proveniente dal microfono collegato al pin ADC0 (GP26), utilizzando un timer ripetitivo che esegue la callback di campionamento ad una frequenza di 16 kHz. Il campione ADC viene inserito in uno dei buffer ping-pong, dopodiché il puntatore al buffer pieno viene inviato ad una *queue* per la trasmissione seriale:

```
1  // ADC sample callback
2  bool adc_sample_callback(struct repeating_timer *t) {
3
4      // Read an ADC sample and store it in the active buffer (only if we haven't reached the
           buffer size yet)
5      if (buffer_index < BUFFER_SIZE) {
6          audio_buffers[active_buffer_index][buffer_index++] = adc_read();
7      }
8
9      // If the active buffer is full, perform the buffer swap
10     if (buffer_index >= BUFFER_SIZE) {
11
12         uint16_t* p_full_buffer = audio_buffers[active_buffer_index];
13
14         BaseType_t xHigherPriorityTaskWoken = pdFALSE;
15         xQueueSendToBackFromISR(xAudioBufferQueue, &p_full_buffer, &xHigherPriorityTaskWoken);
16
17         active_buffer_index = 1 - active_buffer_index;
18
19         buffer_index = 0;
20
```

```
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }

    return true;
}
```

- **Task di trasmissione dati audio**: riceve i puntatori ai buffer pieni dalla coda e si occupa di inviare i dati via USB seriale al PC, incapsulando i campioni con marker di inizio e fine pacchetto per garantire integrità.

```
void vTaskAudioStream(void *pvParameters) {
    (void) pvParameters;
    uint16_t* p_buffer_to_send;

    while (1) {
        // Wait  until a full audio buffer is received from the queue.
        if (xQueueReceive(xAudioBufferQueue, &p_buffer_to_send, portMAX_DELAY) == pdTRUE) {

            // Send the first audio sample in the buffer
            send_packed_sample(p_buffer_to_send[0], MARKER_START);

            // Transmit all intermediate samples.
            for (int i = 1; i < BUFFER_SIZE - 1; i++) {
                send_packed_sample(p_buffer_to_send[i], MARKER_NORMAL);
            }

            // Send the last audio sample in the buffer
            send_packed_sample(p_buffer_to_send[BUFFER_SIZE - 1], MARKER_END);

            stdio_flush();
        }
    }
}
```

- **Task di ricezione comandi seriali**: ascolta sulla seriale i comandi dal PC, analizza la stringa ricevuta e notifica il task corrispondente (on, off, blink, alarm) tramite *task notifications*.

```
void vTaskSerialReceiver(void *pvParameters) {
    (void) pvParameters;

    while (1) {
        PICO_ERROR_TIMEOUT is returned.
        int c = getchar_timeout_us(1000);

        if (c != PICO_ERROR_TIMEOUT) {
            if (c == '\n' || c == '\r') {
                cmd_buffer[cmd_index] = '\0';

                if (strncmp(cmd_buffer, "CMD:", 4) == 0) {
                    char *command = cmd_buffer + 4;

                    // Match known commands and notify the corresponding task.
                    if (strcmp(command, "on") == 0) {
                        xTaskNotifyGive(xOnTaskHandle);
                    }
                    else if (strcmp(command, "off") == 0) {
                        xTaskNotifyGive(xOffTaskHandle);
                    }
                    else if (strcmp(command, "blink") == 0) {
                        xTaskNotifyGive(xBlinkTaskHandle);
                    }
                    else if (strcmp(command, "alarm") == 0) {
                        xTaskNotifyGive(xAlarmTaskHandle);
                    }
                }
                cmd_index = 0;
            }
            else if (cmd_index < CMD_BUFFER_SIZE - 1) {
                cmd_buffer[cmd_index++] = c;
            }
        }
```

```
35
36          vTaskDelay(pdMS_TO_TICKS(10));
37      }
38  }
```

- **Task di controllo LED e buzzer**: ciascuno dedicato a gestire lo stato del LED e del buzzer con sincronizzazione tramite mutex per evitare competizione. I task sono impostati con priorità differenziate in modo da rispettare la gerarchia di comando.

L'uso coordinato di FreeRTOS consente di mantenere un comportamento deterministico, evitando conflitti e garantendo che i segnali hardware vengano controllati in modo tempestivo e coerente.

5.2 Gestione della sincronizzazione e comunicazione inter-task

Nel sistema software sviluppato l'interazione tra i differenti task avviene tramite meccanismi di sincronizzazione e comunicazione propri di FreeRTOS, fondamentali per garantire la correttezza funzionale e il rispetto dei vincoli real-time.

