Università degli Studi di Salerno

Dipartimento di Ingegneria dell'Informazione, Ingegneria Elettrica e Matematica Applicata

Elaborato finale in
INGEGNERIA INFORMATICA

Progettazione e realizzazione di un'applicazione embedded per il controllo e la gestione degli accessi mediante RFID

Relatore:
Prof. Pasquale Foggia

Candidato:
Salvatore Napolitano
Matricola: 0612700038

Anno Accademico 2016/2017

“A volte dimostra più forza una persona che insegue il suo sogno di una che lo afferra.”

-Anonimo-

Introduzione

Negli ultimi anni lo sviluppo "Embedded" si è diffuso in tutto il mondo tra hobbisti, artisti, appassionati di elettronica e programmazione, finanche inesperti. Si sono formate comunità di sviluppatori che condividono le proprie idee e i risultati dei propri lavori, mettendoli a disposizione di chiunque voglia replicarli, modificarli o migliorarli, nella filosofia dell' *"open-source".*

Con il ridursi dei costi e l'aumentare della capacità di computazione dei componenti elettronici sono proliferate piattaforme che permettono al principiante come all'ingegnere di sviluppare un'idea che trasversalmente taglia il mondo reale e quello virtuale. Una collisione tra due mondi che fino a poco tempo fa era accessibile esclusivamente a professionisti. Esistono oggetti che possono acquisire o estendere le funzionalità umane, che ci permettono di estendere la nostra percezione del mondo e di rivalutarne i suoi limiti. Esistono oggetti connessi alla *rete delle reti* che condividono ed elaborano dati per un nuovo utilizzo delle informazioni.

Con questa tesi si vuole andare ad esplorare l'applicazione delle nuove piattaforme dei sistemi embedded, tecnologia abbastanza matura eppure non ancora esplorata a fondo. Questo elaborato ha lo scopo in particolare di esaminare e discutere l'uso di sistemi embedded, tenendo conto anche del linguaggio di programmazione utilizzato.

Nel primo capitolo del presente lavoro introduce il lettore alla conoscenza dei sistemi embedded, presentandone la loro utilità e adattabilità nella realizzazione di sistemici informatici complessi.

Nel capitolo secondo viene illustrata la progettazione dell'intero sistema: proprietà e caratteristiche tecniche di ogni singolo componente, insieme al tipo di interconnessione che andrà a formare il sistema completo. Il tipo di board scelto per la progettazione è un Arduino ATMega2560 capace di supportare fino a 54 Pin digitali. Infatti, lo scenario nel quale si è idealmente collocato il sistema programmato è un albergo o un'abitazione privata, che mediante Tag RFID possono controllare e gestire gli accessi ad un determinato punto di controllo (stanza o abitazione).

Il capitolo terzo mostra la realizzazione e progettazione dell'applicazione embedded che permette il controllo e la gestione degli accessi di cui sopra mediante tecnologia RFID. Il sistema da me progettato, per poter essere utilizzato, necessita di un collegamento ad un sistema più ampio e complesso, denominato MASTER, che permette di gestire e ricevere dati sullo stato del punto di controllo utilizzando il protocollo di comunicazione MODBUS su linea seriale. Il software sviluppato permette così il controllo del punto di accesso tramite device connesso alla rete internet.

Indice

Capitolo 1

Sistemi Embedded

1.1 Introduzione al sistema embedded

Il termine sistema embedded (sistema integrato o incorporato) identifica un sistema elettronico di elaborazione digitale *special purpose*, ovvero, dotato di un microprocessore progettato appositamente per una specifica funzione o compito, quindi non riprogrammabile dall'utente per altri scopi. Un esempio tipico di un sistema embedded sono le centraline elettroniche installate a bordo degli autoveicoli per il controllo dell'autovettura e dell'ABS.

Un sistema embedded si interfaccia con il mondo esterno attraverso una serie di sensori e attuatori (led, rilevatori presenza, relè, bottoni). Alcune caratteristiche comuni tra questi sistemi è la reattività in *real time*, ovvero rispondere in tempi "quasi" reali a sollecitazioni provenienti dal mondo esterno. La stretta relazione tra un sistema embedded ed il mondo fisico con cui si trova ad interagire pone la progettazione in mano alla persona che più conosce e comprende quel mondo. I sistemi special purpose, a differenza dei sistemi general purpose (ad esempio un computer), eseguono uno specifico programma ripetutamente. Sono quindi progettati per eseguire un'applicazione specifica, minimizzando le risorse utilizzate e massimizzando la robustezza.

1.1.1 Caratteristiche di un sistema embedded

Un sistema embedded per svolgere un compito specifico deve possedere alcune caratteristiche. Le principali caratteristiche di un sistema di questo genere si possono raggruppare in tre sezioni e sono: *efficienza, affidabilità e reattività*.

1.1.2 Efficienza

Quando si parla di efficienza nell'ambito dello sviluppo software, solitamente, si intende l'efficienza di un codice. L'obiettivo è costruire un software con un costo computazionale il più basso possibile. Nei sistemi embedded questa è solo una delle caratteristiche legate all'efficienza. Un altro aspetto molto importante è la dimensione: trattandosi di sistemi a specifico compito, le risorse di utilizzo sono limitate, e di conseguenza non è possibile scrivere del codice che occupa più spazio della capacità massima di memorizzazione. Altra caratteristica legata all'efficienza è il peso del dispositivo (similmente ad uno smartphone il cui peso eccessivi verrebbe a inficiarne la portabilità). Come ultima, ma non meno importante caratteristica vi è l'efficienza in termini energetici. Un dispositivo che richiede una ricarica continua potrebbe risultare poco pratico. Un esempio è il peacemaker dove le caratteristiche principali sono sicuramente l'efficienza energetica ed il peso: trovandosi all'interno del corpo umano non può essere molto pesante. Dal punto di vista energetico, la batteria deve durare il più possibile, in modo da evitare che il paziente sia sottoposto a continui interventi.

1.1.3 Affidabilità

Come indicato nel capitolo precedente, alcune applicazioni dei sistemi embedded includono sistemi critici (si pensi ad esempio il settore bio-medico, o in ambito dell'industria meccanica, dai peacemaker agli aerei). Per questi tipi di sistemi sono richieste le seguenti caratteristiche: *reliability, availability, safety e security*.
Per *reliability* si intende la capacità di un sistema di svolgere un compito o una funzione per un determinato periodo di tempo T. La probabilità che un sistema fallisca in un dato intervallo di tempo, è espresso dal tasso di fallimento, misurato in FIT (Failure In Time). L'inverso del FIT, è chiamato MTTF (Mean Time To Failure). Ora se il tasso di fallimento avrà un valore di 10^{-9}guasti/ora o inferiore, allora si parlerà di sistema con requisito di affidabilità molto elevata.
Con il termine *availability* si intende la percentuale di tempo in cui il sistema rimane funzionante. Questa caratteristica è strettamente legata al MTTF (Mean Time To Failure) e alla manutenibilità MMTR (Mean Time To Repair). La percentuale di *availability* è data da: MTTF/(MTTF+MMTR).
Con il termine *safety* si intende la probabilità che il sistema non rechi danni agli utenti o all'ambiente in cui è inserito.

Per concludere, con il termine *security* si intende la capacità di garantire l'autenticità, l'integrità delle informazione e di negare l'accesso ai servizi e alle informazioni da parte di chi non è autorizzato. Negli ultimi anni, la sicurezza ha assunto un ruolo fondamentale nei sistemi *real-time* e in internet, poiché la violazione di un sistema potrebbe causare danni ai dati e all'ambiente in cui esso è immerso.

1.1.4 Reattività

Spesso i sistemi embedded sono utilizzati in contesti dove devono prontamente reagire a stimoli che provengono dall'ambiente. È necessario quindi eseguire elaborazioni ed eventualmente azioni in *real-time*, senza ritardi. I sistemi legati al vincolo della reattività si possono suddividere in sistemi hard *real-time* e sistemi soft *real-time*.

Per sistemi hard *real-time* si intendono tutti quei sistemi che devono svolgere assolutamente un compito o una funzione in un determinato periodo di tempo. L'eventuale violazione di *deadline* può risultare critica per l'intero sistema.

Come intuibile, in un sistema soft *real-time*, la violazione di una *deadline* può essere problematica, ma senza causare delle criticità all'interno del sistema. Un esempio di sistema hard *real-time* può essere un sistema che si occupa della messa in sicurezza di un edificio in caso di incendio. Il sistema, in caso di incendio, deve provvedere alla chiusura automatica delle porte anti-incendio nella sezione dove è avvenuto l'incendio, chiudendo le porte in un tempo T. Se la chiusura delle porte non avvenisse entro il tempo prestabilito, l'incendio potrebbe propagarsi nel resto dell'edificio.

1.2 Tipi di logica

1.2.1 Logica programmata e cablata

I sistemi possono essere sia a logica programmata che cablata: nel primo caso i sistemi utilizzano microcontrollori e software a supporto delle apparecchiature elettro-meccaniche, mentre per quelli a logica cablata le funzionalità vengono ottenute tramite collegamenti delle apparecchiature elettro-meccaniche in modo da realizzare la logica desiderata. Il vantaggio di utilizzare un sistema a logica cablata è la velocità, che lo rende adatto a sistemi di piccole dimensioni, lo svantaggio è nella flessibilità (per cambiare sistema dovrò cambiare sia i collegamenti che componenti).

I sistemi a logica programmata sono più lenti ma molto flessibili (basta cambiare il programma). I microcontrollori permettono anche numerose funzionalità in più (threads, timer, ecc.) che consentono di creare sistemi molto più complessi, come i sistemi di visione artificiale, aggiungendo e togliendo moduli che ovviamente risultano molto costosi per sistemi di piccole dimensioni.

1.2.2 Sistemi Operativi

Un sistema embedded si troverà inserito in un ambiente non noto a priori e avrà bisogno dal suo sistema operativo di funzionalità che generalmente, in un sistema desktop, non vengono considerate prioritarie.
Tuttavia il sistema embedded userà altre funzioni del sistema operativo diverse rispetto ad un pc desktop.

Con la proliferazione di schede economiche per lo sviluppo di sistemi embedded ed il diffondersi degli smartphones i sistemi operativi si sono specializzati offrendo versioni che tengono in conto la scarsità delle risorse e l'autonomia dell'applicativo. Queste nuove versioni dei sistemi operativi tendono a rendere più reattivo il kernel specializzandolo con funzionalità *real-time.*

1.2.3 Tipologie dei sistemi

Le tipologie dei sistemi embedded sono generalmente tre:

- Microcontrollore: singolo chip che esegue il programma. Prevede una CPU limitata, RAM piccola, storage persistente, I/O analogici e digitali, clock. Spesso la memoria dati viene separata da quella delle istruzioni per problemi di parallelismo (8 bit per le istruzioni e 14 bit per i dati).
- System on Chip: dotato di CPU *general purpose* (risolve problemi in generale, es: il Personal Computer che non risponde a una particolare esigenza ma si adatta a quella di chi lo utilizza), il DSP (Digital Signal Processing) e interfacce di comunicazione (USB e ETHERNET). Vengono utilizzati con sistemi operativi specifici.
- Single Board Computer: scheda che contiene tutti i componenti di un sistema di elaborazione. A differenza di un sistema SOC (System on Chip) è molto più modulare (ram espandibile, hard disk di supporto, ecc.) e più compatto di una scheda madre, con la possibilità di avere CPU anche a 64 bit. Vengono utilizzati con sistemi operativi specifici o tradizionali.

1.2.4 Architettura Hardware Sistemi embedded

In generale, l'architettura di un elaboratore è data da una CPU, una memoria centrale e da un insieme di device controller connessi attraverso un bus comune.
Come specificato nell'introduzione del presente capitolo, un sistema embedded è progettato per svolgere uno specifico compito, permettendo di ridurre le risorse utilizzate. La maggior parte di questi sistemi hanno dimensioni ridotte e non è possibile utilizzare l'hardware di un elaboratore. Per i sistemi embedded si utilizzano solitamente i microcontrollori o i SOC (System On a Chip).
I microcontrollori sono dei dispositivi elettronici, nati come una evoluzione nonché una alternativa ai microprocessori. Essi integrano su un singolo chip un sistema di componenti che permette di avere la massima autosufficienza funzionale per applicazioni embedded. Un microcontrollore è composto da un processore, una memoria permanente, una memoria volatile e dei canali di I/O a cui è possibile collegare sensori/attuatori. Inoltre, è dotato di un gestore di interrupt ed eventualmente altri blocchi specializzati.
I SOC, a differenza dei microcontrollori, integrano in un unico chip un sistema completo, dotato solitamente di: microprocessore, RAM, circuiteria di I/O, sottosistema video.
La scelta tra microcontrollore e SOC dipende sostanzialmente dai requisiti del progetto.

1.2.5 General purpose I/O pin

I microprocessori e i microcontrollori utilizzati in ambito embedded, contengono usualmente un certo numero di pin, che possono essere utilizzati direttamente per gestire in I/O e anche i loro controller. I pin sono generalmente *general purpose*, nel senso che possono essere programmati per fungere da input o da output, a seconda delle necessità. Inoltre, i pin possono essere digitali o analogici.
Un sistema di elaborazione, per poter elaborare un segnale analogico, deve prima convertirlo. Tale conversione avviene mediante un convertitore analogico-digitale che mappa il valore continuo in un valore discreto in un certo range. Il numero di bit utilizzati dal convertitore rappresenta la risoluzione del convertitore e ne determina la precisione.

1.2.6 Bus e protocolli

I protocolli di scambio informazioni mediante singoli pin, possono diventare molto complessi, a seconda dei dispositivi di I/O con cui interfacciarsi.
Per facilitare la comunicazione sono stati introdotti nel tempo varie interfacce e protocolli, che possono essere classificati in *seriali e parallele*.
Le interfacce *parallele* permettono di trasferire più bit contemporaneamente, utilizzando un bus a 8,16 o più fili.
Le interfacce *seriali* permettono di inviare i bit in uno *stream* sequenziale, bit per bit, utilizzando un solo filo o comunque un numero ridotto. Tali interfacce possono essere suddivise in due categorie: *sincrone e asincrone*. In modalità *asincrona*, il trasmettitore e il ricevitore si sincronizzano utilizzando i dati stessi. Il trasmettitore invia un bit di ″partenza″, successivamente il dato vero e proprio e, infine, un bit di ″stop″. In modalità *sincrona*, la trasmissione dei dati è sincronizzata con il clock del microcontrollore/microprocessore.

Capitolo 2

Progetto

2.1 Introduzione

Il progetto proposto come oggetto del tirocinio riguarda la creazione di un software su board *Arduino* destinato ad interpretare, elaborare e gestire il controllo di un punto specifico da definirsi (ad es. la stanza di un albergo, o un'abitazione). Tramite device connesso al web, in particolare, dovrà essere in grado di *leggere/ elaborare/scriv*ere la memoria di card *RFID (*Radio Frequency Identification), inviare i dati e lo stato dell'abitazione ad un componente (*PC o board*) denominato *MASTER* attraverso il protocollo di comunicazione *MODBUS* su linea seriale. Tale applicazione è esemplificativa, ma non esaustiva rispetto alla varietà di possibili realizzazioni che il sistema embedded da me progettato offre.

Lo scopo della progettazione specifica da me scelto, fa' del mio sistema embedded un ausilio allo sviluppo della domotica, reso utilizzabile mediante una applicazione per board *arduino* in grado di effettuare una serie di operazioni successive alla lettura di una *card/tag*.

2.1.1 Schema e scenario tipico

Lo scenario tipico analizzato è stato quello di un Albergo con un numero indefinito di stanze, nel quale ogni locale sarà dotato di una board *Arduino* composta da due lettori *RFID* e tre *LED*. I clienti e il personale avranno a disposizione i *TAG RFID* mentre l'albergatore provvisto di una card *RFID* denominata *MASTER*, con una memoria *EEPROM* da *1KB* in cui salverà rispettivamente *ID* univoco del *TAG* e numero di stanza che aprirà. La card *MASTER* verrà caricata con uno *sketch* per arduino, che consiste nella lettura di un *TAG* e l'inserimento del numero di stanza corrispondente (sono previsti più *TAG/ID* per la stessa stanza). Ogni board *Arduino* inoltre sarà inizializzata con il numero di stanza e l'*ID* della card *MASTER*, salvati rispettivamente nelle ultime locazioni in memoria *EEPROM* come singoli *BYTE*, caricati nella fase di avvio del sistema in una struttura dati interna alla board.

