

TESI DI LAUREA TRIENNALE

**Sistema domotico a basso costo
pilotato con protocollo Modbus**

Candidato:
Enrico Giordano
Matricola VR359169

Relatore:
Prof. Graziano Pravadelli

Introduzione

La domotica è una scienza interdisciplinare che si occupa di creare oggetti utili a migliorare la qualità della vita nella casa e più in generale negli ambienti abitati, favorendo la serenità e facendo risparmiare tempo (e soprattutto denaro) per la gestione le faccende domestiche.

Con l'avanzare della tecnologia, vengono utilizzati strumenti sempre più complessi, favorendo un sistema “*user friendly*” ma costoso, che dal punto di vista finanziario non è accessibile per la maggior parte delle famiglie. Un fattore che avvicina gli sviluppatori a queste tecnologie è la semplicità di progettazione: più avanzata è l'architettura, quindi definibile “*general purpoise*”, più è facile sviluppare nuovo software e mantenerlo nel tempo. Queste architetture però non sono dedicate esclusivamente all'ambito di utilizzo; questo è un fattore che impreziosisce tutto il progetto nel complesso e di conseguenza fa aumentare il prezzo di mercato del progetto stesso.

La soluzione a questo problema è progettare un sistema dedicato, “*embedded*”, in grado di occuparsi esclusivamente di alcuni compiti e ottimizzato sia nei costi che nell'esecuzione delle operazioni specifiche. Questo può essere un limite per vari fattori, ossia difficoltà di sviluppo, difficoltà di scelta della componentistica, ottimizzazione di risorse e codice, però offre il vantaggio di essere un progetto a basso costo dal punto di vista sia hardware che software, in quanto si propone qualcosa di più semplice ma efficace.

Questo progetto rappresenta un piccolo sistema domotico con cui si controllano le luci di una casa e i vari sensori che monitorano le stanze, utilizzando un pannello di controllo Master che comunica con le diverse periferiche slave tramite protocollo Modbus.

Indice

1	Il progetto	7
2	Architettura Hardware	9
2.1	LPC1788	9
2.2	LPC1768	10
2.3	PIC16F77	10
2.4	Considerazioni	11
2.5	Altri componenti	11
3	Architettura Software	13
3.1	Dispositivo Master	13
3.1.1	uEZ	13
3.1.2	emWin	14
3.2	Dispositivi Slave	15
3.2.1	FreeRTOS	15
4	Modello di sistema	17
4.1	Use Case Diagram	18
4.2	Sequence Diagram	20
5	Implementazione del sistema	21
5.1	Protocollo Modbus	21
5.1.1	Tipi di messaggi	21
5.1.2	Registri	21
5.1.3	Tipi di richiesta	22
5.1.4	CRC16	23
5.1.5	Sincronizzazione	23
5.2	Diagramma dei componenti	23

Capitolo 1

Il progetto

Il progetto consiste in un sistema domotico utilizzabile da un qualunque utente (quindi non esperto di informatica) mediante interfaccia grafica intuitiva ed essenziale. È composto essenzialmente da diversi dispositivi che devono comunicare tra loro in base alle scelte che l'utente attua navigando nell'interfaccia. Deve esserci quindi un dispositivo centrale che trasforma gli ordini ad alto livello dell'utente in istruzioni per gli altri dispositivi.

L'utente è in grado di eseguire operazioni di controllo della casa, tramite sensori presenti nelle stanze, e di interazione con oggetti fisici, ossia luci e motori per aprire o chiudere porte. Tutto ciò deve risultare semplice e intuitivo all'utente, quindi la comunicazione e il controllo devono essere gestiti interamente dai dispositivi, mentre l'utente deve essere solo in grado di fare la scelta tramite appositi pulsanti che compaiono nell'interfaccia. È presente quindi un menù in base al quale si sceglie il tipo di comando da eseguire e, in base alla scelta principale si viene indirizzati alla schermata apposita.

Ogni sintomo di guasto può essere diagnosticato tramite un'interfaccia particolare, chiamata di *debug*, che permette ad un tecnico di interfacciarsi con il sistema e capire il tipo di guasto (se è un guasto delle periferiche, se dei dispositivi, ecc ...). Anche l'utente può interfacciarsi con questa schermata seguendo le istruzioni del manuale; essendo basata la comunicazione su protocollo standard, è possibile imparare e comprendere i comandi per potersi interfacciare direttamente con i dispositivi fisici.