Per la protezione delle risorse hardware condivise, in particolare il LED e il buzzer, sono stati utilizzati i **mutex**, che assicurano l'accesso esclusivo a tali periferiche evitando *race conditions* e pericolose interferenze tra task concorrenti. Negli header del firmware si possono osservare le seguenti definizioni e dichiarazioni:

```
1 // Mutex used for shared resources
2 SemaphoreHandle_t xLedMutex = NULL;
3 SemaphoreHandle_t xBuzzerMutex = NULL;
```

Prima di effettuare qualsiasi operazione su LED o buzzer, il task corrispondente esegue una chiamata a:

```
1 // Take the LED mutex for exclusive access
2 if (xSemaphoreTake(xLedMutex, portMAX_DELAY) == pdTRUE) {
3     xSemaphoreGive(xLedMutex);
4 }
```

In questo modo si evita che più task tentino simultaneamente di modificare lo stato della stessa periferica, assicurando così un comportamento deterministico.

La comunicazione tra il task di acquisizione dati audio e quello di trasmissione seriale è invece basata su una **coda (queue)** di FreeRTOS, che sostiene il paradigma produttore-consumatore. Quando un buffer audio viene popolato completamente con nuovi campioni, il puntatore a tale buffer viene inviato alla coda:

```
// Send full buffer to the queue for transmission
xQueueSendFromISR(xAudioBufferQueue, &pfullbuffer, NULL);
```

Il task seriale aspetta di ricevere dalla coda il puntatore e avvia la trasmissione via USB dei dati audio:

```
if (xQueueReceive(xAudioBufferQueue, &pbuffer, portMAX_DELAY) == pdTRUE) {
    // Send data over USB
}
```

Questa architettura consente il buffering e il passaggio sicuro di dati tra interrupt e task di differente priorità, evitando perdite o corruzioni di dati.

L'adozione dei timeout opportuni nelle chiamate alle primitive di FreeRTOS, come `xSemaphoreTake`, previene deadlock e permette al sistema di gestire anche situazioni di contesa prolungata evitando blocchi permanenti.

L'insieme di queste tecniche permette di costruire un sistema software modulare, robusto e che rispetta i requisiti di progettazione real-time per la gestione efficiente di segnali audio e comandi di controllo hardware.

5.3 Implementazione della priorità real-time

Per garantire il corretto funzionamento del sistema in un contesto real-time, è stata implementata una gerarchia di priorità tra i vari task di FreeRTOS, assicurando che le operazioni più critiche prevarichino sulle meno urgenti.

Nel firmware, ogni task associato ad un comando specifico è settato con una priorità differente:

- `vTaskAudioStream` per la gestione del campionamento audio ha la priorità più alta (`PRIORITYAUDIO` = 5), poiché la continuità del flusso dati è fondamentale per la qualità del riconoscimento vocale.
- `vTaskAlarm`, relativo al comando `down`, ha una priorità elevata (`PRIORITYALARM` = 4) in quanto deve attivare immediatamente buzzer e lampeggio rapido del LED, funzione critica di allarme.
- `vTaskBlink`, associato al comando `up`, ha priorità intermedia (`PRIORITYBLINK` = 3), permettendo il lampeggio del LED senza bloccare task più urgenti.
- `vTaskOff` e `vTaskOn`, per lo spegnimento e l'accensione del LED, hanno priorità inferiori (`PRIORITYOFF` = 2, `PRIORITYON` = 1) data la minore criticità.
- Infine, il task `vTaskSerialReceiver` ha priorità bassa (`PRIORITYRECEIVER` = 1), occupandosi della ricezione dei comandi dal PC senza interrompere i task più urgenti.

Lo scheduler di FreeRTOS esegue i task secondo queste priorità, consentendo la pre-emption dei task a priorità inferiore da parte di quelli più critici, assicurando quindi tempi di risposta minimi sui segnali di allarme.

Nel file `main.c` sono definite le priorità come segue:

```
#define PRIORITYAUDIO      5
#define PRIORITYALARM      4
#define PRIORITYBLINK      3
#define PRIORITYOFF        2
#define PRIORITYON         1
#define PRIORITYRECEIVER   1
```

e i task sono creati e associati a tali priorità:

```
xTaskCreate(vTaskAudioStream, "Audio", 512, NULL, PRIORITYAUDIO, NULL);
xTaskCreate(vTaskAlarm, "Alarm", 512, NULL, PRIORITYALARM, &xAlarmTaskHandle);
xTaskCreate(vTaskBlink, "Blink", 256, NULL, PRIORITYBLINK, &xBlinkTaskHandle);
xTaskCreate(vTaskOff, "Off", 256, NULL, PRIORITYOFF, &xOffTaskHandle);
xTaskCreate(vTaskOn, "On", 256, NULL, PRIORITYON, &xOnTaskHandle);
xTaskCreate(vTaskSerialReceiver, "SerialRx", 256, NULL, PRIORITYRECEIVER, NULL);
```

Questa implementazione della priorità real-time, combinata con la sincronizzazione tramite mutex, assicura che le azioni critiche non vengano ritardate e che il sistema risponda in modo affidabile e deterministico alle variazioni dei comandi vocali ricevuti.