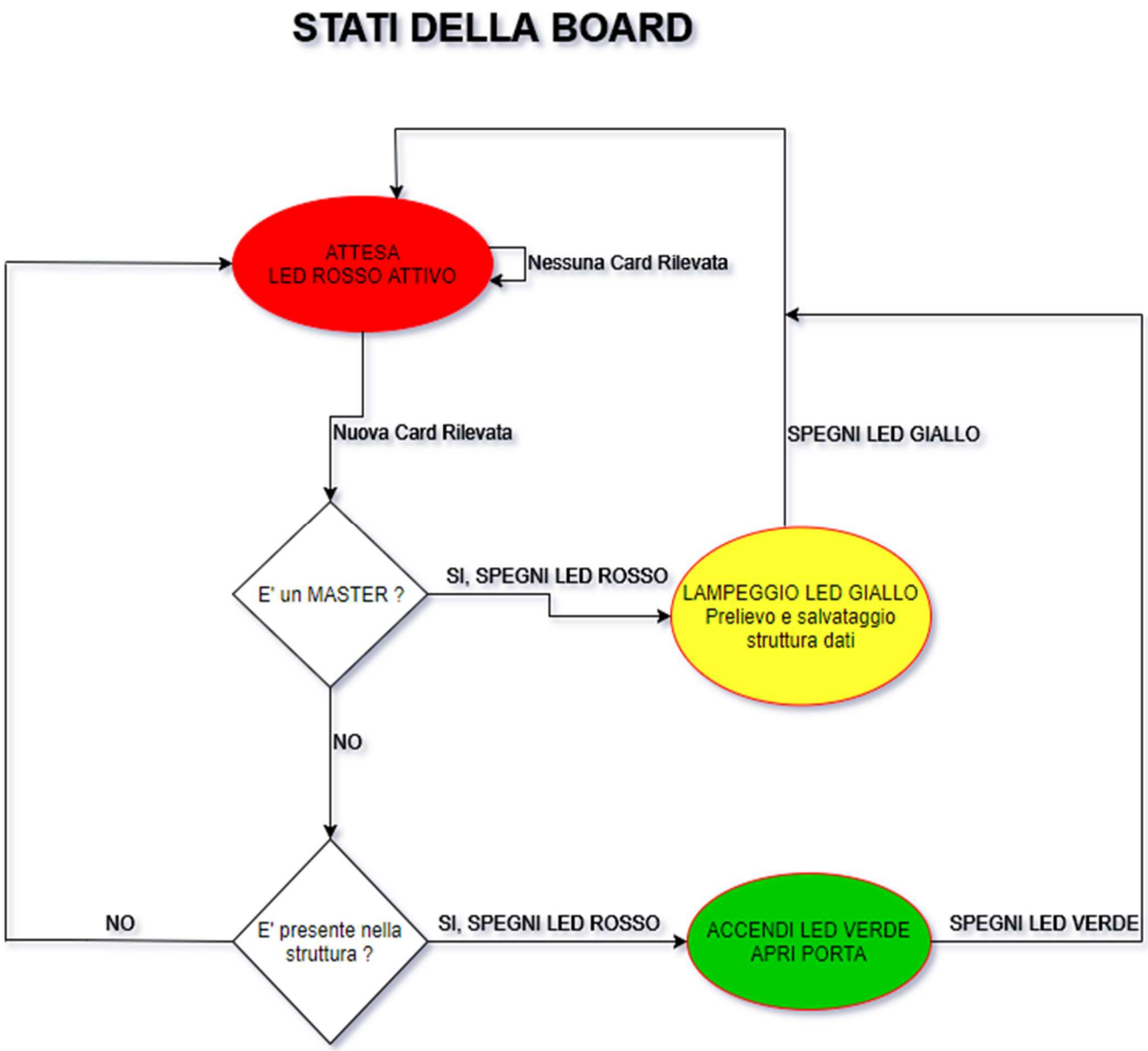
Al passaggio della card *MASTER* sul lettore esterno l'*ID* verrà letto, si effettuerà un *check* con quello presente nella struttura dati, se corrispondente, la board passerà dallo stato di attesa (*LED* rosso attivo) a quello di programmazione (*LED* rosso spento, *LED* giallo lampeggiante) dove gli *ID* verranno scaricati e filtrati in base al numero di stanza (se ad es. il numero della stanza è 108, verranno prelevati solo gli *ID* che aprono la stanza 108). Infine, gli ID saranno memorizzati in modo permanente nella *EEPROM* della board.

Al passaggio di un *TAG RFID*, se presente all'interno della memoria, verrà abilitata l'apertura della porta e la board passerà dallo stato di attesa a quello di apertura (*LED* verde *ACCESO* per un 1 sec), per poi ritornare nello stato di attesa. Una volta entrati in stanza, un secondo lettore, collegato sempre alla stessa board, svolgerà la funzione di "*presenza in stanza*". Fintanto che il *TAG* rimarrà appoggiato sul lettore, la board resterà nello stato di presenza in stanza (*LED* verde sempre acceso), per questo motivo sarà collegato in parallelo al secondo lettore, così anche se presente una persona in stanza, il lettore esterno non smetterà di funzionare e la board continuerà ad eseguire operazioni. Nel momento in cui il *TAG* viene meno, la board ritornerà nello stato di attesa e tutte le luci verranno spente.

2.1.2 Problema da risolvere e definizione specifiche

Nella *prima fase* della progettazione sono state definite alcune specifiche, dopo una serie di osservazioni per la tecnologia disponibile:

1. ***Ambiente di collocazione della schedina*** : Albergo o casa.
2. ***Strumenti a disposizione:*** *3 tag,2 card,2 lettori RFID e MODBUS RTU.*
3. ***Grado di automazione della board*** : Medio-Alto. Si è utilizzato un grado di automazione che prevede la programmazione automatica della schedina al passaggio della card *MASTER*, in modo da limitare quanto più è possibile lo smontaggio dall'ambiente di lavoro per la programmazione attraverso gli sketch. Nonostante ciò è richiesta lo stesso una fase di inizializzazione del progetto.
4. ***Grado di interazione con il mondo esterno:*** Si è optato per un grado semplice stile user-friendly. La schedina prevede 3 *LED* (giallo, rosso e verde) che si illuminano a seconda dello stato in cui si trova la board:
    - Stato di Attesa → *LED* rosso sempre acceso, tutti gli altri spenti.
    - Stato di programmazione della *EEPROM* → *LED* giallo lampeggia, tutti gli altri spenti.
    - Stato di apertura porta → *LED* verde si accende per qualche ms, e tutti gli altri spenti.
    - Stato di presenza in stanza → *LED* verde sempre acceso fin tanto che il *tag* è sul lettore), tutti gli altri spenti.
    - Stato di errore → Tutti i led accesi per qualche ms. Poi la board ritorna nello stato di attesa.

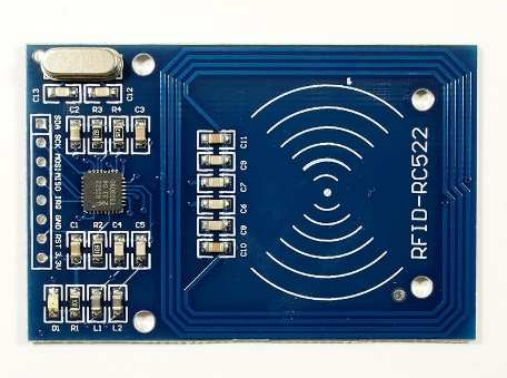


In questa fase si è concluso che era necessario una fase di inizializzazione del progetto, dove la board è programmata con alcuni dati in *EEPROM* indispensabili, come l'*ID* del *MASTER* e il numero di stanza.

2.2 Componenti utilizzati

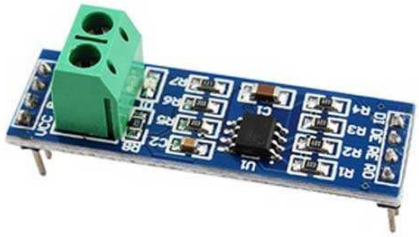
Elenco dei componenti:

1. *2xLettori RFID MFRC522 collegati in parallelo.*



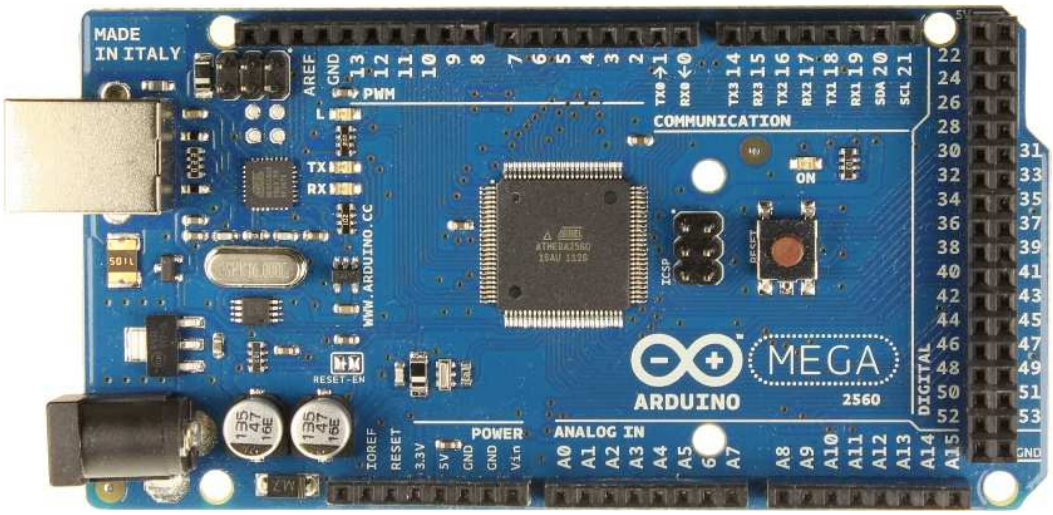
2. *3x LED (Giallo, Rosso, Verde).*

3. *MODBUS RS-485 Over Serial Line.*



4. *1xTAG RFID e 1xCARD RFID MIFARE Classic 1k MF1S503x*

5. Board Arduino ATMEGA 2560



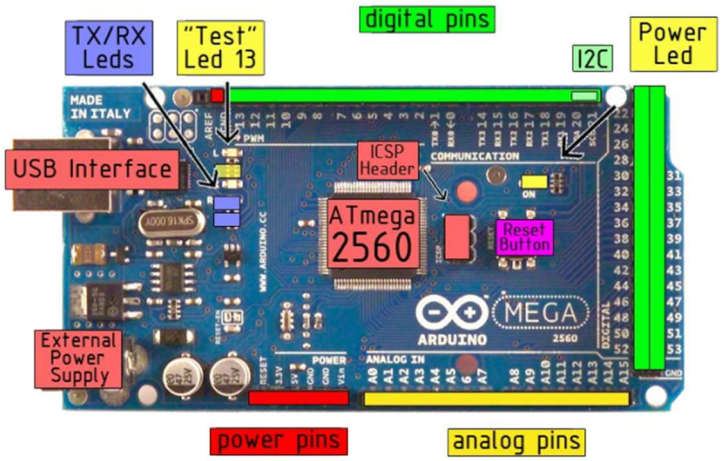
2.2.1 Arduino

Arduino è una piattaforma di sviluppo *open source* basata su microcontrollore. È composto da una piattaforma hardware sviluppata presso l'Interaction Design Institute, un istituto di formazione post-dottorale con sede a Ivrea, fondato da Olivetti e Telecom Italia. All'hardware viene affiancato un ambiente di sviluppo integrato multipiattaforma.
La scheda Arduino è in grado di interagire con l'ambiente in cui si trova, ricevendo informazioni dai sensori ed è in grado di modificarlo tramite l'utilizzo di attuatori.
Nel corso degli anni sono state immesse sul mercato varie tipologie di schede, con caratteristiche hardware simili, a seconda dello scopo specifico. Sono ora disponibili soluzioni nelle fasce: entry level, internet of things e wearable.

2.2.2 Hardware

La maggior parte delle schede Arduino consistono in un microcontrollore a 8 bit AVR prodotto da Atmel, con l'aggiunta di componenti complementari per facilitarne l'incorporazione in altri circuiti. In queste schede sono usati chip della serie megaAVR – nello specifico i modelli ATmega8, ATmega168, ATmega328, ATmega1280 e ATmega2560, a seconda del modello.
Per implementare il comportamento interattivo, tutti i modelli Arduino sono forniti di funzionalità di I/O, grazie alle quali essi ricevono i segnali raccolti da sensori esterni. In base a tali valori, il comportamento della scheda è gestito dal microcontrollore, in base alle decisioni determinate dal particolare programma in esecuzione in quel momento.
L'interazione con l'esterno avviene attraverso gli attuatori pilotati dal programma. Il numero di connettori di I/O a disposizione variano a seconda del modello. I connettori sono suddivisi in I/O digitale, di cui una parte possono produrre segnali in modulazione di frequenza PWM (Pulse-width Modulation) e in input analogici. Gli input analogici possono essere riprogrammati per funzionare come normali pin di I/O digitali. Per poter leggere i valori analogici, le schede Arduino sono dotate di un convertitore analogico-digitale.
Per favorire il caricamento del programma sul microcontrollore, le schede Arduino sono dotate di una porta USB. Integrano inoltre un chip per la conversione del segnale digitale da USB a seriale. Nei modelli più recenti, in particolare Arduino Leonardo e Arduino Esplora, tale chip non è presente, perché il microcontrollore integra già questa funzione. Oltre a supportare la comunicazione seriale, le schede Arduino sono in grado di supportare i protocolli di comunicazione I2C e SPI.

Molte schede includono un regolatore lineare di tensione a 5 Volt e un oscillatore a cristallo a 16 MHz.
L'alimentazione della scheda può avvenire tramite porta USB, attraverso un adattatore in corrente continua oppure utilizzando i pin Vin e GND.
Si sconsiglia quest'ultimo metodo in quanto un inversione dei poli potrebbe danneggiare la scheda.



Architettura Arduino ATmega2560

Caratteristiche Tecniche:

Microcontrollore: ATmega2560
Tensione di lavoro: 5V
Tensione input (raccomandata): 7-12V
Tensione input (limite): 6-20V
Pin I/O Digitali: 54 (di cui 14 forniscono segnali in PWM)
Pin input analogici: 16
Corrente DC per pin I/O: 40mA
Corrente DC per pin a 3.3V: 50mA
Memoria Flash: 256 KB dove 8 KB vengono utilizzati dal *bootloader*
SRAM: 8 KB
EEPROM: 4 KB
Frequenza Clock: 16 MHz

2.3 Software

2.3.1 Linguaggio di programmazione

Il *framework open-source* di riferimento per la programmazione di Arduino e altri microcontrollori è *Wiring*, un linguaggio di programmazione derivato da *C/C++*. Sebbene prima dell'avvenuta di Arduino la programmazione di un microcontrollore avveniva tramite i classici linguaggio a basso livello, come *C* o *Assembly*, essa richiedeva la conoscenza dei principali concetti di elettronica, come gli interrupt e le porte logiche. Con l'utilizzo di *Wiring* si è semplificato tutto questo permettendo di programmare il microcontrollore facilmente, attraverso un ambiente *C/C++* appositamente realizzato per questo tipo di sistemi.

I programmi realizzati per Arduino vengono chiamati in gergo "*sketch*".

Durante la realizzazione di un programma per Arduino, al programmatore viene richiesto di definire almeno due metodi principali:

void setup()

Funzione invocata una sola volta all'inizio di un programma o al reset della board. Generalmente utilizzata per i settaggi iniziali, come l'inizializzazione di un pin (input o output), di una connessione seriale, ecc...

void loop()

Funzione invocata ripetutamente, la cui esecuzione si interrompe solo con il reset della board o lo spegnimento.