Per questioni di prestazioni, di modularità e di costi, è necessario che ci sia un dispositivo centrale (che mostra l'interfaccia grafica) in grado di comunicare con protocollo standard con gli altri dispositivi. Nell'architettura deve quindi esserci un dispositivo di tipo "Master" che comunica con dispositivi di tipo "Slave" in modo da inviargli i comandi scelti dall'utente tramite GUI. Il motivo di questa scelta è dovuto essenzialmente a due fattori:

- se ci fosse un dispositivo che si occupa di tutte le operazioni del sistema, le prestazioni sarebbero nettamente inferiori rispetto a ciò che ci si aspetta, altrimenti bisogna utilizzare un sistema più potente e quindi più costoso;
- più dispositivi possono essere posizionati in posti diversi in un'abitazione: in questo caso si assicura che ogni dispositivo sia efficace nel controllo di una sola stanza;
- per questioni estetiche, avere troppi fili in una casa può essere "brutto" e scomodo;
- tirare troppi fili da un unico dispositivo crea diversi problemi, sia di modularità (*da dove viene*

questo filo?) sia di disponibilità da parte del dispositivo centrale (*quanti fili posso controllare con il dispositivo?*) piuttosto che di diagnostica (*questo filo cosa controlla?*).

A livello di costi, avere un dispositivo molto potente è molto più costoso di avere più dispositivi dedicati che, se ottimizzati, costano molto poco. Per questo progetto verranno utilizzate delle demoboard, quindi oggetti non ottimizzati e progettati per avere un ambiente di sviluppo facile da configurare; si vedrà alla fine che ottimizzando le risorse i costi generali saranno molto ridotti.

Infine il sistema è stato progettato per essere espandibile: seguendo questo documento, è possibile creare un nuovo dispositivo in grado di interfacciarsi con il dispositivo "Master", in modo da controllare più periferiche e quindi più stanze. Inoltre, poichè la comunicazione utilizzata è standard, è possibile controllare anche dispositivi di diversa natura utilizzando la schermata di *debug*.

I sensori utilizzati devono generare un segnale comprensibile ai dispositivi che li devono controllare: devono essere quindi convertitori di condizioni ambientali analogici o digitali, il cui tasso di discretizzazione non deve superare i 16 bit. Questo vincolo è imposto per questioni economiche e di sistema, si vedrà in seguito che l'architettura scelta e il protocollo utilizzato gestiscono al meglio dati a 16 bit.

Le luci da pilotare devono essere a LED, per diversi fattori:

- sono la nuova tecnologia di luci;
- il voltaggio richiesto è adeguato al sistema, in questo modo si risparmia scegliendo un unico alimentatore di potenza adeguata;
- risultano essere un investimento, in quanto costano di più rispetto alle normali lampadine, però durano per molto più tempo, quindi si ha solo la spesa aggiuntiva iniziale, ma risulta meno costosa come tecnologia perché non ha ricambi (se non in casi eccezionali).

Capitolo 2

Architettura Hardware

È stato deciso di attuare maggiore modularità possibile, in modo da poter descrivere nel dettaglio ogni componente e la sua funzione, ma soprattutto per potenziare l'intero sistema nel corso del tempo. Per la realizzazione di questo sistema, sono state scelte 3 tecnologie differenti, associate ognuna ad un compito diverso e con software specifico.

2.1 LPC1788

Questo è un microprocessore ARM di famiglia Cortex-M3 utilizzato come dispositivo Master che controlla tutti gli altri dispositivi. È un processore a 32 bit, quindi in grado di avere 2^{32} spazi di indirizzamento. Le sue periferiche principali sono: GPIO, I2C, UART, USB, EMAC, Ethernet, AUX Stereo, INPUT Mono, PWM a 32 bit, ADC, DAC. Può essere alimentato con $3,3 \sim 5V$ e assorbe $200mA$. Possiede $512KB$ di memoria FLASH, $96KB$ di memoria SRAM, $8MB$ di SDRAM (esterna), $1KB$ di memoria sicura EEPROM (esterna) per poter resettare guasti irreparabili (perdita di BIOS, anomalie su codice, ecc...). La grande quantità di memoria FLASH permette di sviluppare un software molto esteso. Poiché al suo interno è stata integrata una GPU, offre maggiori prestazioni grafiche, quindi è stato utilizzato per presentare l'interfaccia grafica per poter interagire con tutto il sistema. Di questo processore sono state utilizzate queste periferiche:

- **GPU** per gestione di LCD TouchScreen VGA 640x480 5.7, utilizzato quindi per mostrare l'interfaccia grafica;
- **RS232** per la comunicazione seriale tra dispositivi;
- **Timer** per la gestione asincrona del tempo rispetto al ciclo di clock;
- **Cicalina** per riprodurre suoni di avviso o di errore.