5.4 Formato dei pacchetti audio

La comunicazione seriale USB tra Raspberry Pi Pico e PC utilizza un formato di pacchetti specifico per garantire l'integrità e corretto sincronismo dei dati. Il segnale audio campionato a 16 kHz e a 12 bit viene impacchettato in frame contenenti un buffer di 256 campioni.

Ogni campione è rappresentato da un valore a 12 bit, che viene combinato con un marcatore di controllo a 4 bit per formare una parola a 16 bit. Il marcatore identifica se il campione è di inizio frame, fine frame o un campione normale, permettendo al ricevente di discriminare e ricostruire correttamente la sequenza audio.

I marcatori definiti nel firmware sono:

- `MARKERSTART` (0xF000): segna il campione iniziale del buffer,
- `MARKEREND` (0xE000): segna l'ultimo campione del buffer,
- `MARKERNORMAL` (0x0000): indica un campione intermedio.

La funzione di impacchettamento in `main.c` è la seguente:

```
static inline void sendpackedsample(uint16_t adcvalue, uint16_t marker) {
    // Keep the 12 least significant bits of the ADC value
    uint16_t packed = (marker & 0xF000) | (adcvalue & 0x0FFF);

    putcharraw(packed >> 8);
    putcharraw(packed & 0xFF);
}
```

Questa codifica rende immediatamente identificabili i confini dei frame nella trasmissione continua, facilitando la decodifica lato PC e garantendo la sincronizzazione anche in presenza di eventuali troncamenti o errori di trasmissione.

Il PC riceve questo flusso seriale, simile a un flusso RAW di dati, dal quale estrae i pacchetti utilizzando i marcatori come delimitatori. Successivamente, invia indietro al Pico comandi di controllo basati sul riconoscimento vocale effettuato.

Questa strategia agevola un protocollo semplice ma robusto, adatto alle capacità hardware del microcontrollore e alle necessità di un sistema embedded real-time.

5.5 Modello di Keyword Spotting

Il modello di keyword spotting implementato nel progetto si basa su una rete neurale convoluzionale (CNN) su cui è stato addestrato un classificatore per l'individuazione in tempo reale di segmenti audio rappresentativi di specifiche parole chiave. Il modello è contenuto nel file `keyword_spotter.py` ed è responsabile dell'elaborazione dei dati audio grezzi trasmessi dal microcontrollore Raspberry Pi Pico al PC.

La pipeline di elaborazione del segnale audio alimenta il modello con feature vettoriali di tipo MFCC (Mel-Frequency Cepstral Coefficients), estratte da segmenti temporali del segnale stesso. Le MFCC vengono calcolate applicando una serie di passaggi tra cui la frammentazione del segnale, la moltiplicazione per funzioni di finestra, il calcolo della trasformata di Fourier per ottenere lo spettro di frequenze, la successiva mappatura sullo spazio delle frequenze Mel (scala psicoacustica che rappresenta come l'orecchio umano percepisce le diverse frequenze), e infine la trasformata discreta del coseno per ottenere coefficienti compatti e inequivocabili. Questa rappresentazione spettrale è compatta e robusta a variazioni di rumore e condizioni ambientali, facilitando la classificazione del segnale audio.

Il modello utilizzato è una **CNN standard**, composta da layer convoluzionali con funzioni di attivazione (ReLU), layer di pooling per ridurre la dimensionalità e fully connected layers per la parte di classificazione finale. Questo tipo di architettura è stato selezionato per la sua efficacia nel riconoscimento di pattern audio e per il bilanciamento tra accuratezza e complessità computazionale, risultando ideale per applicazioni in realtime su piattaforme con risorse limitate.

L'output del modello consiste in un vettore di probabilità, in cui ogni elemento rappresenta la confidenza associata alla classe corrispondente alla

parola chiave prevista dal modello. Per ogni segmento audio processato, viene quindi calcolata una misura di confidenza per ciascuna parola chiave `on`, `off`, `up` e `down`. Se la confidenza massima supera una soglia predeterminata (0.85), la parola chiave corrispondente viene considerata valida e il comando associato viene inviato al sistema di controllo. In caso contrario, l'output viene ignorato per evitare falsi positivi, garantendo così l'affidabilità e la precisione del riconoscimento vocale.