2.3.2 Bootloader

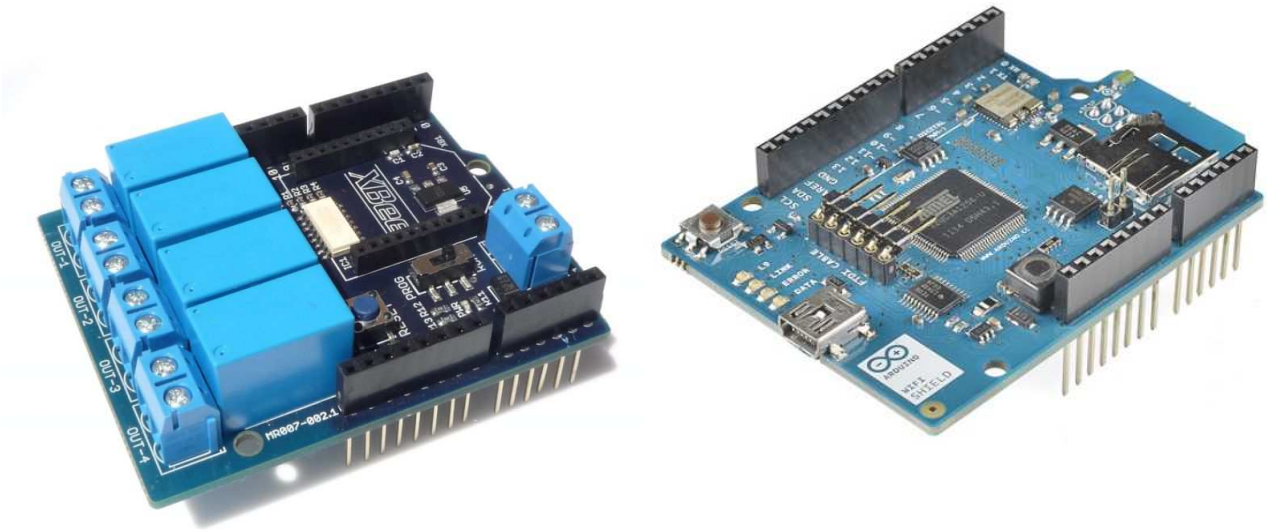
Il *bootloader* è un firmware che i progettisti di Arduino hanno inserito nei chip, il cui scopo è quello di caricare gli sketch nel microprocessore senza l'ausilio del programmatore. Se si desidera costruire una propria scheda, è necessario definire un proprio *bootloader*, in modo da evitare l'utilizzo di un hardware esterno per il caricamento degli sketch.
Quando una scheda Arduino è avviata, il *bootloader* viene eseguito, ritardando di pochi secondi l'avvio di un eventuale sketch presente in memoria.
All'avvio il *bootloader* verifica la presenza di comunicazioni inerenti ad eventuali sketch da caricare. In caso di uno sketch da caricare, il *bootloader* effettuerà il trasferimento dello sketch sulla memoria del microcontrollore.
Una volta terminato il trasferimento, il *bootloader* avvierà lo *sketch* appena caricato. Nel caso in cui non ci fossero comunicazioni, il *bootloader* provvederà ad avviare l'ultimo *sketch* caricato in memoria.

2.3.3 Shield

Le *shield* non sono altro che schede compatibili con Arduino e, grazie ad esse, è possibile estendere le funzionalità che fornisce il modello base della scheda.

Solitamente le *shield* hanno un basso assorbimento tanto che i progetti possono funzionare tramite la sola alimentazione fornita dalla porta *USB* (5V a 0,5mV). Le *Shield* quindi permettono di interfacciare le nostre piattaforme elettroniche ottimizzando gli spazi grazie al concetto di modularità.

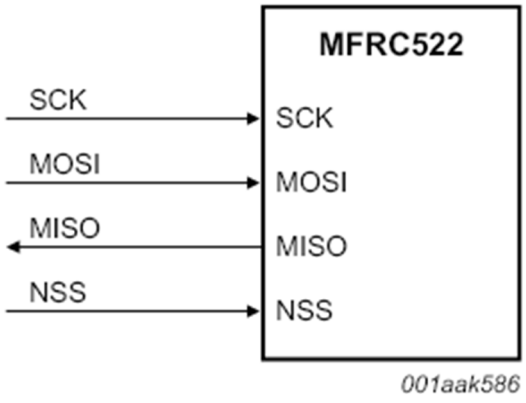
Tramite *shield* è possibile connettere Arduino a una rete *WI-FI*, a una *GSM*, *Bluetooth*, controllare attuatori e relè in modo semplice, inviare messaggi o email, integrare sensori in *Arduino* e molto altro ancora.



A Sinistra una shield per il controllo dei relè. A destra una shield per il modulo WI-FI.

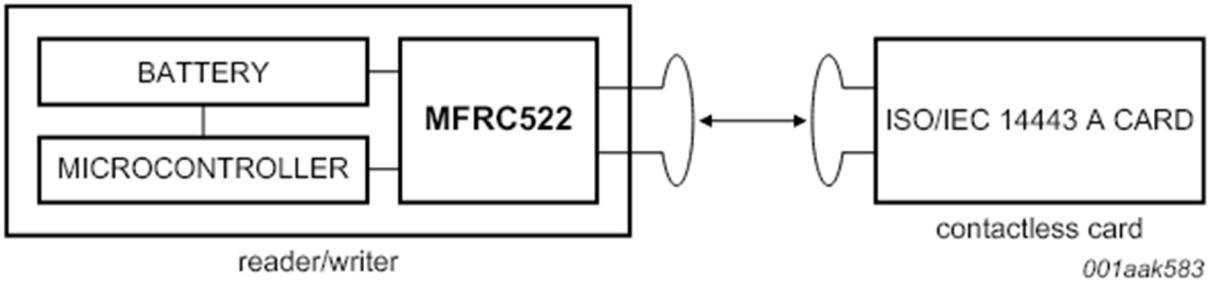
2.4 Lettore Mifare MFRC522 RFID Reader/Writer

Il lettore *RFID (Radio Frequency Identification)* è un lettore che consente di leggere i *tag* con identificazione a radio frequenza. Composto da microcontrollore e lettore card che utilizza la comunicazione tramite SPI (Serial Peripheral Interface) un protocollo di dati su seriale sincrono.
Il lettore card e i *tag* comunicano utilizzando un campo elettromagnetico a 13.56MHz (*ISO 14443A standard tag*). Il lettore emette un campo elettromagnetico che tramite il processo di induzione genera nell'antenna del *tag* una corrente che alimenta il chip. Il chip così alimentato comunica tutte le sue informazioni che vengono irradiate tramite l'antenna verso il lettore, che può anche scrivere i dati sul *tag.*



Nella foto lo schema per il collegamento ad un host.

Il modulo supporta la modalità di *lettura/scrittura* in accordo allo standard *ISO/IEC 14443 A/Mifare* dove sono definiti velocità di trasferimento e protocolli di comunicazioni.



Schema Modalità lettura/Scrittura.

La libreria nativa utilizzata, dove sono implementate le principali funzioni per la lettura, riconoscimento, scrittura e controllo dell'errore è stata prelevata da: https://github.com/miguelbalboa/rfid

Alcune funzioni sono state riviste, corrette e adattate allo scopo perché sprovviste di funzionalità specifiche, in particolare le funzioni di lettura, scrittura e autenticazione dei blocchi di memoria nei *tags*, insieme a quelle per il riconoscimento del tipo di *tag/card* (1KB,4KB ecc..).
I lettori adottano un sistema di anti-collisione intelligente per far sì che operino più di una *card/tag* contemporaneamente.
Quando due o più dispositivi tentano di trasmettere simultaneamente sullo stesso canale di comunicazione, avviene una collisione fra i dati, che non possono essere ricevuti correttamente. È necessario perciò che tutte le informazioni possano essere ricevute correttamente.

Esistono diversi *algoritmi* che possono essere applicati alle collisioni, in particolare, la collisione fra lettori avviene quando due o più trasmettitori fisicamente vicini tentano, nello stesso momento, di effettuare la lettura di un *tag*.

2.4.1 Algoritmi anticollisione

Gli *algoritmi* di anticollisione permettono a più lettori, che operano sulla stessa frequenza, di trasmettere e ricevere dati velocemente e senza perdite. Tra questi troviamo *LBT (Listen Before Talk).*

Nella tecnica utilizzata dall'algoritmo *LBT*, è necessario che il *reader* che desidera effettuare la trasmissione *ascolti il canale prescelto,* verificando che esso non sia già occupato da altri dispositivi.

Se è occupato, il *reader* attende, oppure tenta di trasmettere su un altro canale, scelto fra quelli disponibili.

E' possibile rendere l'algoritmo più efficiente imponendo un limite, al tempo massimo che il *reader* può tenere occupato un canale e al tempo minimo che deve attendere prima di rioccupare lo stesso canale, se disponibile.

2.5 MIFARE Classic Tag/Card MF1S503X

Sviluppate dalla *NXP* (*Next eXPerience*) *Semiconductors*, azienda di semiconduttori fondata dalla Philips, le MIFARE MF1S503x sono principalmente delle *smart card/tag contactless* (senza contatto) come definito nello standard *NFC TAG ISO/IEC 14443 Type A*.

Queste *card/tag* trovano applicazione nel mondo dei trasporti pubblici, biglietteria, uffici, ecc…

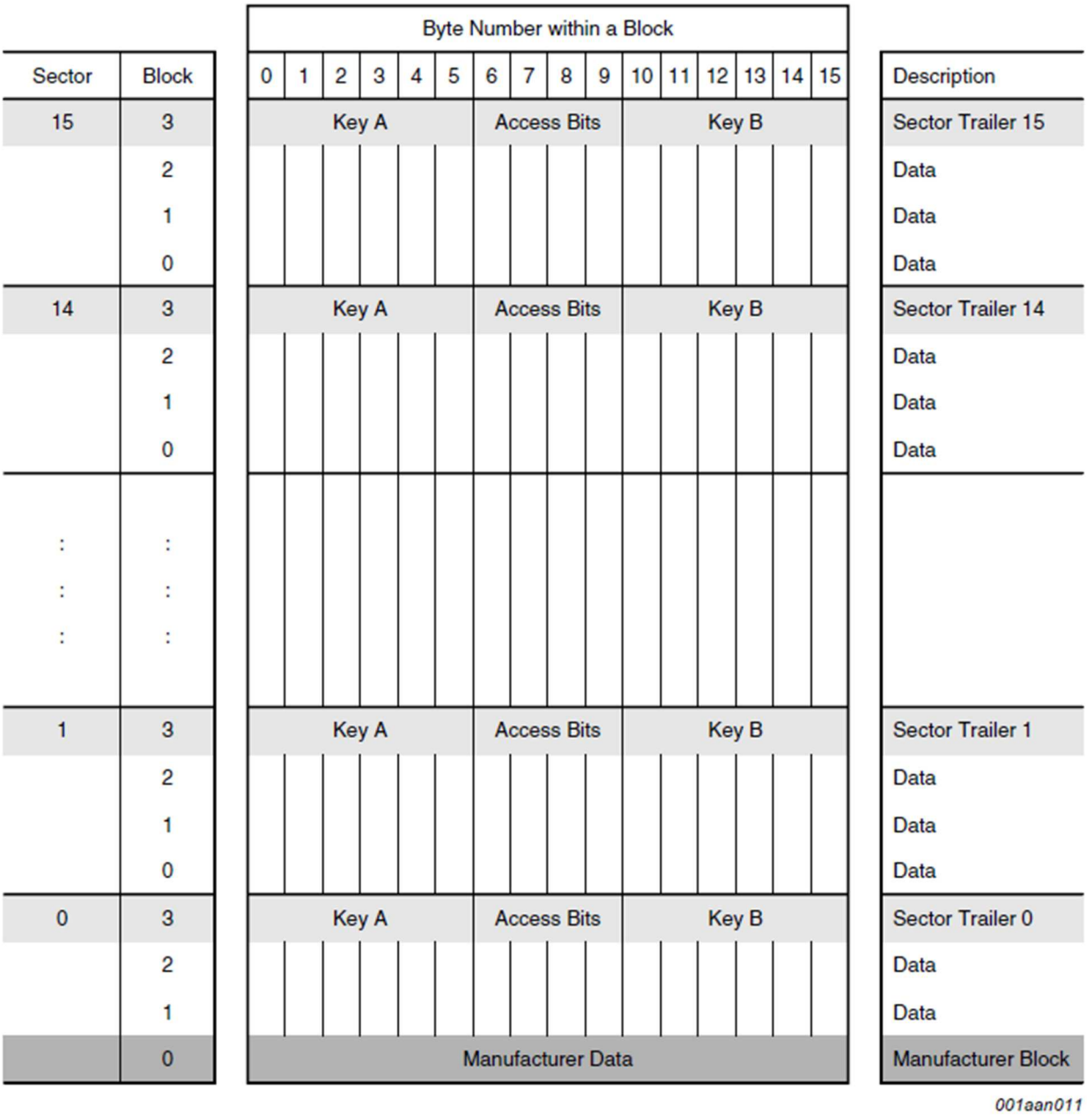
Dal punto di vista della sicurezza *MF1S503x* è dotato di un *NUID (Non-Unique IDentifier)* di 4-byte e di due *KEY* utilizzate per l'autenticazione ad ogni settore in memoria per l'abilitazione alla *scrittura/lettura* in essi.

2.5.1 Funzionalità e descrizione dei componenti

Il chip *MF1S503X* è composto da una memoria *EEPROM* di 1 KB, interfaccia *RF (Radio Frequency)* e da un'unità di controllo digitale. L'energia e dati sono trasmessi attraverso un'antenna.

- RF Interface:
  - Modulatore/demodulatore
  - Raddrizzatore di corrente
  - Rigeneratore Clock
  - POR (Power-On Reset)
  - Regolatore tensione

- Anticollisione: Più card possono essere gestite contemporaneamente (selezionate) in sequenza.
- Autenticazione: Qualsiasi operazione in memoria, deve essere autorizzata utilizzando una delle due chiavi presenti nel sector trailer.
- Controllo valori: I dati sono salvati in uno speciale formato ridondante. E' possibile incrementare o decrementare questi valori.
- Interfaccia *EEPROM*
- Unità criptazione: Lo scambio di dati e l'autenticazione durante la comunicazione utilizza l'algoritmo di criptazione *CRYPTO1*.
- *EEPROM*: Memoria da 1 KB (esistono anche altri formati) organizzata in 16 settori da 4 blocchi. Ogni blocco contiene 16 bytes. L'ultimo blocco di ogni settore è chiamato *trailer*, contiene due chiavi segrete e le condizioni (programmabili) di accesso per ogni blocco in quel settore.

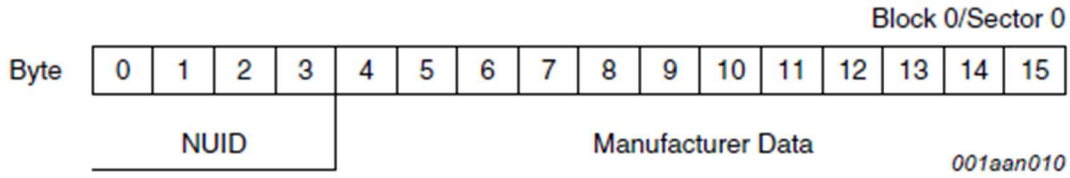


Sector	Block	Byte Number within a Block 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Description
15	3	Key A						Access Bits				Key B						Sector Trailer 15
	2																	Data
	1																	Data
	0																	Data
14	3	Key A						Access Bits				Key B						Sector Trailer 14
	2																	Data
	1																	Data
	0																	Data
:	:																	
:	:																	
:	:																	
1	3	Key A						Access Bits				Key B						Sector Trailer 1
	2																	Data
	1																	Data
	0																	Data
0	3	Key A						Access Bits				Key B						Sector Trailer 0
	2																	Data
	1																	Data
	0	Manufacturer Data																Manufacturer Block

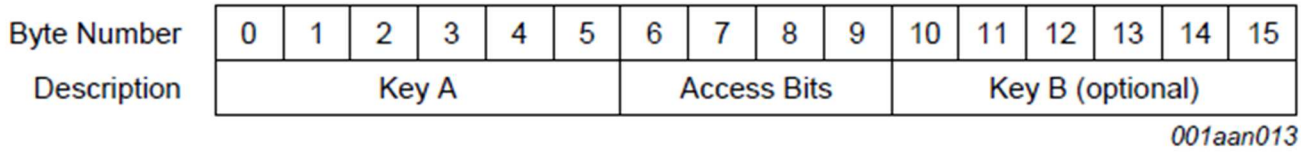
001aan011

Struttura della memoria 1024 x 8 bit EEPROM tag/card utilizzata nel progetto.