Il costo associato quindi a questo componente è:

oggetto	Costo (in Euro)
processore	18
convertitore segnali TTL - RS232	2
LCD	20
Totale:	40

2.2 LPC1768

Questo è un microprocessore ARM di famiglia Cortex-M3 utilizzato come dispositivo Slave che viene controllato dal Master e interagisce direttamente con i sensori e luci. Anche questo processore è a 32 bit con quasi le stesse caratteristiche del processore precedentemente descritto, però risulta essere meno potente in quanto non possiede GPU integrata e possiede meno memoria (circa la metà). Per questo è stato deciso di usarlo per controllare i sensori e pilotare luci, anche perché il codice del dispositivo Master sarebbe risultato troppo grande e computazionalmente oneroso per questo processore. Possiede un piccolo LCD TouchScreen, che viene utilizzato solo per presentare il firmware e le caratteristiche del settaggio del protocollo per comunicare. Può essere alimentato con $3,3 \sim 5V$ e assorbe $200mA$. Di questo processore è stato utilizzato:

- **RS232** per la comunicazione seriale con il Master;
- **LCD** per presentare le impostazioni del protocollo;
- **GPIO** per controllare i sensori e le luci;
- **ADC** per convertire i segnali analogici provenienti dai sensori analogici in segnali digitali e trattarli nel sistema.

Il costo associato quindi a questo componente è:

oggetto	Costo (in Euro)
processore	18
convertitore segnali TTL - RS232	2
Totale:	20

2.3 PIC16F77

Questo è un microcontrollore PICMicro di famiglia PIC16 con memoria FLASH. Essendo un microcontrollore, è un'architettura diversa da quelle precedentemente descritte, in quanto risulta essere meno potente ma più ottimizzato a livello di costi. È un piccolo sistema a 8 bit, quindi possiede 2^8 locazioni di indirizzamento; questo significa che è molto meno performante, in quanto sono disponibili poche locazioni e quindi meno memoria contigua utilizzabile. La memoria FLASH è un quarto di quella del microprocessore LPC1768, quindi il codice scritto deve risultare il più piccolo possibile. Le periferiche disponibili sono: UART, PWM, GPIO, DAC, TIMER. Può essere alimentato con $3,3 \sim 5V$ e assorbe $50mA$ (quindi consuma molto meno dei sistemi precedenti).

Di questo processore è stato utilizzato:

- **RS232** per la comunicazione seriale con il Master;
- **GPIO** per controllare i sensori e le luci;
- **ADC** per convertire i segnali analogici provenienti dai sensori analogici in segnali digitali e trattarli nel sistema.

Il costo associato quindi a questo microcontrollore è:

oggetto	Costo (in Euro)
microcontrollore	5
convertitore segnali TTL - RS232	2
<i>Totale:</i>	8

2.4 Considerazioni

È stato deciso di sfruttare tecnologie diverse per far capire come questo sistema possa essere versatile, sfruttando la modularità, e quanti limiti possono esserci utilizzando tecnologie sempre più semplici. Ovviamente si potrebbero prediligere periferiche a basso costo, come nel nostro caso il PIC, però come si vedrà di seguito i limiti di progettazione sono talmente vincolanti che è difficile con un singolo PIC gestire tante periferiche quante ne gestisce un Cortex-M3.

Sostituire LPC1768 con PIC16F77 sarebbe svantaggioso perché non si avrebbe la stessa potenza di calcolo, quindi servirebbero più PIC e quindi costerebbe relativamente di più, oltreché non si avrebbero le stesse features. Non si avrebbe nemmeno la stessa versatilità, in quanto se si vuole collegare un nuovo sensore particolare a LPC1768, la riprogrammazione sarebbe molto più veloce e semplice del PIC16F77, quindi porterebbe dei costi maggiori anche a livello di sviluppo software.

Se invece il sistema può essere calato in un contesto statico, in cui i cambiamenti di apparecchiature sono molto rare e i sensori che vengono installati sono semplici e pochi, si può pensare di utilizzare un PIC in sostituzione al Cortex-M3.

L'unico oggetto che non può essere sostituito con qualcosa di meno costoso è il dispositivo Master, poiché porterebbe a problemi di potenza di calcolo, di rimappatura di tutto il sistema ma soprattutto di costi maggiori di sviluppo software, in quanto il software di questo oggetto è il più complesso di tutto il sistema e di conseguenza riprogettarlo per un'architettura diversa, magari meno potente, porterebbe a dei costi aggiuntivi inutili.