L'architettura distribuita, dove il Raspberry Pi Pico si occupa esclusivamente dell'acquisizione e della trasmissione dei dati audio grezzi, mentre l'elaborazione complessa è effettuata sul PC, consente di ottimizzare l'uso delle risorse computazionali e di mantenere bassa la latenza di risposta, requisito fondamentale nei sistemi realtime embedded.

Questa soluzione rappresenta un compromesso funzionale efficace tra capacità hardware e accuratezza del riconoscimento vocale, dimostrando come l'impiego di un modello CNN standard sia adeguato per applicazioni di keyword spotting in contesti embedded.

Capitolo 6

Implementazione

Nel corso dello sviluppo del progetto sono stati utilizzati due linguaggi di programmazione distinti, selezionati in base alle caratteristiche e ai compiti specifici delle diverse componenti del sistema.

6.1 C per il firmware embedded

Il firmware eseguito sul Raspberry Pi Pico è stato sviluppato interamente in **C**, con l'ausilio dell'SDK ufficiale per RP2040 e del sistema operativo real-time FreeRTOS. L'uso del linguaggio C permette un controllo fine e diretto sull'hardware, la gestione efficiente delle risorse di sistema e prestazioni elevate.

Il codice, presente nel file `main.c`, gestisce l'acquisizione analogica tramite ADC, la sincronizzazione dei task, il controllo di periferiche hardware come LED e buzzer, e la comunicazione USB seriale con il PC host. Le caratteristiche del linguaggio C consentono l'uso di funzionalità tipiche del basso livello, necessarie per l'accesso diretto ai registri hardware e la programmazione di driver.

6.2 Python per il modello di intelligenza artificiale

Per l'elaborazione del segnale audio in ambito PC e il riconoscimento vocale, è stato scelto il linguaggio **Python**. Il file `keyword_spotter.py` contiene la definizione e l'implementazione del modello. Python è un linguaggio ideale per lo sviluppo rapido e per la prototipazione di modelli di machine learning, grazie alla ricca disponibilità di librerie quali NumPy e PyTorch.

Questa scelta consente di separare chiaramente la parte di acquisizione e controllo hardware, implementata nel firmware embedded, e quella di elaborazione e riconoscimento, trasferendo la complessità computazionale più elevata al PC host.

6.3 Ambienti di sviluppo

Il firmware in C è stato sviluppato e testato principalmente tramite linea di comando, utilizzando la toolchain basata sugli SDK ufficiali del Raspberry Pi Pico. Questo approccio ha permesso di ottenere un controllo diretto sui processi di build, compilazione e caricamento del firmware, garantendo la piena compatibilità con l'hardware e l'ottimizzazione delle prestazioni.

Per il codice Python relativo al modello di intelligenza artificiale, è stato utilizzato l'ambiente di sviluppo integrato **PyCharm**. Questo IDE ha facilitato la scrittura, il debugging e il testing del modello per il keyword spotting, offrendo funzionalità avanzate per l'analisi del codice e il supporto per le librerie di machine learning impiegate.

Questa combinazione di strumenti ha consentito di mantenere una pipeline di sviluppo efficiente e modulare, distinguendo chiaramente tra la parte embedded a basso livello implementata in C e la componente AI sviluppata in Python.

Capitolo 7

Conclusioni

Questo progetto ha dimostrato come sia possibile integrare con successo un sistema embedded in un'applicazione di controllo vocale realtime, gestendo in maniera efficace la sincronizzazione e la priorità di task critici tramite meccanismi software avanzati come i mutex.

Una delle principali caratteristiche di UnMutex è la sua facile estensibilità. Il sistema può essere adattato a molteplici contesti e applicazioni reali, come ad esempio:

- la notifica prioritaria di allarmi o emergenze, che devono sovrascrivere altri comandi o task in esecuzione,
- il controllo vocale di dispositivi domotici, quali luci, termostati, o serrature, per ambienti domestici o aziendali,
- l'interfacciamento con macchinari industriali dove la sicurezza personale richiede l'attivazione immediata di procedure di emergenza basate su comandi vocali,
- applicazioni nel settore medicale per la gestione hands-free di dispositivi medici o di monitoraggio.

Grazie alla separazione tra la parte embedded dedicata alla raccolta e trasmissione dei dati audio e la parte software AI eseguita su un sistema host, il progetto coniuga efficienza hardware e potenza computazionale, garantendo reattività e affidabilità.

In sintesi, UnMutex rappresenta un esempio concreto di come le tecnologie di interazione vocale, unite alle best practice nella programmazione concorrente e realtime, possano essere utilizzate per realizzare sistemi di controllo innovativi, flessibili e scalabili, con ampie prospettive di applicazione in numerosi ambiti tecnologici e industriali.

Appendice A

Codice sorgente

Il codice sorgente è disponibile al seguente repository di GitHub.