Nel primo blocco di dati del settore 0, sono contenute alcune informazioni che riguardano il produttore e il *NUID (Non-Unique IDentifier)*. Questo blocco è programmato nel momento della creazione del *tag* e protetto da scrittura.



Manufacturer block.



Sector trailer.

2.5.2 Principio di comunicazione

I comandi sono inizializzati dal lettore e controllati dall'unità di controllo digitale. La risposta ad un comando dipende dallo stato dell'identificazione e per le operazioni in memoria, dalla condizione di accesso per il corrispondente settore.

2.5.3 Richiesta Standard ISO/IEC 14443

Dopo il *POR (Power-on Reset)* la card risponde ad una richiesta *REQA (Request Type-A)* oppure ad un comando wakeup *WUPA (Wake-Up Type-A)* con un codice, in accordo allo standard *ISO/IEC 14443.*

2.5.4 Loop Anticollisione

Nel loop anticollisione l'identificatore della card è nello stato di pronto. Se è presente più di una card nel campo d'azione del lettore, solo una è selezionata in base all' identificatore per le future operazioni. Le altre card passano nello stato di *Idle (Attesa)* in attesa di un nuovo comando. Se l' *UID (Unique Identifier)* di una card è usato per anticollisione e selezione, sono richiesti due livelli in cascata per il processo, come definito dallo standard *ISO/IEC 14443-3*.

2.5.5 Selezione Card/Tag

Con il comando di selezione della card, il lettore, seleziona un'unica card per l'autenticazione e le operazioni in memoria ad essa associate. La card ritorna un codice di *Select AcKnowledge (SAK)* che determina il tipo di card utilizzata.

2.5.6 Autenticazione in tre passi (Three pass authentication)

L'accesso alla memoria del *tag* non è semplice: bisogna rispettare una sequenza ed effettuare delle autenticazioni con la chiave presente nel *sector trailer*.

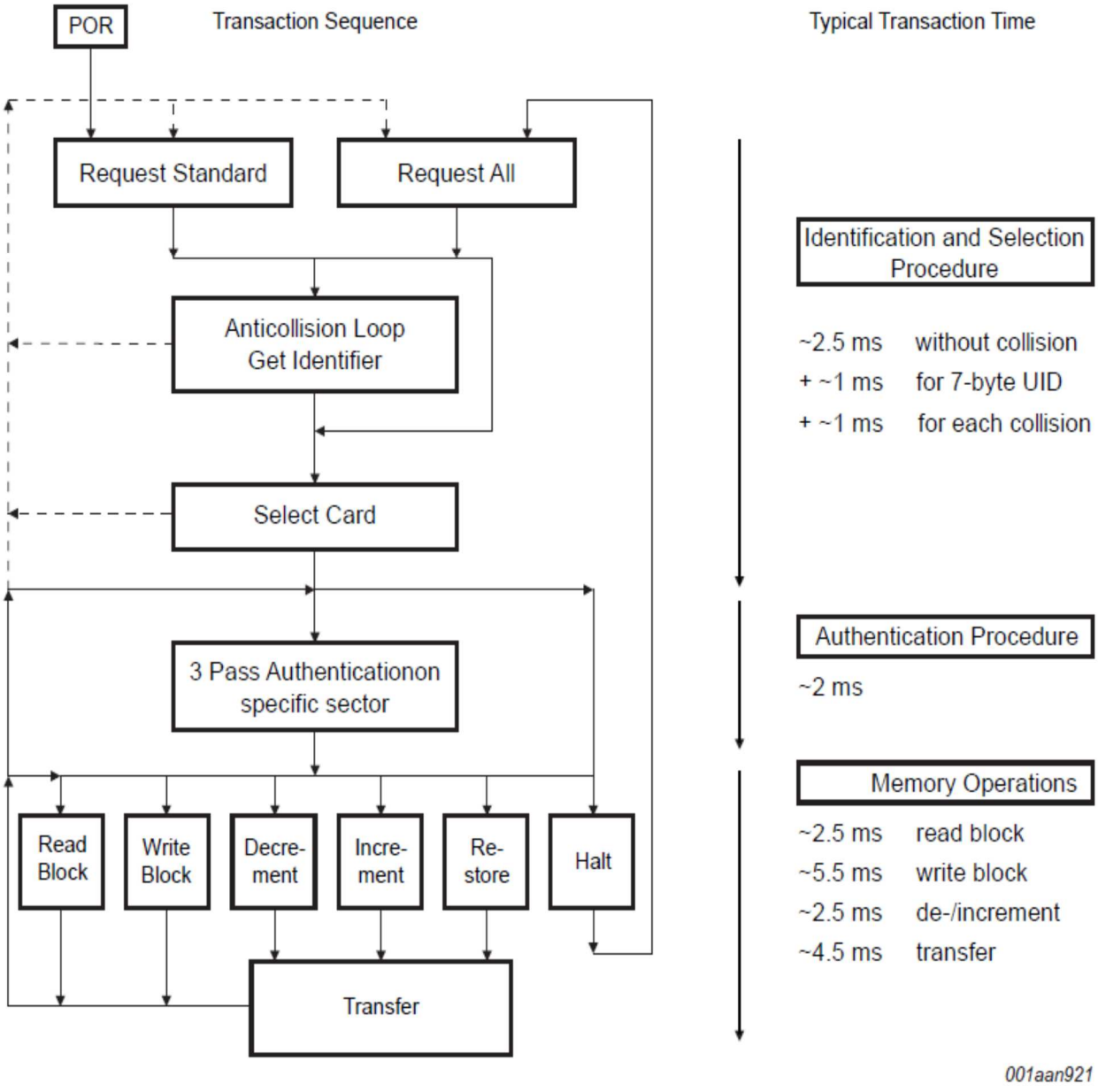
La sequenza di accesso alla memoria "*Three pass authentication*" è la seguente*:*

1. Il lettore specifica il settore dove deve accedere e sceglie la *chiave A* o *B*;
2. La carta legge la chiave segreta e le condizioni di accesso dal *sector trailer*. Successivamente, la carta invia un numero casuale al lettore come *challenge* (*passo uno*).
3. Il lettore calcola il responso usando la chiave segreta e input aggiuntivi. Il responso è trasmesso alla carta, insieme a una *challenge* casuale da parte del lettore (*passo due*).

4. La carta verifica il responso del lettore comparandolo con la propria *challenge* e poi calcola il responso della *challenge* e lo trasmette (*passo tre*).

5. Il lettore verifica il responso della carta comparandolo alla propria *challenge*.

Dopo la trasmissione della prima *challenge* casuale, la comunicazione tra la carta e il lettore viene cifrata.



La figura mostra i passi ed i tempi tipici di una transazione completa.

2.5.7 Operazioni in Memoria

Le possibili operazioni in memoria, disponibili dopo l'autenticazione sono:

- Lettura blocco
- Scrittura blocco
- Decremento: Decrementa il contenuto di un blocco e salva il risultato in un *transfer buff* interno
- Incremento: Incrementa il contenuto di un blocco e salva il risultato in un *transfer buff* interno.
- Spostamento: Sposta il contenuto di un blocco nel *transfer buff* interno.
- Trasferimento: Scrive il contenuto del *transfer buff* interno in un blocco di tipo valore.

2.5.8 Blocchi dati

Ogni settore contiene solo 3 blocchi da 16 *bytes* per lo storage dei dati (il settore 0 contiene solo 2 blocchi per i dati).

I blocchi dati possono essere configurati attraverso l'*access bits* del *sector trailer*, in uno dei seguenti modi:

- Blocco lettura/scrittura
- Blocco valore (Value Block)

I *value block* sono utilizzati da applicazioni in ambito elettronico, dove solo le operazioni di incremento e decremento sono effettuate.

Byte Number	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Description	**value**				**value**				**value**				**adr**	**adr**	**adr**	**adr**
Values [hex]	87	D6	12	00	78	29	ED	FF	87	D6	12	00	11	EE	11	EE

La foto mostra il contenuto di value bock.

2.5.9 Sector Trailer

Il *sector trailer* è l'ultimo blocco di ogni settore, per ogni *sector trailer* sono presenti:

- Una chiave segreta A (obbligatoria) ed una chiave segreta B (opzionale)
- La condizione di accesso (*Access Bits*) ai settori di quel blocco è memorizzata dal byte 6 al byte 9. I bit di accesso specificano anche il tipo di blocchi di quel settore (dati o value block).

Se la chiave B non è richiesta è possibile utilizzare ulteriori 6 bytes come data bytes.

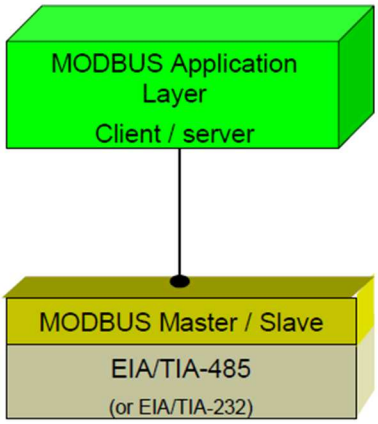
Tutte le chiavi sono settate al valore *FFFF FFFF FFFFh* all'uscita dalla fabbrica e i bytes dell'*access bits* sono settati al valore *FF0780h.*

In conclusione è necessario autenticarsi leggendo il valore della chiave presente nel *sector trailer* prima di effettuare una qualsiasi operazione in memoria (lettura o scrittura).

2.6 Protocollo di comunicazione Modbus

Il protocollo in questione viene utilizzato principalmente nei sistemi di automazione, definisce il formato e le modalità di comunicazione tra un "*master*" che gestisce il sistema e uno o più "*slave*" che rispondono alle interrogazioni del master. Il protocollo definisce inoltre come il *master* e gli *slave* stabiliscono e interrompono la comunicazione, come trasmettitore e ricevitore sono identificati, come i messaggi vengono scambiati e come gli errori rilevati.

Layer	ISO/OSI Model	
7	Application	MODBUS Application Protocol
6	Presentation	Empty
5	Session	Empty
4	Transport	Empty
3	Network	Empty
2	Data Link	MODBUS Serial Line Protocol
1	Physical	EIA/TIA-485 (or EIA/TIA-232)



Le applicazioni che utilizzano il protocollo si piazzano al livello 7 della pila *ISO/OSI*.

Nel *modbus* su linea seriale il ruolo di client è ricoperto dal *master* mentre i nodi *slave* occupano il ruolo di server.

È possibile connettere fino a 247 *slave* su una linea comune (questo è solo un limite logico). L'interfaccia fisica può limitare ulteriormente. Se, ad esempio, l'interfaccia standard *RS-485* permette di avere solo 31 *slave* connessi alla linea, è possibile superare questo limite sostituendo l'ultimo elemento con un "*bridge* o *ripetitore*" per poter connettere altri 31 *slave*.

Le fasi importanti della comunicazione su *MODBUS* sono:

- Solo il *master* può iniziare una transazione.
- Una transazione può avere il formato "*domanda/risposta*" verso un singolo *slave* oppure *broadcast* in cui il messaggio viene inviato a tutti i dispositivi sulla linea che non danno risposta.
- Alcune caratteristiche sono definite da: Standard di interfaccia, parità, numero di bit ed il formato *RTU (Remote Terminal Unit)* binario.
- Indirizzo dello *slave* a cui inviare il messaggio.
- Formato dei messaggi, così composto: indirizzo del dispositivo con cui il *master* ha stabilito la transazione, il codice della funzione che deve essere o è stata eseguita, i dati che devono essere scambiati, il controllo dell'errore secondo l'algoritmo *CRC16* (se viene rilevato un errore, il messaggio viene scartato).

Funzioni più utilizzate per la comunicazione:

Funzione	Codice	Descrizione
01	Read Coil Status	Legge il valore dei discrete output di uno slave.
02	Read Input Status	Legge il valore dei discrete input di uno slave.
03	Read Holding Register	Legge il valore binario degli holding register di uno slave.
04	Read Input Register	Legge il valore binario degli input register di uno slave.
05	Force Single Coil(Writer)	Forza un coil al valore ON od OFF.
…	….	…
08	Diagnostics	Esegue le funzionalità di diagnostica.

Nel progetto è stata utilizzata la funzione 03 (*Read Holding Register*) per inviare i seguenti dati al *master*:

- Tipo di Board (lettore *RFID (Radio Frequency IDentification)*, *board* temperatura, *board* tapparelle, ecc).
- Presenza in stanza.
- Numero di stanza.
- Numero di chiavi memorizzate in *EEPROM*.
- Numero di accessi che ha effettuato una singola chiave.

Esistono altre varianti del *modbus*, in particolare quella utilizzata è *RS-485 over Serial line* (su linea seriale).

2.6.1 MODBUS RS-485 over Serial Line con USB TO TTL/RS-485 Interface (per PC-MASTER)

1.2 Specifiche Modbus

La tabella qui sotto descrive le specifiche dell'interfaccia Modbus presente:

Specifiche Modbus	Descrizione	Commenti
Protocollo	Modbus RTU	E' supportata solo modalità "Slave"
Connettore	Terminale a vite	
Connessione Modbus	RS485 - 2 wire	
Indirizzo slave	1-247	Al primo avvio va settato mediante display, altrimenti tramite messaggio Modbus [a]
Terminazione di linea	Assente sull'apparato	Se necessaria procedere come descritto in 2.1
Velocità di trasmissione supportate	1200, 2400,4800,9600,19200,38400 Kb/s	Settare tramite display o messaggio Modbus [a]
Start bit	1	
Data bit	8	
Stop bit	1 o 2	Settare tramite display o messaggio Modbus [a]
Parità	Nessuna,Pari o Dispari	Settare tramite display o messaggio Modbus.v [a]

I registri hanno dimensione 16 *bit*, se il contenuto del registro è *0x7FFF*, il contenuto non è disponibile.

I registri di tipo *R/W* sono disponibili in lettura tramite i function code *0x03*, *0x04*, in scrittura mediante i function code *0x06*, *0x10*.

I registri di tipo *R* sono disponibili in sola lettura mediante i function code *0x03* e *0x04*.

I dati sono tutti di tipo *UNSIGNED*, a meno dei registri dedicati alla temperatura o altri valori, i cui dati sono di tipo *SIGNED*.

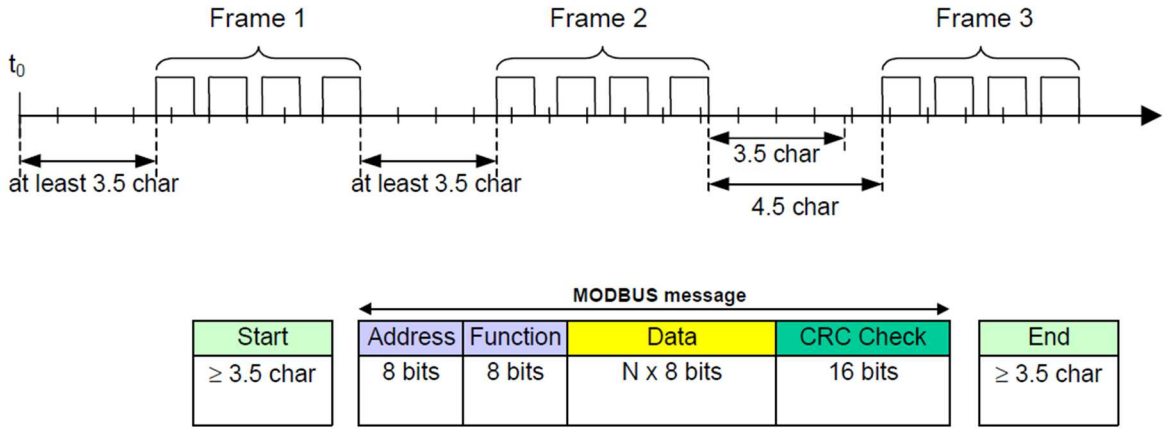
RS485 utilizzato su *Arduino* prevede l'invio di dati seriali su lunga distanza (fino a 1KM) con un rate di 20Mbps, e può essere usato anche in ambienti dove la presenza di rumore è costantemente elevata, ad esempio negli ambienti industriali automatizzati. In una rete *RS485* solo un singolo *slave/device* è nello stato di invio tutti gli altri rimangono in quello di ricezione, su *Arduino* questo viene effettuato tramite i due pin *RE* (*receive enable*) e *DE* (*data enable*) che sono dedicati al controllo, abilitazione alla ricezione dei dati e l'invio di essi sulla rete.