2.5 Altri componenti

ANCORA DA SCRIVERE

Capitolo 3

Architettura Software

La parte più interessante di questo preambolo al progetto realizzato è lo scheletro software su cui si basa tutta la progettazione. È facile pensare alla programmazione in un ambiente in cui il sistema operativo offre un "hardware abstraction layer" che garantisce una facile programmazione. Nell'ambito però dei sistemi embedded, un HAL è pressochè assente, quindi la programmazione si complica in quanto bisogna ragionare sul funzionamento dell'architettura stessa.

3.1 Dispositivo Master

Per LPC1788 il problema principale era quello di gestire l'interfaccia grafica e l'invio di messaggi agli Slave. Per gestire facilmente ciò, è stato utilizzato un sistema preconfezionato che offre delle API utili alla gestione di periferiche e per la progettazione.

3.1.1 uEZ

Questo sistema si chiama μEZ , chiamato così perché dovrebbe "ispirare" il designer di software embedded durante la progettazione e semplificarla. In realtà ci sono stati dei problemi di scrittura di queste API da parte degli sviluppatori che hanno fatto l'esatto opposto (sono comunque state corrette e successivamente verrà spiegato).

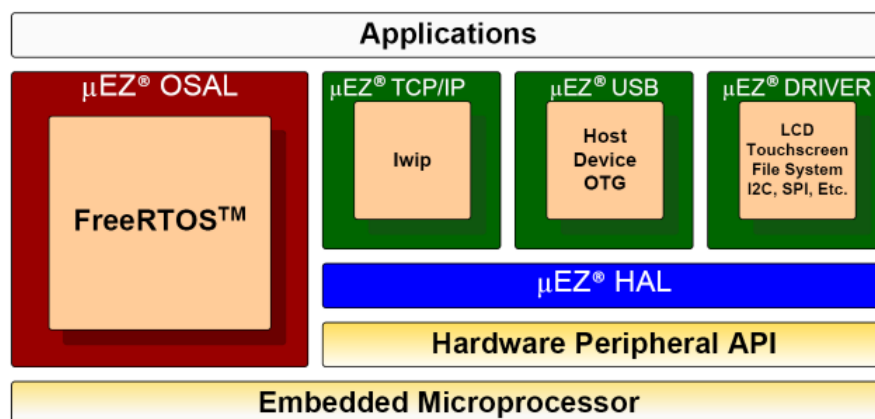


Figura 3.1: architettura del sistema μEZ

Il sistema offre innanzitutto un HAL piuttosto potente, in quanto permette di trattare ad alto livello la configurazione di tutte le periferiche. Inoltre non è necessario definire il boot di sistema (cosa non banale per questi sistemi). Esiste un porting standard per ogni processore, se non è presente

è possibile scriverlo facilmente. Il sistema deve essere compilato in due passaggi: il primo consiste nel compilare μEZ per la propria architettura, ottenendo un file di libreria da linkare al proprio progetto, il secondo consiste nello sviluppare il proprio progetto basandosi su questo sistema. È possibile, al tempo di compilazione, decidere quali periferiche utilizzare prima di generare la libreria da linkare, ottimizzando per area il proprio software.

Ciò che ha portato allo sviluppo su questo sistema software è stato, oltre all'astrazione dell'hardware che facilita la programmazione, alla predisposizione da parte del sistema di usare risorse di rete come lo stack di rete lwIP e alla presenza di sistema operativo FreeRTOS, è stata la capacità di utilizzare una libreria grafica molto importante e potente per i sistemi embedded, la libreria **emWIN**.

3.1.2 emWin

Questa è una libreria grafica per sistemi embedded molto potente, basata su una grafica molto simile alle prime versioni di Windows (sia per l'estetica sia per la gestione ottimale delle risorse grafiche). Essa offre un HAL ad alto livello molto semplice da utilizzare, con cui è possibile progettare interfacce con oggetti grafici ben noti (bottoni, input text, ecc...) senza dover interfacciarsi direttamente con l'oggetto hardware. Può sembrare una soluzione poco ottimale, visto che tutti gli elementi ad "alto livello" nella progettazione di sistemi embedded può essere reputata poco efficiente in ambito di risorse e computazione, in realtà è stata progettata in maniera talmente sofisticata che risulta ancora più efficiente della scrittura "a mano" delle stesse librerie.

La gestione dell'interfaccia tramite questa libreria ha un'implementazione molto semplice:

1. si descrive l'interfaccia implementandola personalmente o tramite appositi programmi di generazione di codice (GUIBuilder incluso nel pacchetto emWin);
2. si associa al codice corrente l'interfaccia appena creata ricordandosi di usare il comando *ExecDialogBox()* (vedere manuale emWin);
3. si associa ai widget creati del codice che viene eseguito in base agli eventi creati.

Particolare interesse deve essere prestato al metodo con cui si può interagire con i widget: per modificare o eseguire azioni generiche sul widget, è necessario farsi restituire un puntatore particolare e lavorare su di esso tramite il puntatore. Un esempio di codice:

```
#define ID_WIDGET ID_GUI + 0x00

...

EMWIN_WIDGET_TYPE my_widget;

...

WM_HWIN hItem = WM_GetDialogItem(pMsg->hWin, ID_WIDGET);
my_widget = my_value;
```

Il porting che è stato fatto di questa libreria per questo processore dipende dalla board di sviluppo che si vuole considerare (soprattutto per il nostro LCD TouchScreen), nel nostro caso dipende in maniera esclusiva da μEZ . Infatti per questioni di modularità, emWin si appoggia alle API di μEZ , in modo da sfruttarle senza interfacciarsi direttamente con l'hardware sottostante, essendo già state ottimizzate le API di μEZ (quindi emWin nel nostro caso non può funzionare senza di μEZ).

3.2 Dispositivi Slave

Questi dispositivi non hanno una parte software che funge da scheletro per l'applicazione, essenzialmente per due motivi:

- le architetture sono troppo semplici, non è necessario interfacciarsi con l'hardware tramite API visto che i comandi che devono eseguire sono diretti alle diverse periferiche;
- non è presente molta memoria e non è garantita molta capacità di calcolo, quindi sarebbe svantaggioso utilizzare qualcosa di più "astratto" (e quindi inefficiente) per gestire le operazioni.

L'unica eccezione di questo argomento è l'utilizzo, per il dispositivo ARM, di un sistema operativo. Per il dispositivo PIC non è stato possibile inserire il sistema operativo, in quanto non è presente abbastanza memoria e il sistema di calcolo è troppo semplice.

3.2.1 FreeRTOS

Il sistema operativo utilizzato sul dispositivo di tipo ARM Cortex-M3 si chiama FreeRTOS.

È un sistema operativo di tipo Real Time, ossia il sistema deve essere "prevedibile" o piuttosto "determinista", nel senso che nel sistema si possa conoscere il tempismo reale (nei migliori o peggiori dei casi) di un determinato processo o elaborazione. In pratica un sistema real-time deve garantire che una elaborazione (o task) termini entro un dato vincolo temporale o scadenza (detta in gergo *deadline*). Per garantire questo è richiesto che la schedulazione delle operazioni sia fattibile. Il concetto di fattibilità di schedulazione è alla base della teoria dei sistemi real-time ed è quello che ci permette di dire se un insieme di task sia eseguibile o meno in funzione dei vincoli temporali dati.

Il sistema operativo non è stato utilizzato per utilizzare una HAL per programmare, ma per creare dei task separati per gestire la parte di protocollo e la parte di applicazione, in quanto continui polling sui sensori possono disturbare l'invio della risposta al dispositivo Master.

Capitolo 4

Modello di sistema

Di seguito si riporta il modello del sistema secondo lo standard ingegneristico, ossia presentando:

- “*Use Case Diagram*”, ossia la presentazione ad alto livello dei diversi attori del sistema (nel nostro caso gli attori sono: l’utente, il dispositivo Master e il dispositivo Slave), in modo da rendere chiari i ruoli degli attori e le loro azioni all’interno dello scenario;
- “*Sequence Diagram*”, ossia la presentazione ad alto livello dello scambio di “messaggi” tra i diversi attori e dispositivi del sistema (vengono identificate le interazioni nel sistema come messaggi);

4.1 Use Case Diagram

Con questo tipo di diagramma, viene presentata una tabella per ogni attore, ciascuna contenente nell'intestazione il nome dell'attore considerato e all'interno le rispettive azioni. Ogni attore coinvolto nel sistema ha delle precondizioni da rispettare affinché tutte le operazioni vadano a buon fine e una postcondizione. Vengono presentate infine le azioni sotto forma di algoritmo o insieme di attività da svolgere per eseguire un task.