USB to RS485 si occupa invece della conversione, ed è stato usato per la simulazione del *MASTER* su *PC*, in modo da poter leggere i dati inviati dallo slave, dopo aver simulato una richiesta, tramite un software per *PC* (*ModbusMAT*).

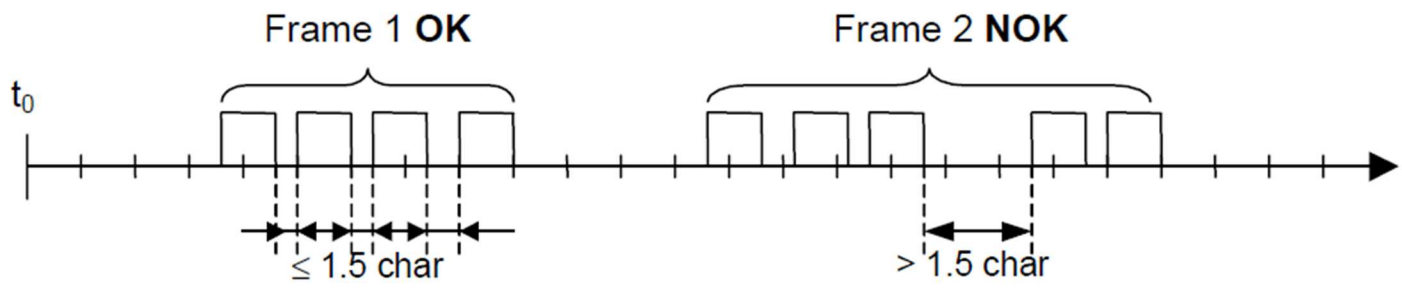
2.6.2 Tempi di trasmissione messaggi su MODBUS

I messaggi trasmessi su linea seriale tramite il protocollo *MODBUS*, sono inseriti in un frame di cui si conosce il punto di inizio e fine. Questo permette ai dispositivi che ricevono i messaggi di sapere se un messaggio è stato inviato completamente oppure no. I messaggi parziali infatti, vengono scartati, ritornando un codice di errore.

Nella modalità *RTU* (utilizzata nel progetto), i frame che contengono dati, sono separati da un piccolo intervallo, pari alla trasmissione di 3.5 caratteri, questo tempo è denominato *t3.5*.



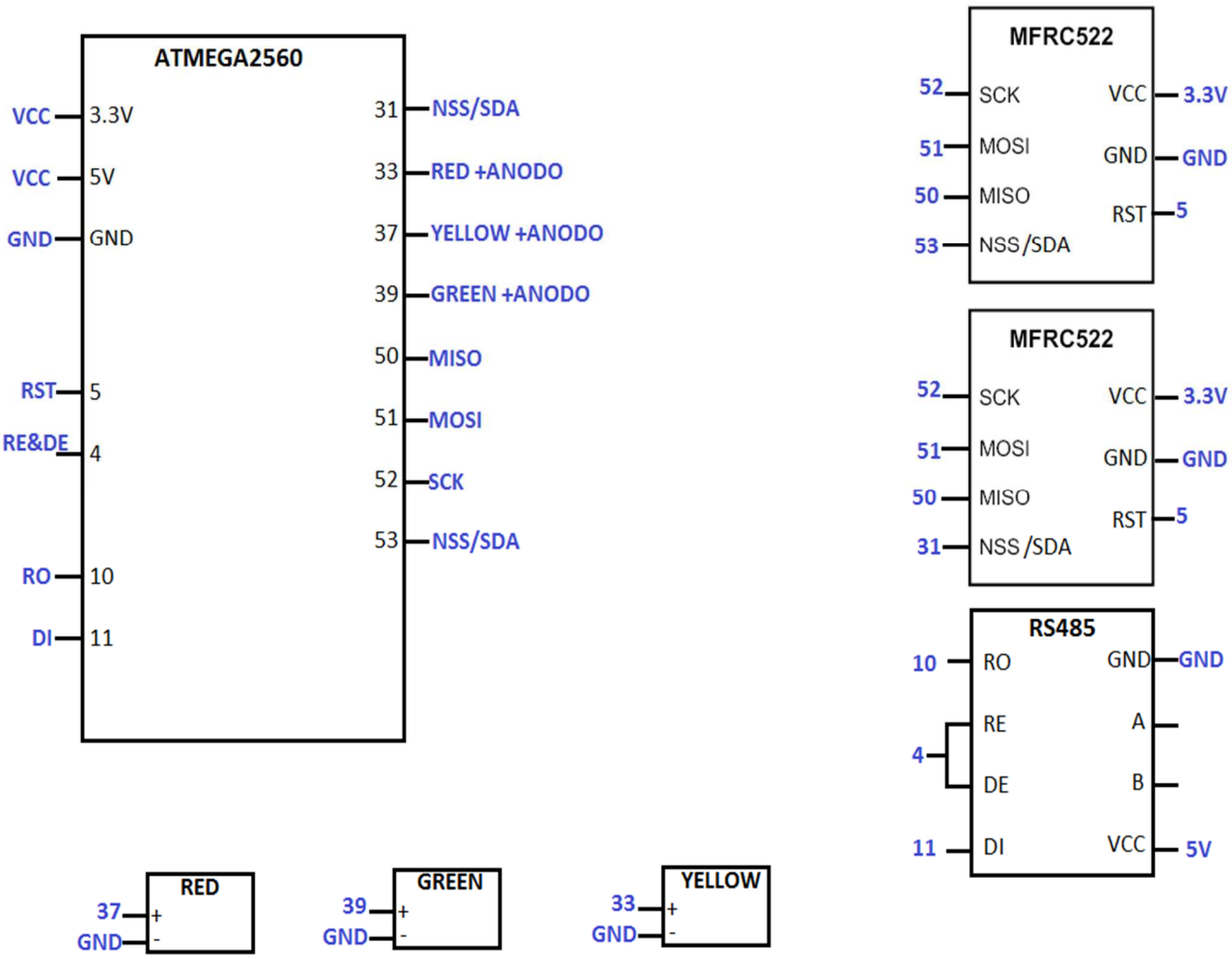
Come è possibile vedere dalla figura il messaggio iniziale deve essere trasmesso come un flusso continuo di caratteri. Se tra un messaggio e l'altro l'intervallo di tempo è maggiore di 1.5 caratteri, il frame è dichiarato incompleto e scartato dal ricevitore.



L'implementazione del protocollo *RTU* implica una notevole gestione di interruzioni da parte del ricevitore, causata dai tempi t1.5 e t3.5. Usando un alto livello di *baud rate* si genera un enorme carico per la *CPU*. Di conseguenza questi due tempi devono essere ristretti se il *baud rate* è uguale o minore a 19200 Bps. Per *baud rate* superiori a 19200 Bps è consigliato utilizzare tempi dell'ordine di 750µs per il time-out dei caratteri interni (t1.5) e 1.750ms per il delay dell'intero frame (t3.5).

2.6.3 Collegamento dei componenti alla board

Dalla figura è possibile vedere il collegamento, dei singoli componenti alla board principale ATMEGA2560.



I due lettori *RFID (Radio Frequency Identificator)* sono alimentati da una tensione di *3.3 Volts* e collegati in parallelo in modo da poterli utilizzare indipendentemente e contemporaneamente.

Nei pin *53* e *31* è collegato l'*SDA (Serial data address),* linea dati dei rispettivi lettori, dato che non è possibile averlo in comune come per gli altri pin.

Passando al *MODBUS RS485* i pin *DE* (*data enable*) e *RE* (*receive enable*) sono collegati insieme nel pin 4.

Mentre *A* e *B* sono collegati rispettivamente ad un convertitore *RS485* to *USB*, collegato a sua volta ad un *PC*, utilizzato per la simulazione del *master*.

CAPITOLO 3

Software Sviluppato

3.1 Introduzione

Risulta essere estremamente non corretto giudicare la programmazione sistemi embedded come la programmazione di piccoli computer, in quanto l'obiettivo principale di questi sistemi è l'interazione con il mondo fisico. I sistemi embedded richiedono che alcune astrazioni di programmazione vengano integrate e assimilate dal software che si intende sviluppare per rispondere al meglio alle possibili situazioni nelle quali il sistema è calato. Proprietà di un sistema embedded:

· Tempestività

· Concorrenza

· Vivacità

· Interfacce

· Eterogeneità

· Reattività

La programmazione di un sistema embedded richiede, il più delle volte, una stretta interazione tra hardware e software. Avendo a disposizione risorse estremamente limitate (per esempio, un processore la cui potenza computazionale sia solo sufficiente alle esigenze e non sovradimensionato permette un risparmio consistente in termini di potenza elettrica) lo sviluppatore è spinto a conoscere intimamente la piattaforma hardware utilizzata al fine di sfruttarne tutte le capacità e in alcuni casi, lo sviluppatore si troverà persino a modificare le caratteristiche tecniche del sistema per migliorare le proprietà sopra citate relative all'ambiente in cui il suo sistema dovrà operare.

Questo non accade nella totalità dei casi perché esistono piattaforme per sviluppare sistemi embedded con una elevata ed efficace astrazione dell'hardware.

Questo permette al programmatore di concentrarsi solo sul livello applicativo confidando che gli strumenti di sviluppo ottimizzino il software al meglio per il sistema embedded finale.

Proprietà del software embedded

· Efficienza del codice:

La capacità di rendere ogni singola riga di codice il più efficace in termini di prestazioni fa la differenza nel momento in cui si hanno a disposizione limitate risorse.

· Interfacce delle periferiche:

Le periferiche rappresentano in un sistema embedded le estensioni grazie alle quali è possibile interagire con l'ambiente circostante: percependolo (sensori) e modificandolo (attuatori).

· Robustezza del codice:

Una delle differenze più entusiasmanti dal punto di vista della programmazione è la necessità che il sistema software sul dispositivo non debba ricorrere ad un riavvio (più o meno forzato che sia) ma che lavori in un periodo virtualmente infinito senza la necessità dell'intervento dell'uomo.

· Riusabilità del codice:

Sebbene sistemi embedded di produttori diversi abbiano peculiarità a volte molto differenti gli uni dagli altri, la possibilità di riutilizzare il codice per progetti diversi e magari su sistemi hardware diversi rimane uno strumento molto potente in mano allo sviluppatore, che in questo modo potrà, con pochi accorgimenti, copiare funzionalità anche complesse da un sistema al altro in poco tempo.

· Strumenti di sviluppo:

Gli strumenti per il *debugging* sono generalmente abbastanza limitati in quanto risulta piuttosto difficile, nonostante il livello di astrazione fornito, riuscire a catturare le peculiarità di una piattaforma di cui non si possono conoscere le finalità. Generalmente il produttore di piattaforme embedded correda di strumenti di emulazione software che fornisce al programmatore un buon inizio per la fase di *debugging*. Avendo a che fare con ambienti le cui variabili non possono essere previste in maniera esaustiva all'interno dei modelli utilizzati per l'astrazione del mondo circostante ai fini del proprio progetto, la fase, per ora, di *debugging* prevede inevitabilmente prove sul "campo".

3.2 Strumenti e tecnologie utilizzati

Quando si avvia un progetto, dopo aver definito le funzionalità e i tratti essenziali, arriva la necessità di scegliere il linguaggio di programmazione.
La scelta del linguaggio di programmazione dipende essenzialmente dalle funzionalità del sistema da implementare, dalla piattaforma e, soprattutto, dal contesto. Altro fattore da non sottovalutare è la presenza di librerie utili per il progetto in questione.

A seguito di quanto detto, dopo una accurata analisi e studio delle principali tecnologie presenti nel mondo delle board *Arduino*, si è optato per un ambiente di sviluppo *C-like*, quale *ATOM* con package *platformIO*.

Il compilatore usato da Arduino è *AVR-GCC* dove *GCC* sta per *GNU Compiler Collection*, ed è una collezione di compilatori (*per C, C++, JAVA..*) completamente gratuiti. In questo modo è possibile compilare il codice *C/C++*, generando codice in linguaggio macchina adatto per essere caricato sui microcontrollori *Atmel* della famiglia *AVR*.

Inoltre è stato possibile sfruttare a pieno le potenzialità fornite da *ATOM* e *platformIO* che prevedono un aggiornamento automatico delle librerie e *packages* presenti su *GitHub* o in *Arduino IDE*.

3.3 Sketch Init: Fase Inizializzazione board

La fase prevede:

1. Salvataggio in *EEPROM* dell'*ID* del *MASTER* e numero stanza, agli indirizzi che si trovano nella parte finale della memoria (4095):

```
Memory content from EEPROM.length()-1
Struct saved:
Number: 500
ID CARD:
In hex:  93 FF 3E 02 (CARD)
In dec:  147 255 62 02

4095    1
4094    244
4093    2
4092    62
4091    255
4090    147
4089    0
```

   Questi dati (*ID Master* e *numero stanza*) sono caricati a ogni *RESET* della board in una struttura dati alla posizione 0.

2. Ogni locazione ad uno specifico indirizzo in memoria è grande 1 byte (8 bit). La lettura (per il salvataggio non accade) di un valore è fatta indirizzo per indirizzo, ed è possibile memorizzare solo numeri da 0 a 255, per il numero di stanza che è *unsigned int* (2 byte). Questo costituisce un problema (utilizza due locazioni in memoria). Per superare questo ostacolo ed evitare la forzatura di inserire solo stanze con numeri da 0 a 255, si effettua una pseudo-conversione *Byte* to *Int*, dove, dopo aver letto il byte alla locazione in cui è memorizzato (essendo 2 byte ci sarà un *byte high* e *byte low*), il *byte high* viene shiftato di 8 posizioni verso sinistra e messo in *OR (logica)* con il *byte low* per ottenere il valore finale. L'*ID* del *tag* è su 4 byte.

3. Programmazione della card master, con salvataggio degli *ID* dei *tags* e numero di stanza che abilitano. Ci sarà uno sketch caricato sulla board, per il salvataggio nel master dei vari *ID*, lo sketch è molto semplice, dopo aver riconosciuto l'*ID* del *tag*, e aver digitato il numero di stanza corrispondente, chiede all'utente di appoggiare la card *MASTER* per il salvataggio dei dati appena letti. Lo sketch può terminare in qualsiasi momento a meno che la memoria della card non sia esaurita. Ecco un esempio della card master prima e dopo il salvataggio dell'*ID* di un *tag* e del *numero stanza* ad essa associato (la card è di 1KB quindi ci saranno 16 settori dove ogni settore prevede 4 blocchi per un totale di 64 blocchi da 16 byte):

PRIMA

```
Card UID: 93 FF 3E 02
Card SAK: 08
PICC type: MIFARE 1KB
Sector Block   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15  AccessBits
  15     63   00 00 00 00  00 00 FF 07  80 69 FF FF  FF FF FF FF  [ 0 0 1 ]
         62   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
         61   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
         60   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
```

DOPO

Tag ID (in HEX): 64 9C 20 DC

Numero Stanza = 500 unsigned int (in HEX)

```
Card UID: 93 FF 3E 02
Card SAK: 08
PICC type: MIFARE 1KB
Sector Block   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15  AccessBits
  15     63   00 00 00 00  00 00 FF 07  80 69 FF FF  FF FF FF FF  [ 0 0 1 ]
         62   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
         61   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
         60   64 9C 20 DC  01 F4 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
```

Al termine della fase di inizializzazione, si avvia la fase di implementazione dello sketch principale. Tale sketch prevede due funzioni *void setup()* e *void loop()*. La *void setup()* è eseguita singolarmente nella fase di *Accensione/Reset* della scheda, mentre *void loop()* è eseguita ciclicamente all'infinito (è come il corpo di un *while(true)*). Nella *void setup()* è presente tutta una serie di operazioni, quali: inizializzazione dei vari pin, caricamento struttura dati dalla *EEPROM*, istanza degli oggetti, *init* degli *RFID* e porte seriali, setup del *modbus*.