L'attore principale, cioè l'utente, ha tre tipi di attività, ossia la scelta del task da far eseguire al sistema, l'avvio di esecuzione di tale task (premendo l'opportuno tasto nell'interfaccia) e l'osservazione dei valori che vengono mostrati nella GUI.

L'altro attore è il dispositivo Master, che ha il compito di mostrare ed aggiornare l'interfaccia grafica e di interrogare il dispositivo Slave in base alla scelta dell'utente. Questo attore ha quindi queste attività: mostrare l'interfaccia grafica, avviare i task scelti dall'utente e inviare i messaggi ai dispositivi Slave.

Gli ultimi attori da considerare sono i dispositivi Slave, che hanno il compito di eseguire polling sui sensori a loro associati e di rispondere alle richieste Modbus del dispositivo Master. Le loro attività quindi sono: controllo sensori e ricezione e risposta di messaggi Modbus.

Attore	Utente
Precondizione	Il sistema è avviato e mostra la GUI a seguito del logo μEZ , tutti i tasti sono selezionabili e tutti i dispositivi sono avviati.
Postcondizione	L'utente è munito di penna specifica per interfacciarsi col dispositivo TouchScreen.
Scelta task	<ol style="list-style-type: none"> 1. interfaccia gestione LED 2. interfaccia monitoraggio sensori 3. interfaccia di debug
Esecuzione task	<ol style="list-style-type: none"> 1. accendere o spegnere luci 2. controllo sensori 3. attività di debug
Osserva valori	<ol style="list-style-type: none"> 1. valutazione valori mostrati dalla GUI

Attore	Master
Precondizione	Il sistema è avviato.
Postcondizione	L'utente esegue scelte.
Mostra GUI	1. mostra schermata principale
Avvia task	1. ottieni scelta dall'utente
Invia messaggi	1. messaggi di tipo read 2. messaggi di tipo write
Attore	Slave
Precondizione	Il sistema è avviato.
Postcondizione	I sensori rispondono e si ricevono correttamente messaggi Modbus.
Controllo	1. esegue polling sui sensori 2. aggiorna il valore dei registri col valore ottenuto dai sensori
Ricezione	1. riceve messaggi Modbus 2. invia o setta i registri richiesti

4.2 Sequence Diagram

Questo diagramma rappresenta una generica interazione tra utente e sistema. Ad ogni scelta dell'utente, visualizzabile come un messaggio (ovviamente non testuale) inviato dall'utente al dispositivo Master, viene trasformata in messaggio Modbus da inviare al corrispondente dispositivo Slave. Il dispositivo Slave, prima di essere interrogato via Modbus, continua ad eseguire polling sui sensori a lui collegati, per tenersi aggiornato sul loro valore. Al momento della richiesta Modbus, il dispositivo Slave termina momentaneamente di eseguire polling sui sensori e invia l'ultimo valore acquisito da questi al dispositivo Master. Quest'ultimo, appena riceve i valori, aggiorna l'interfaccia grafica e mostra all'utente il risultato dell'interrogazione. Fatto ciò, il dispositivo Master rimane in attesa di ordini dall'utente, mentre il dispositivo Slave continua a fare polling sui sensori a lui collegati.