3.3.1 Sketch principale: lettura/gestione TAG RFID

Lo sketch principale si occupa della lettura, gestione e salvataggio in memoria EEPROM dei TAG adoperati per l'apertura della stanza dove la board è installata.

Come anticipato, si suppone che, sulla board siano già stati avviati e configurati i due sketch secondari, prima di quest'ultimo.

Immaginiamo il codice di implementazione dello sketch, suddiviso in tre parti principali, in cui ogni singola parte svolge una determinata funzionalità necessaria allo sviluppo dell'intero progetto e all'inserimento successivamente in un contesto molto più ambio e generalizzato.

3.3.2 Sketch principale: Prima parte

In questa prima parte sono definite tutte le variabili utilizzate all'interno del codice, compresa la definizione della struttura dati, e l'inclusione delle librerie. I tipi di variabile utilizzati sono:

Il tipo *BYTE* viene rappresentato in memoria con 8 *bit* ed è possibile memorizzare numeri con un *range* da 0 a 255.

Il tipo *unsigned int* è rappresentato in memoria con 2 *byte* ed è possibile memorizzare numeri da 0 a 65535 ($2^{16} - 1$).

Il tipo *bool* (come il *byte*) occupa solo 1 *byte* in memoria ed è utilizzato per memorizzare due valori *true* o *false* (0 o 1).

Il tipo *MFRC522,* classe definita nella libreria *MFRC522.h,* specifica alcuni comandi e funzioni utilizzate per la gestione dei *TAG* e lettori.

Tra questi possiamo identificarne alcuni.

Comandi e status per la gestione dei registri MFRC522*:*

Comandi esecuzione.
Comandi start e stop.
Bit di errore del comando.
Bit stato di comunicazione.
Numero di byte nel FIFO buffer (First in First Out).

Posizione del primo bit-collisione trovato sull'interfaccia RF (Radio Frequency).

Ecc...

Comandi generici:

Cancellazione del comando in esecuzione.

Memorizzazione di 25 byte nel buffer interno.

Generazione di un numero random da 10 byte.

Attivazione auto-test.

Trasmissione dati dal FIFO buffer.

Attivazione del circuito ricevitore.

Trasmissione dei dati dal FIFO buffer all'antenna e attivazione automatica del ricevitore in seguito alla trasmissione.

Autenticazione standard in modalità lettura.

Reset lettore.

Definizione del guadagno espresso in db con un minimo di 18 db ad un massimo di 48 db.

Comandi inviati alla carta di prossimità con circuito integrato **PICC (Proximity Integrated Circuit Cards):**

Comandi usati per la gestione della comunicazione con molteplici PICC (ISO 14443-3, Type A, section 6.4).

Comandi per l'autenticazione con chiave A o B alla scrittura in memoria.

Comandi di lettura/scrittura blocco (16 byte).

Lettura del registro interno.

Scrittura del registro interno in un blocco.

Incremento/decremento di un valore nel blocco (in caso sia settato come value block).

Tipo di card riconosciute:

```
PICC_TYPE_UNKNOWN
PICC_TYPE_ISO_14443_4
PICC_TYPE_ISO_18092
PICC_TYPE_MIFARE_MINI
PICC_TYPE_MIFARE_1K
PICC_TYPE_MIFARE_4K
PICC_TYPE_MIFARE_UL
PICC_TYPE_MIFARE_PLUS
PICC_TYPE_TNP3XXX
```

Codici di status:

```
STATUS_OK
STATUS_ERROR
STATUS_COLLISION
STATUS_TIMEOUT
STATUS_NO_ROOM
STATUS_INTERNAL_ERROR
STATUS_INVALID
STATUS_CRC_WRONG
STATUS_MIFARE_NACK
```

Struttura dati per contenere UID (Unique Identifier):

Un campo per la grandezza, espressa in byte dell'UID (4,7 o 10).

Un campo per il byte SACK (Select Acknowledge) inviato dal PICC dopo la lettura e selezione corretta del tag.

Un array di byte che contiene i valori dell'UID.

Struttura dati usata per la chiave CRYPTO1:

Un array di byte, settato di default a FFFFFFFFFFFFh, come da documentazione.

Funzioni per interfacciamento e comunicazione con il lettore.

Funzioni per la manipolazione del lettore:

Accensione/Spegnimento antenna.

Reset.

Settaggio e lettura del guadagno dell'antenna.

Test funzionamento antenna.

Inizializzazione lettore (attivazione Power Down Pin).

Funzioni per la comunicazione tra card/tag e lettore MFRC522:

WakeUP card.

Selezione card.

Lettura/scrittura memoria card.

Funzione PCD˙StopCrypto1() (stop del traffico criptato), utilizzata dopo le operazione effettuate in seguito ad una autenticazione in memoria. In caso non venga utilizzata nessuna nuova comunicazione può iniziare.

Funzioni di supporto (debugging):

Funzioni debugging (scrittura su linea seriale del dump della memoria di una card).

Funzioni dettagliate di debugging (compresi raw bytes).

Funzioni di debugging in base al tipo di card (MIFARE Classic, MIFARE Ultralight).

Funzioni Avanzate:

Settaggio dell'Access Bits.

Settaggio dell'UID (cambia UID di una card), da utilizzare con cautela.

Unbrick del settore 0 contenente l'UID, così la card può essere letta.

Ritornando allo sketch principale, un array di due elementi è dichiarato come oggetto di tipo *MFRC522*. Questo per tenere separati gli *UID* letti da entrambi i lettori, uno utilizzato per l'*apertura porta*, l'altro per la *presenza in stanza*.

È definita anche una *key* di tipo *MIFARE_KEY* della classe *MFRC522*, utilizzata come array di byte per l'autenticazione alla *lettura/scrittura* dei settori presenti in memoria card/tag, l'array contiene la default key *FFFF FFFF FFFF*.

```
MFRC522 rfid[2];
MFRC522::MIFARE_Key key;
```

Vengono definiti in questa prima parte anche i *default PIN* visti nel capitolo precedente ed utilizzati per il settaggio e la programmazione della board.

```
#define SS0_PIN 53
#define SS1_PIN 31
#define RST_PIN 9
#define RED_LED 37
#define YELLOW_LED 33
#define GREEN_LED 39

byte ssPins[] = { SS0_PIN, SS1_PIN };
```

L'array di byte, contenente i due pin *SDA,* dei rispettivi lettori, non in comune, ed utilizzati nella seconda parte dell'implementazione del codice per istanziare l'array di tipo *MFRC522*.

Come ultima, ma non meno importante, viene definita una struttura dati di tipo *ROOM*

```
struct Room
{
    byte id[4];
    unsigned int number;
};
```

La struttura, come illustrato in figura, è molto semplice, composta da due soli campi. Il primo campo è predisposto a contenere l'*UID,* il secondo il *numero di stanza* definito come *unsigned int* per rendere semplice la gestione delle stanze evitando l'inserimento di un eventuale numero negativo.

Questa struttura è utilizzata per dichiarare un array di tipo *ROOM di 10 elementi,* dove in ogni posizione dell'array è presente la *coppia id/numero stanza* delle varie *card/tag* che ne abilitano l'apertura.

```
Room card[10];
```

La generalizzazione del codice e l'allocazione dinamica della struttura, rappresentano un grosso problema per i *sistemi embedded* che vengono programmati per compiere una determinata funzione. Ad esempio in quasi tutti i *sistemi embedded* molto critici, l'allocazione dinamica è esplicitamente proibita o deprecata, oltre che di notevole complessità. Da un punto di vista tecnico non avrebbe avuto senso definire una struttura dinamica anche perché non si avranno mai stanze con più di dieci chiavi abilitate all'apertura.

Nelle direttive di inclusione è inclusa anche la libreria sviluppata nel corso del progetto denominata *MoMatic.h*, che implementa le principali funzioni di comunicazione su *MODBUS*.

La classe *MoMatic* presente all'interno della libreria, definisce funzioni, effettua l'assegnazione dell'*ID* ad una board fino all'invio e ricezione dati su *MODBUS*.

Vediamo nel dettaglio il concetto alla base di ogni funzione:

MoMatic(); costruttore della classe.

Void momatic_set_id(); assegna un *ID* univoco alla board, differente dalle altre board. In un albergo ci saranno installate *n board* e dovranno avere tutte un *ID* differente tra di loro.

Momatic_get_id(); ritorna l'*ID* corrente della board.

Momatic_setup(*parametro1, parametro2); inizializzazione dei registri utilizzati dal *MODBUS.* Il primo parametro è un puntatore all'array dei registri, il secondo la grandezza.

Update(); utilizzata per il *refresh* dei registri.

I dati inviati al master tramite *MODBUS* sono essenzialmente pochi: presenza in stanza, numero stanza, numero di chiavi abilitate all'accesso, contatore che indica il numero di accessi che ha effettuato una singola chiave. Questo problema è stato risolto dichiarando un array di interi senza segno composto da tanti elementi quante sono il numero di chiavi, tutti inizializzati a zero.

3.3.3 Sketch principale: Seconda parte

Questa seconda parte del codice è affidata all'Inizializzazione della *board* con istanza degli oggetti e chiamate alle principali funzioni di settaggio:

Caricamento dati dalla EEPROM alla struttura dati.

Salvataggio dati dalla struttura alla EEPROM.

Cancellazione memoria EEPROM.

Conversione byte a intero.

Presenza in stanza.

Abilitazione apertura.

Riconoscimento master.

Conteggio elementi della struttura dati.

Caricamento master dalla EEPROM.

Lampeggio LED.

Check presenza card sul primo lettore.

Check presenza card sul secondo lettore.

Lettura memoria master con prelievo delle card memorizzate.

Stampa su linea seriale le card/tag (debugging).

Andiamo a vedere nel dettaglio la logica dietro ogni singola funzione.

3.3.4 Inizializzazione degli oggetti e implementazione funzioni

La funzione che compie il settaggio dei vari *PIN* e inizializza le comunicazioni seriali e gli oggetti, è la *void setup()*.

Nei capitoli precedenti si è discusso di come questa funzione venga avviata ad ogni *RESET* della board, prima di eseguire la funzione principale *void loop()*;

Proprio per questo motivo alcune chiamate di funzioni sono eseguite in quest'ultima, come ad esempio il caricamento della struttura dalla *EEPROM*.

Caricamento dalla EEPROM

L'idea alla base di tutto è che a partire dalla prima posizione in memoria *EEPROM* sia salvato il primo indirizzo utile, successivo all'ultimo byte della struttura e infine la struttura dati.

In questa situazione è possibile distinguere due casi limite:

- Memoria vuota (nessun dato presente al suo interno)
- Memoria parzialmente piena (è presente almeno una *card/tag*)

Il primo caso è facilmente gestibile, infatti basta effettuare un controllo sul valore presente nella prima locazione in memoria, se pari a zero allora non viene caricato nessun dato, altrimenti sono caricati tutti i byte a partire dalla prima locazione utile.

Nel secondo caso, è molto importante che durante il prelievo dei dati in *EEPROM* ci sia un ciclo che tenga conto dell'indirizzo corrente, aggiornando l'indice con una *sizeof (elemento struttura)*, perché i dati all'interno della memoria sono organizzati in maniera sequenziale.

Esempio, un intero senza segno occuperà due locazioni in memoria essendo rappresentato su due byte, e l'*UID* della card quattro, quindi supponendo che la prima card sia salvata a partire dall'indirizzo 1, la seconda si troverà all'indirizzo 7, 4 *byte* per l'*UID* + 2 *byte* per il *numero stanza* = 6 *byte*.

Sotto una figura che mostra il l'*UID del tag* con il *numero di stanza* (500) memorizzati in *EEPROM* con un indice che tiene conto del primo indirizzo utile da cui iniziare a salvare la successiva *card/tag*.

```
Memory content from 0
0       7 -> Indirizzo utile
1       100 -> UID TAG
2       156
3       32
4       220
5       244 -> Numero stanza memorizzato su due byte (500)
6       1
7       0 -> Card successiva se presente o primo indirizzo utile
8       0
9       0
10      0
11      0
12      0
13      0
14      0
```

Stampa su linea seriale le card/tag (debugging)

Con una *Serial.Println()* stampa a video *UID* con *numero stanza* memorizzate in struttura, formattandone il testo.

Ecco un esempio della stampa a video:

```
Struct saved:
Number: 500
ID: 93FF3E2
```

Conteggio elementi della struttura dati

Realizzata con un ciclo *for* che con un semplice confronto tra ogni elemento della struttura con il carattere speciale *NULL*, conta il numero di *card/ tag* presenti.

Se durante il ciclo l'elemento corrente è diverso da *NULL* allora viene conteggiato altrimenti passa al successivo tenendo presente che la struttura ha una grandezza fissa in memoria (10 elementi), per i problemi elencati nei capitoli precedenti.

La funzione ritorna un contatore che indica il numero di elementi presenti.

Salvataggio dati dalla struttura alla EEPROM

Il salvataggio della struttura contenente due soli campi *UID* e numero stanza ha come obiettivo quello di memorizzare con una sola istruzione tutta la struttura all'interno della *EEPROM* come si è visto per il caricamento, i dati sono collocati in maniera ordinata sequenzialmente.

Il *worst case* che può verificarsi riguarda la presenza di dati già in memoria. In quel caso la funzione salverà i dati iniziando dal primo indirizzo vuoto.

Come per il caricamento è previsto un ciclo fino al numero di elementi della struttura, con successivo aggiornamento dell'indirizzo corrente tramite una *sizeof()* che ritorna la grandezza espressa in *byte* dell'elemento della struttura. Infine viene aggiornato l'indirizzo utile con il valore dell'indirizzo corrente che ha concluso la scrittura dell'ultimo elemento, e salvato in memoria alla locazione zero.

Cancellazione memoria EEPROM

Cancella il contenuto della *EEPROM* iniziando dall'indirizzo zero fino all'indirizzo utile, impostando come valore di locazione lo zero, laddove sono presenti byte.

Questa funzione è invocata dopo l'aggiornamento della struttura dati che si verifica nello stato di programmazione della board, quando una card *master* viene appoggiata al lettore per il *download* dei dati dalla sua memoria interna, evitando che una card eliminata dalla memoria del *master* sia ancora presente in memoria *EEPROM* della board; la board effettua questi controlli in modo del tutto autonomo con vari *checks* tra gli *ID* appena letti e quelli presenti in struttura.

Conversione byte a intero

Ogni locazione ad uno specifico indirizzo in memoria è grande 1 *byte* (8 *bit*), la lettura di un valore (per il salvataggio non accade) è fatta indirizzo per indirizzo, ed è possibile memorizzare solo numeri da 0 a 255, per il numero di stanza che è *unsigned int* (2 *byte*). Questo è un problema (utilizza due locazioni in memoria): per superare ciò ed evitare la forzatura a inserire solo stanze con numeri da 0 a 255, si effettua una conversione *Byte to Int*, dove, dopo aver letto il *byte* alla locazione in cui è memorizzato (essendo su 2 *byte* ci sarà un *byte high* e *byte low*), il *byte high* viene shiftato di 8 posizioni verso sinistra e messo in *OR (logica)* con il *byte low* per ottenere il valore finale.

Presenza in stanza

Con un ciclo viene effettuato un *check* tra l'*ID* letto sul secondo lettore e gli elementi della struttura. Se positivo attraverso la funzione predefinita di arduino *digitalWrite(PIN_LED,HIGH)* viene attivato il led verde e ritornato il valore booleano *true*, nel caso in cui l'esito dei *check* sia negativo (nessuna corrispondenza tra l'*ID* letto e memorizzato) la funzione ritorna un valore booleano *false*.

Abilitazione apertura

Come per la funzione descritta precedentemente anche qui dopo aver eseguito un *check* tra l'*ID* appena letto sul primo lettore e quelli memorizzati, ed una corrispondenza viene trovata, oltre ad abilitare il relè per l'apertura della porta dopo aver attivato il *LED* verde, viene aggiornato anche il contatore che tiene conto del numero di accessi di ogni card per la card corrispondente rilevata.

Qui si è supposto che la *card[i-esima + 1]* della struttura corrisponda all' *accessCounter[i-esimo],* nella posizione 0 dell'array *card[n]* è memorizzato il *master* che abilita la board al download dei dati.

La funzione ritorna un valore booleano pari a *true* se le condizioni sono verificate altrimenti ritorna *false*.

Caricamento master dalla EEPROM

Carica nella posizione zero dell'array di tipo *Room* il *master* memorizzato in *EEPROM*, a partire dalla locazione *EEPROM.LENGTH() - 1*, attraverso la funzione *EEPROM.get(INDIRIZZO˙MEMORIA, VARIABILE).*

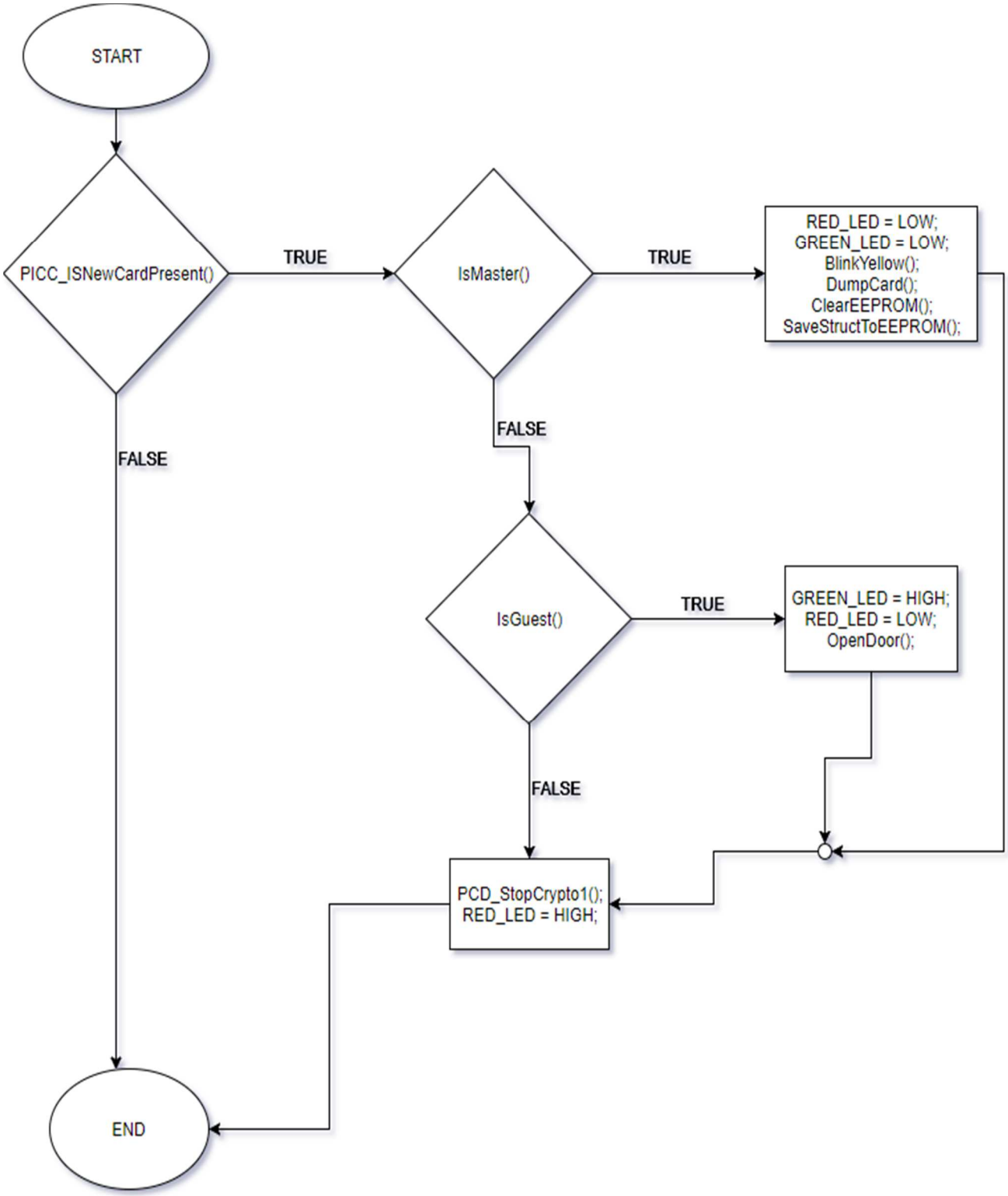
Riconoscimento master

Check tra la card memorizzata alla posizione zero della struttura e l'*ID* appena letto dal primo lettore, se positivo ritorna il valore booleano *true* altrimenti *false*.

Check presenza card sul primo lettore

Questa funzione verrà illustrata attraverso una rappresentazione grafica tramite diagrammi di flusso delle funzionalità che andrà a svolgere.

Sostanzialmente la funzione deve essere in grado di riconoscere la card *MASTER* e distinguere i vari *tag* che abilitano o meno la chiusura della stanza.


START
PICC_ISNewCardPresent()
TRUE
IsMaster()
TRUE
RED_LED = LOW;
GREEN_LED = LOW;
BlinkYellow();
DumpCard();
ClearEEPROM();
SaveStructToEEPROM();
FALSE
FALSE
IsGuest()
TRUE
GREEN_LED = HIGH;
RED_LED = LOW;
OpenDoor();
FALSE
PCD_StopCrypto1();
RED_LED = HIGH;
END

Dal diagramma di flusso si nota da subito la presenza di un blocco condizionale per la presenza della card sul lettore, se la condizione è verificata si passa al secondo blocco condizionale dove con l'aiuto della funzione riconoscimento *MASTER* si attiveranno tutta una serie di funzioni atte a svolgere in ordine:

Prelievo tag dalla memoria del MASTER.

Salvataggio tags nella struttura dati.

Cancellazione EEPROM board.

Salvataggio struttura nella EEPROM.

Queste operazioni prevedono inoltre l'accensione e lo spegnimento dei vari *LED* in modo da distinguere i vari stati della board.

Check presenza card sul secondo lettore

L'idea della funzione è quella di assegnare ad una variabile booleana, il valore *true* fintanto che la card è appoggiata sul lettore.

Durante l'avvicinamento della card sul lettore si effettua un *check* tra l'*ID* della card e quelli presenti in struttura stabilendo se la card è abilitata.

Infine ci sarà un ciclo dove la condizione di uscita è la non presenza della card su di esso, realizzato con l'apposita funzione della libreria *MFRC522 PICC_IsNewCardPresent().*

Nel corpo del ciclo oltre l'assegnazione del valore *true* alla variabile booleana che indica la presenza in stanza, con una *digitalWrite(PIN˙LED,HIGH)*, il led verde viene acceso e permane nello stato di accensione fino alla rimozione della card.

All'uscita della funzione (come descritto nel paragrafo riguardante la libreria *MFRC522*) viene invocata la funzione *PCD_StopCrypto1()* per terminare la comunicazione e quindi fermare il flusso di dati criptato tra il lettore e la *card/tag* così da poter iniziare una nuova comunicazione.

Lettura memoria master con prelievo delle card memorizzate

Da un punto di vista progettuale la funzione deve eseguire un *dump* della memoria del chip contenuto nella card *MASTER*. Come si è visto nei capitoli precedenti ogni *card/tag* è provvisto di una memoria interna che varia a seconda del modello utilizzato (nel progetto la card è di *1024 Kb*).

I *byte* memorizzati in ogni singolo blocco sono 16, per ragioni progettuali e semplicità del codice si è scelto di memorizzare per ogni blocco, la coppia *UID/NUMERO STANZA*.

Come mostrato in figura:

```
Card UID: 93 FF 3E 02
Card SAK: 08
PICC type: MIFARE 1KB
Sector Block   0  1  2  3   4  5  6  7   8  9 10 11  12 13 14 15  AccessBits
  15     63   00 00 00 00  00 00 FF 07  80 69 FF FF  FF FF FF FF  [ 0 0 1 ]
         62   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
         61   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
         60   64 9C 20 DC  01 F4 00 00  00 00 00 00  00 00 00 00  [ 0 0 0 ]
```

Nell'ultimo settore del blocco 60 è memorizzato *UID* (primi 4 byte) e *numero stanza* (i successivi 2 byte).

Con questo metodo ogni card master potrà contenere solo 45 coppie *UID/NUMERO STANZA*.

E' possibile superare questo limite, raddoppiando il numero di stanze memorizzabili, semplicemente utilizzando gli altri 6 *byte* del blocco.

Durante l'implementazione si è tenuto conto di due fattori molto importanti:

- Il primo riguarda il numero di stanza, memorizzato su 2 *byte*, che dovrà essere convertito in intero dopo il prelievo in memoria.
- Il secondo riguarda l'autenticazione e la ricerca di una regola, da effettuare prima dell'accesso ai settori, in modo da leggere i 3 blocchi di dati.
  Ad esempio: dopo la lettura di 3 blocchi viene eseguita una nuova autenticazione.

Nell'implementazione come primo passo vengono dichiarate le variabile temporanee utilizzare nella funzione (che qui indicheremo con un pseudo-codice):

```
CURRENT_BLOCK = 4
TRAILER_BLOCK = 7
BUFFER[18]
BUFFER_SIZE = SIZEOF(BUFFER)
STATUS_CODE
ROOM_NUMBER
TOTAL_BLOCKS = PICC_SIZE / 16
PICC_SIZE = 1024
CARD_INDEX = 1
DEFAULT_KEY = FF FF FF FF FF FF
```

Si è scelto di utilizzare il costrutto di iterazione *WHILE (CONDIZIONE)* per scorrere all'interno dei blocchi di memoria. Questa scelta è stata dettata dall'avere un maggiore controllo sulla condizione da verificare tenendo traccia dell'aggiornamento dell'indice di iterazione. Ciò ha comportato però un'ulteriore istruzione per l'autenticazione al difuori del ciclo, per poter leggere il primo blocco di dati.

Non è stato scelto il *DO_WHILE (CONDIZIONE);* perché c'era la necessita di verificare la condizione prima dell'esecuzione del corpo del ciclo e non alla fine.

Il costrutto *FOR* invece comportava un problema sull'aggiornamento della condizione di iterazione che doveva essere effettuata solo in particolari condizioni e non al termine del ciclo.

Ritornando alla progettazione della funzione usando una pseudo-codifica è possibile distinguere i passi, eseguiti nelle particolari condizioni a seconda dello stato in cui si trova la board, successivo alla lettura di una *card master.*

INIZIO

```
STATUS_CODE=
lettore[1].PCD_Authenticate(KEY_A,TRAILER_BLOCK,DEFAULT_KEY,&letto
re[1].UID);
IF ( STATUS_CODE != STATUS_OK)
     ERRORE
     LAMPEGGIA_TUTTI_LED
     RETURN
ENDIF


WHILE ( CURRENT_BLOCK < TOTAL_BLOCKS – 1)
     IF ( CURRENT_BLOCK == TRAILER_BLOCK )
          TRAILER_BLOCK = TRAILER_BLOCK + 4
          CURRENT_BLOCK++
          STATUS_CODE=
     lettore[1].PCD_Authenticate(KEY_A,TRAILER_BLOCK,DEFAULT_KEY,
          &lettore[1].UID);
          IF ( STATUS != STATUS_OK)
               ERRORE
               LAMPEGGIA_TUTTI_LED
               RETURN
          ENDIF
     ENDIF
```

```
// LETTURA BLOCCO E CARICAMENTO DATI NEL BUFFER
STATUS_CODE = lettore[1].MIFARE_READ ( CURRENT_BLOCK, BUFFER, &BUFFER_SIZE );
IF ( STATUS_CODE != STATUS_OK)
     ERRORE
     LAMPEGGIA_TUTTI_LED
     BREAK;
ENDIF
IF ( BUFFER[0] == 0 AND BUFFER[1] == 0 AND BUFFER[2] == 0 AND BUFFER[3] == 0 AND BUFFER[4] == 0 AND BUFFER[5] == 0 )
     BLOCCO_VUOTO
     BREAK;
ENDIF
ELSE
     ROOM_NUMBER = BYTE_TO_INT (BUFFER[4], BUFFER[5]);
     IF ( ROOM_NUMBER == NUMERO_STANZA_BOARD)
          FOR( i=0 ; i < 4 ; i++)
               CARD[CARD_INDEX].UID[i] = BUFFER[i];
               CARD_INDEX++;
               CARTA_CARICATA
          ENDFOR
     ENDIF
END_ELSE
CURRENT_BLOCK++;
SVUOTA_BUFFER();
END_WHILE
lettore[1].PCD_StopCrypto1();
DUMP_COMPLETATO
FINE
```

Nel *WHILE* la condizione di iterazione sta ad indicare:

Il blocco corrente è un indice che, iniziando dal blocco di partenza 4, incrementerà il suo valore ad ogni iterazione fino al raggiungimento del numero totale dei blocchi meno uno.

Fintanto che il blocco corrente (si è scelto di saltare i primi due blocchi liberi del settore 0 ed iniziare direttamente dal settore 1) che corrisponde a 4 è minore del numero totale dei blocchi meno 1 (escluso il settore 0) allora, *SE* il blocco corrente è un *TRAILER BLOCK* , aggiorna la variabile che tiene conto dei *sector trailer* (in totale sono 15 *sector trailer* che si trovano rispettivamente nell'ultimo blocco di ogni settore quindi ogni 4 blocchi) e passa al blocco successivo nel settore ed effettua l'autenticazione, perché va fatta ad ogni cambio di settore.

Se l'autenticazione fallisce si verifica un caso di errore e tutti e 3 i *LED* vengono fatti lampeggiare per 5 secondi per poi ritornare nello stato di attesa con solo il *LED* rosso attivo.

Nel caso in cui l'autenticazione vada a buon fine attraverso il metodo *MIFARE_READ* della classe *MFRC522* è memorizzato all'interno del *BUFFER* il contenuto del blocco corrente di 16 *byte*, se la lettura fallisce il *loop* viene interrotto e fatti lampeggiare i 3 *LED*.

Nei capitoli precedenti si è visto che la coppia *UID/NUMERO STANZA* è memorizzata per ogni blocco in maniera sequenziale, se durante il prelievo è presente un blocco vuoto in uno dei qualsiasi settore (escluso il settore 0) la funzione si interrompe.

Ricapitolando, si effettua la scansione della memoria della card leggendo ogni singolo blocco fino al numero totale dei blocchi meno uno, con caricamento dei dati all'interno della struttura *card[n]*, fino a quando un blocco vuoto non viene rilevato, se rilevato il *loop* è interrotto con un *break* (non ha senso scansionare tutti e 64 blocchi di memoria, i dati contenuti sono salvati in maniera sequenziale in ordine di successione).

Come ultimo *step* nella posizione quattro e cinque del *buffer* è presente il numero di stanza rappresentato su 2 *byte* (*unsigned int*), utilizzando la funzione di conversione, il numero è convertito in un intero per un *check*, perché devono essere prelevate solo le card che corrispondono al numero di stanza della board in cui stiamo operando. Esempio: ci troviamo alla stanza numero 500 – preleva solo card con numero di stanza 500.

Infine viene aggiornato il blocco corrente per poi passare al successivo, ricominciando la catena dei *checks*.

A fine *loop* il *dump* della memoria e il *download* dei dati è completato.