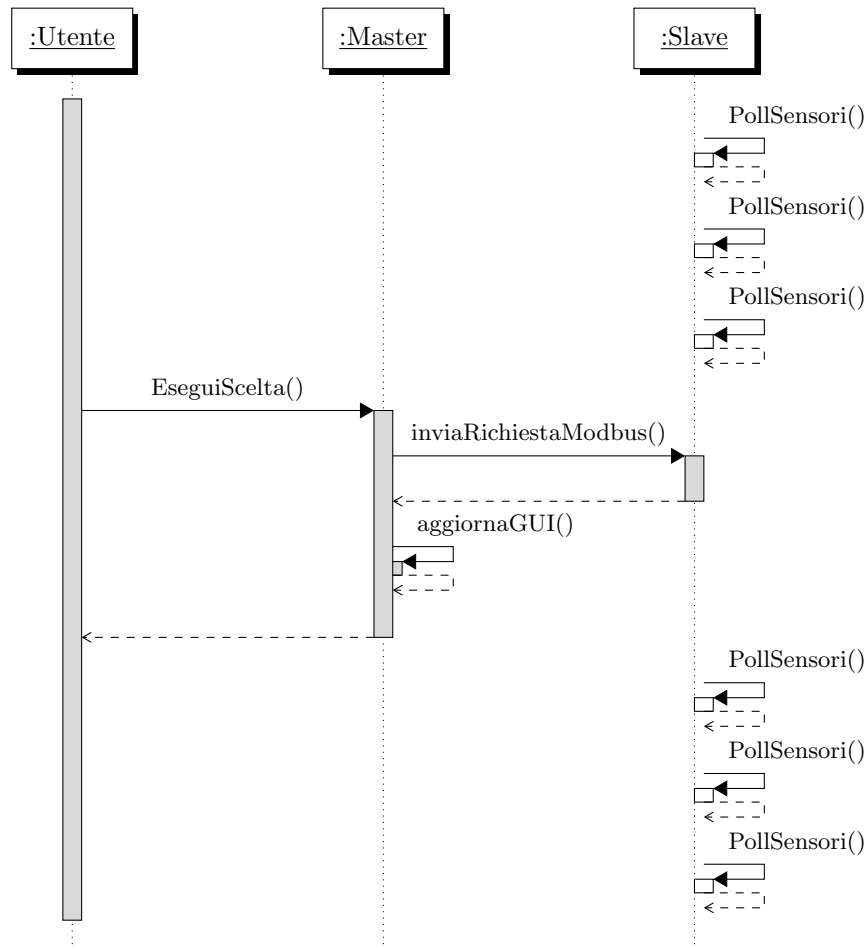


Figura 4.1: Esempio di attività generica con il sistema.

Capitolo 5

Implementazione del sistema

5.1 Protocollo Modbus

Modbus è un protocollo semplice di tipo master-slave, cioè esiste un dispositivo che invia una richiesta e i diversi slave eseguono la richiesta dando una risposta al master.

È stato creato in modo da far comunicare un master con fino a 256 slave, inviando un messaggio in cui si specifica l'indirizzo del destinatario slave. Quando uno slave riceve un messaggio, controlla se gli appartiene, altrimenti lo scarta; questo deve avvenire perché la comunicazione seriale non necessita di apparecchiature che indirizzano il messaggio al giusto indirizzo, ma la comunicazione è garantita da fili che collegano i diversi dispositivi seriale.

5.1.1 Tipi di messaggi

Il messaggio tipico del protocollo è formato da questi campi:

1. l'indirizzo del dispositivo con cui il master ha stabilito la transazione, l'indirizzo 0 corrisponde ad un messaggio broadcast inviato a tutti i dispositivi slave (1 byte);
2. il codice della funzione che deve essere o è stata eseguita (1 byte);
3. i dati che devono essere scambiati tra i due dispositivi (2 byte · n dati);
4. il controllo d'errore composto secondo l'algoritmo CRC16 (2 byte).

Se un dispositivo individua un errore nel messaggio ricevuto (di formato, di parità o nel CRC16) il messaggio viene considerato non valido e scartato, uno slave che rilevi un errore nel messaggio quindi non eseguirà l'azione e non risponderà alla domanda, così come se l'indirizzo non corrisponde ad un dispositivo in linea.

5.1.2 Registri

Il protocollo Modbus permette la comunicazione tra master e slave tramite diversi tipi di variabili chiamate "registri". In base al tipo di registro, sono permesse diverse operazioni:

- i registri di "input" discreti rappresentano dei booleani (registri a bit singolo) che possono essere solo letti, quindi in questi può essere eseguita una funzione di tipo read;
- i registri di "input discreti" hanno lo stesso funzionamento, in generale il master può solo leggere i registri di questo tipo;

- i registri di "input" (diversi da input discreti) rappresentano una word a 16 bit;
- i registri di coil rappresentano un booleano che può essere sia letto che scritto dal master;
- i registri holding rappresentano una word di 16 bit che può essere sia letta sia scritta.

Ogni registro può essere quindi associato ad una variabile di stessa grandezza o che rappresenta una grandezza simile (facendo un opportuno cast).

5.1.3 Tipi di richiesta

Modbus ha diversi tipi di richiesta, ognuna riguardante un registro specifico:

- 01 Read Coil Status
- 02 Read Input Status
- 03 Read Holding Registers
- 04 Read Input registers
- 05 Force Single Coil
- 06 Prese Single register
- 07 Read Status
- 15 Force multiple Coils
- 16 Preset Multiple Registers

Viene riportata di seguito la descrizione dettagliata delle funzioni più utilizzate:

- Read Output Status (01): permette di richiedere lo stato ON o OFF di variabili logiche binarie (registri output R/W). La modalità broadcast non è permessa.
- Read Input Status (02): è operativamente identica alla precedente ma riguarda i registri input.
- Read Output Registers (03): permette di richiedere il valore di registri a 16 bit (word) contenenti variabili numeriche (registri output R/W). La modalità broadcast non è permessa.
- Read Input Registers (04): è operativamente identica alla precedente ma riguarda i registri Input.
- Force Single Coil (05): permette di forzare lo stato di una singola variabile binaria ON o OFF. La modalità broadcast è permessa.
- Preset Single Register (06): permette di impostare il valore di un singolo registro a 16 bit. La modalità broadcast è permessa.
- Force Multiple Coils (15): permette di forzare lo stato di ciascuna variabile binaria in un blocco consecutivo. La modalità broadcast è permessa. Preset Multiple Registers (16): permette di impostare il valore di un blocco consecutivo di registri a 16 bit. La modalità broadcast è permessa.

5.1.4 CRC16

Gli ultimi due caratteri del messaggio contengono il codice di ridondanza ciclica (Cyclic Redundancy Check) calcolato secondo l'algoritmo CRC16.

Per il calcolo di questi due caratteri il messaggio (indirizzo, codice funzione e dati scartando i bit di start, stop e l'eventuale parità) viene considerato come un unico numero binario "continuo" di cui il bit più significativo (MSB) viene trasmesso prima. La procedura passo-passo per il calcolo del CRC16 è la seguente:

1. Caricare un registro a 16 bit con FFFFh (tutti i bit a 1).
2. Fare l'OR esclusivo del primo carattere con il byte superiore del registro, porre il risultato nel registro.
3. Spostare il registro a destra di un bit.
4. Se il bit uscito a destra dal registro (flag) è un 1, fare l'OR esclusivo del polinomio generatore 1010000000000001 con il registro.
5. Ripetere per 8 volte i passi 3 e 4.
6. Fare l'OR esclusivo del carattere successivo con il byte superiore del registro, porre il risultato nel registro.
7. Ripetere i passi da 3 a 6 per tutti i caratteri del messaggio.
8. Il contenuto del registro a 16 bit è il codice di ridondanza CRC che deve essere aggiunto al messaggio

5.1.5 Sincronizzazione

La sincronizzazione del messaggio tra trasmettitore e ricevitore viene ottenuta interponendo una pausa tra i messaggi pari ad almeno 3.5 volte il tempo di un carattere. Se il dispositivo ricevente non riceve per un tempo di 3,5 caratteri, ritiene completato il messaggio precedente e considera che il successivo byte ricevuto sarà il primo di un nuovo messaggio e quindi un indirizzo.

5.2 Diagramma dei componenti

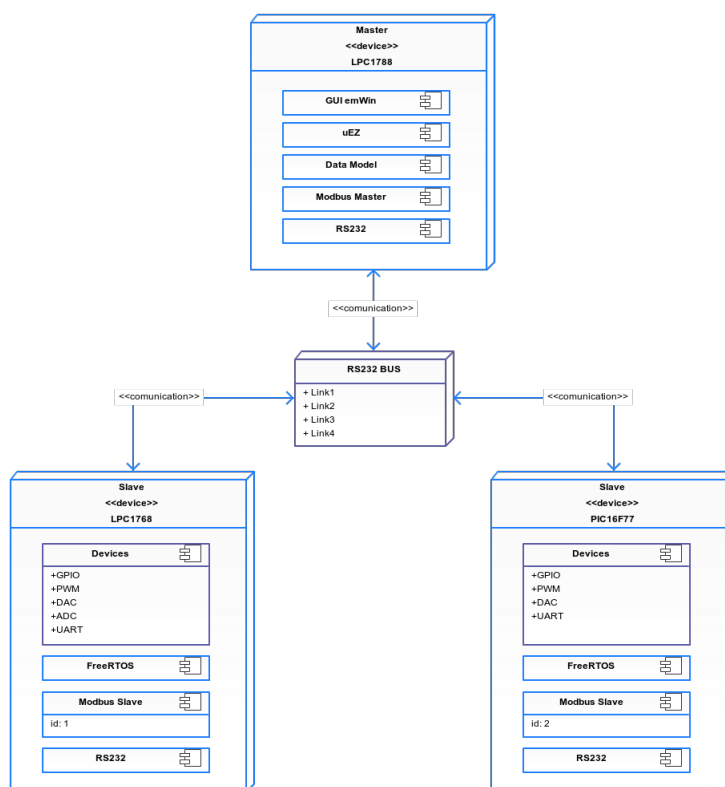


Figura 5.1: architettura del sistema uEZ