Lampeggio LED

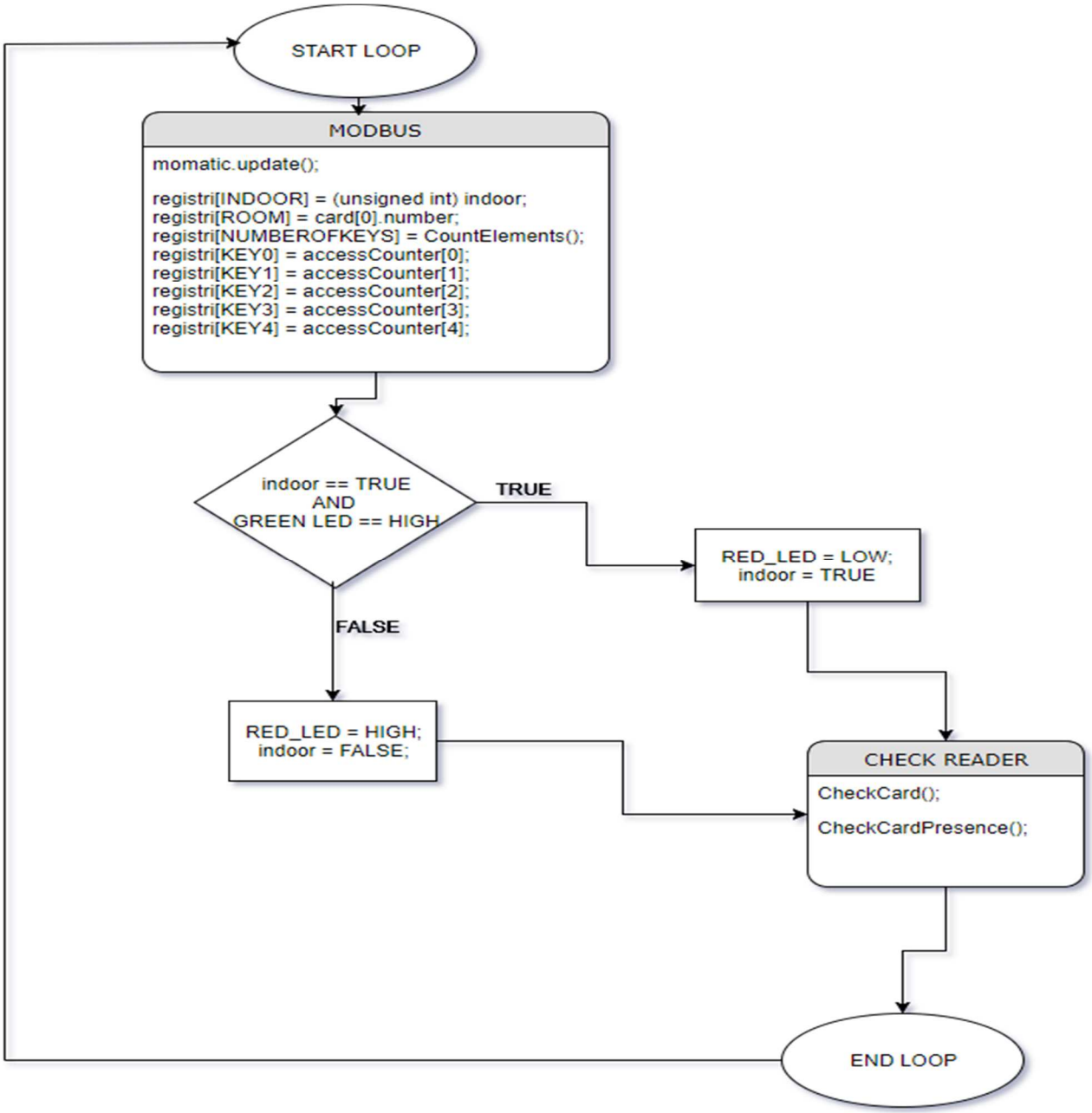
Funzione che ha come argomento il numero del *PIN* da lampeggiare.

E' realizzata con delle *digitalwrite(PIN,HIGH)* e *digitalWrite(PIN,LOW)*; invocate rispettivamente per un tot di millisecondi.

Nell'ultima parte di codice che riguarda la prima sezione dello sketch, vengono istanziati i principali oggetti come l'array di *MFRC522* utilizzato per eseguire le operazioni di gestione dei lettori e lettura *tag/card*, l'oggetto *momatic* della libreria impiegata nella comunicazione tramite *MODBUS*, l'array che conterrà la chiave di *default* usata per l'autenticazione alla *lettura* delle *card/tag* memorizzate all'interno della *card master*, insieme al settaggio dei *PIN* usati dai *LED* in modalità *output*.

3.3.5 Sketch Principale: Terza parte

In questa ultima parte dello sketch è presente la funzione *void loop()* eseguita all'infinito (a meno che non venga resettata la board). Tutto il corpo della funzione quindi è eseguito in un ciclo infinito come se si trovasse all'interno in un *WHILE(TRUE)*, dalla foto con i diagrammi di flusso è possibile vedere l'andamento della funzione e le chiamate alle varie funzioni di gestione elencate durante l'illustrazione della seconda parte dello sketch:



Nella prima parte della funzione come è possibile vedere dal diagramma, sono eseguite tutte quella serie di operazioni riguardanti l'aggiornamento dei dati inviati tramite *MODBUS* alla *board master.*

La funzione *momatic.update()* mette la board nello stato di attesa della richiesta dei dati da parte della *board master,* durante il progetto in particolare nella fase di testing, il master è stato simulato da un pc. Infine l'array *registri* aggiorna con i dati interni alla board i registri che saranno inviati alla *board master*.

Nella seconda parte della funzione vengono effettuati alcuni *check* riguardanti l'accensione dei *LED,* invocando le funzioni di presenza card su entrambi i lettori.

3.3.6 Funzioni realizzate

Una foto illustra tutte le funzioni realizzate all'interno dello sketch:

Funzioni

```
void setup();  // Inizializzazione
void loop();   // Funzione principale
void CheckCard();  // Check presenza card sul primo lettore
void CheckCardPresence();  // Check presenza card sul secondo lettore e utente in stanza
void BlinkLed(unsgined int pin);  // Lampeggio led
void LoadMaster();  // Caricamento master dalla EEPROM
void StampUser();  // Stampa a video tramita linea seriale il contenuto della struttura
void LoadStructFromEEPROM();  // Carica la struttura dalla EEPROM
unsigned int CountElements();  //  Conta il numero di elementi presenti nella struttura
void SaveStructToEEPROM();  //  Salva il contenuto della struttura in memoria EEPROM
void DumpCard();  // Preleva tutte le card/tag memorizzate nella memoria della card MASTER
void ClearEEPROM();  // Cancella il contenuto della EEPROM
bool IsGuestIndoor();   //  Effettua un check per rilevare la presenza in stanza dell'utente
bool IsGuest();  // Effettua un check per abilitare l'apertura della stanza
bool IsMaster();  // Controllo per stabilire se una card è MASTER
unsigned int ByteToInt(unsigned char highB , unsigned char lowB );  // Usata per la conversione da byte a intero
```

3.4 Testing Software

Nella fase di testing (fase di ampia durata) è stato caricato lo sketch principale sulla board *ATMEGA2560*, e sono stati eseguiti vari test di cui:

1. *Polling e lettura dei dati inviati dalla board al master.*
2. *Caso in cui la scheda master veniva tolta dal lettore durante la fase di lettura della memoria (stato di errore con 3 led attivi per vari secondi).*
3. *Lampeggio e accensione corretta dei vari LED a seconda dello stato in cui si trovava la board.*
4. *Card non riconosciuta dal lettore, la board permane nello stato di attesa (LED rosso attivo).*
5. *Il secondo lettore permane nello stato di "card presente" (LED verde sempre attivo) fin tanto che la card non viene rimossa da esso, nel frattempo l'altro lettore è in grado di eseguire le varie operazioni di lettura di altri tag ecc…*
6. *Visualizzazione a schermo, con invio su porta seriale dei dati presenti nella EEPROM e nei vari tags/card.*
7. *Revisione di altri sketch Arduino per altre boards; quali controllo della temperatura, controllo di relè, controllo tapparelle, controllo dei sensori,ecc. Quest'ultimi erano tutti collegati tra di loro ad un master gestito da una board Orange PI, compresa la board che si occupa della gestione delle card/tag realizzata.*
8. *Test dell'intero progetto connesso alla rete locale tramite cavi ethernet.*

Dopo aver connesso tutte le boards tra di loro ci si è occupato della verifica tramite browser dell'effettiva funzionalità del sistema. La board *orange PI* su cui era implementato un website in javascript era connessa ad un router, per poter comandare i vari relè e controllare i dati della temperatura e del numero di accessi dei *tags* anche da remoto tramite un qualsiasi browser per dispositivo. L'interfaccia utente era semplicissima, con dei *"button"* che azionavano i rispettivi relè e ne leggevano lo stato, a seconda della stanza in cui ci si trovava. Veniva inoltre visualizzata la temperatura e il numero di accessi dei *tags* per quella determinata stanza.

Conclusioni

In questa tesi, dopo una panoramica su due tecnologie, i sistemi embedded e arduino, abbiamo visto una loro possibile integrazione. In particolare, lavorando a questo progetto, e così come mostra la parte tecnica di questa tesi, la programmazioni di sistemi embedded richiede una elevata articolazione di logiche di programmazione, nonché una certa praticità del linguaggio informatico specifico. Tuttavia, questi strumenti di sviluppo hanno dei costi di produzione e di conservazione molto bassi rispetto ad altri tipi di sistemi analoghi in commercio oggi. Al di là delle specifiche piattaforme, quello che emerge chiaramente è quanto queste tecnologie, di basso costo ma di elevata complessità abbiano forti potenzialità se pensate in simbiosi tra di loro.

I sistemi embedded da sempre occupano un'importante ruolo all'interno dell'elettronica. I grandi produttori di componenti permettono di sperimentare e progettare prototipi grazie a delle schede modulari e a strumenti per lo sviluppo *ad hoc*. Tuttavia solo recentemente grazie a piattaforme estremamente economiche e versatili (per esempio Arduino) è possibile per chiunque avvicinarsi al mondo dei sistemi embedded. Se da un lato questa nuova tecnologia si concretizza in community di curiosi con le idee più stravaganti e originali a soluzione dei problemi di vita quotidiana nel mondo fisico, dall'altro significa che a ″progettare″ e implementare l'apertura automatica del cancello di casa, una mangiatoia in grado di riconoscere l'animale con cui interagisce e scegliere se somministrargli un alimento piuttosto di un altro, oppure ancora un sistema domotico in grado di controllare i consumi delle utenze e in caso di anomalie avvisare, non saranno necessariamente programmatori altamente specializzati in architetture *specific purpose* con esperienza e conoscenza di elettronica, meccanica e di programmazione concorrente, bensì programmatori inesperti e dilettanti.

Tutto questo ci consente di creare nuove configurazioni inedite ai nostri supporti digitali: come i nostri personali devices, con le loro funzionalità che possano interfacciarsi con sistemi già esistenti, utilizzare i loro standard, aggiungerne delle funzionalità.
E' dalla realizzazione delle piccole idee che prendono vita i grandi progetti.

Bibliografia

[1] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Springler, second edition.

[2] M. Barr. *Programming Embedded Systems : with C and GNU development tools*. O'Reilly, 2007.

[3] O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. - *multi-agent oriented programming with jacamo . Science of Computer Programming* , 78(6),747-761, 2013.

[4] *N. R. C. Committee on Networked Systems of Embedded Computers. Embedded, everywhere: A research agenda for networked systems of embedded computers*. The National Academies Press, 2001.

[5] Y.-H. Fan and J.-O. Wu. *Middleware software for embedded systems. In Advanced Information Networking and Applications Work-shops (WAINA),* 2012 26th International Conference on, pages 61-65, March 2012.

[6] J. A. Holgado-Terriza and J. Vi_udez-Aivar. *A exible java framework for embedded systems. In Proceedings of the 7th International Work-shop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 21-30, New York, NY, USA, 2009. ACM.

[7] E. A. Lee. *Embedded software. volume 56 of Advances in Computers*, pages 55-95. Elsevier, 2002.

[8] F. Mattern and C. Floerkemeier. *From the internet of computers to the internet of things*. In K. Sachs, I. Petrov, and P. Guerrero, editors, From *Active Data Management to Event-Based Systems and More*, volume 6462 of Lecture Notes in Computer Science, pages 242{259. Springer Berlin Heidelberg, 2010.

[9] F. Palermo. *Internet of things done wrong sties innovation*, 2014. http://www.informationweek.com/strategic-cio/executive-insightsand-innovation/internet-of-things-done-wrong-sties-innovation/a/did/1279157

[10] F. Rincon, J. Barba, F. Moya, F. Villanueva, D. Villa, J. Dondo, and J. Lopez. *System-level middleware for embedded hardware and software communication. In Intelligent Solutions in Embedded Systems*, 2007 Fifth Workshop on, pages 127-138, June 2007.

[11] A. Sorici, O. Boissier, G. Picard, and A. Santi. *Exploiting the jacamo framework for realising an adaptive room governance application.* In Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, - VMIL'11, SPLASH '11 Workshops, pages 239-242, New York, NY, USA, 2011. ACM.

[12] D. Uckelmann, M. Harrison, and F. Michahelles. *Architecting the Internet of Things*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[13] Arduino. http://arduino.cc

[14] Arduino RFID Library for MFRC522. https://github.com/miguelbalboa/rfid

[15] Modbus. http://www.modbus.org/

[16] Modbus Automative. http://www.modbus-ida.org/

[17] Modbus Sketch. https://github.com/charlesbaynham/simple-modbus

[18] Simple Modbus documentation. https://code.google.com/archive/p/simple-modbus/

[19] Datasheet MIFARE CLASSIC EV1. http://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf

[20] Mifare Classic 1k Datasheet. http://www.mouser.com/ds/2/302/MF1S503x-89574.pdf

[21] Rafael H. Bordini, Jomi Fred Hubner, and Michael Wooldridge. *Programming multi-agent system in AgentSpeak using Jason*. Wiley, 2007.

[22] Alessandro Ricci, Michele Piunti, and Mirko Viroli. *Environment programming in multi-agent systems: an artifact based perspective. Autonomous Agents and Multi-Agent Systems*, 23(2),158-192, 2010.

[23] S. Franklin and A. Graesser. *Is it an agent, or just a program ? , A taxonomy for autonomous agents. In Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, ECAI '96, pages 21-35, London, UK, UK, 1997. Springer-Verlag.

[24] Sergio Congiu, *Architettura degli Elaboratori*, Patròn Editore, Bologna, 2007.

[25] Atmel R , *ATmega328P datasheet*, 2009.

[26] Honeywell, *HIH-4000 series, Humidity Sensors*, 2005.

[27] Texas Instruments, *LM35, Precision Centigrade Temperature Sensor*, 2000.

[28] Richard H. Barnett, Sarah Cox, Larry O'Cull, *Embedded C Programming and the Atmel AVR, 2nd Edition*, Thomson Delmar Learning, 2007.

[29] Atmel R Site: http://www.atmel.com

[30] AVRfreaks Site: http://www.avrfreaks.net

[31] Datasheet Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V: http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

[32] ELECTRONICS,*RFID*, http://electronics.howstuffworks.com/gadgets/high-tech-gadgets/rfid3.htm

[33] THEIET, <RFID>, http://www.theiet.org/factfiles/it/rfid-page.cfm

[34] SMARTCARDBASIC,TIPOLOGIE,SMARTCARD, http://www.smartcardbasics.com/smart-card-types.html

[35] NXP SEMICONDUCTORS, <HOME PAGE> http://www.nxp.com/

[36] ISO, <HOME PAGE> www.iso.org/

[37] ENGADGET, <NXP GESTURE SMART CARD>

http://www.engadget.com/2012/01/11/nxp-gesture-smart-card-nfc/

[38] D. CHAUHAN, <APDU - PRATICA>

http://www.devshed.com/c/a/Practices/Smart-Cards-An-Introduction/5/

[39] J. SIMON,<SMARTCARD LIFECYCLE>

http://www.jguru.com/faq/view.jsp?EID=471391

[40] IBM, <SMART CARD RED BOOK>

http://www.redbooks.ibm.com/redbooks/pdfs/sg245239.pdf

[41] INNOVATRON, <CALYPSO SPECIFICHE>

http://www.innovatron.fr/CalypsoFuncSpecification.pdf

[42] ICAO, <DOCUMENTO 9303>

http://www.icao.int/Security/mrtd/Pages/Document9303.aspx

[43] NIST, <SMARTCARD FIPS>

http://csrc.nist.gov/groups/SNS/smartcard/

[44] Casaleggio Associati,*L'evoluzione di Internet of Things*,Febbraio

2011, voluzione-di-internet-of-things, 2012

http://www.slideshare.net/casaleggioassociati/

[45] Justin Lahart, *Taking an Open-Source Approach to Hardware,* 27 novembre 2009,The Wall Street Journal.

[46] Wikipedia, *Arduino (hardware)*, 2012 http://it.wikipedia.org/wiki/Arduino(hardware)/ , 2012.

[47] Bonifaz Kaufmann, *Design and Implementation of a Toolkit for the Rapid Prototyping of Mobile Ubiquitous Computing*, Alpen-Adria-Universität Klagenfurt, August 2010.

[48] Massimo Banzi,BetaBook, *il manuale di Arduino: Cap. 3 - Un po' di storia di Arduino*.Apogeo. http://arduino.apogeolab.it/03-un-po-di-storia-di-arduino/, 12 luglio 2011