

Università degli studi di Modena e Reggio  
Emilia

Dipartimento di Scienze e Metodi dell’Ingegneria

*Corso di Laurea Magistrale in Ingegneria Meccatronica*

***Progettazione di un sistema automatico di  
controllo accesso ad aree riservate***

**Studenti:**

Bartolotta Miriam  
Bertolani Mattia  
Fantuzzi Denise  
Giovanardi Christian  
Lasagni Simone  
Pellegrino Andrea

**Docenti:**

Prof. Bertacchini Alessandro  
Prof. Santinelli Paolo

## Sommario

1	Introduzione .....	3
1.1	Idea originale .....	3
1.2	Layout componenti.....	4
1.3	Descrizione funzionale.....	6
1.4	Descrizione Hardware .....	9
1.4.1	Raspberry Pi.....	10
1.4.2	FPGA .....	11
1.4.3	Microcontrollori.....	13
1.4.4	Altri componenti.....	18
2	Raspberry.....	22
2.1	Invio e ricezione carattere di alive e altri messaggi.....	24
2.2	Interfaccia grafica .....	30
2.3	Invio codice di apertura porta al PIC Master .....	37
2.4	Invio database all'FPGA.....	39
2.5	Approfondimento codice dell'interfaccia grafica e del database .....	42
3	FPGA .....	53
3.1	Gestione accessi .....	55
3.1.1	COM_Raspberry.....	58
3.1.1.1	Writer.....	61
3.1.2	Safe_Mode.....	68
3.1.2.1	State_Machine_Pagine.....	71
3.1.2.2	Ins_ID .....	80
3.1.2.2.1	Inserimento_ID .....	82
3.1.2.3	Finder.....	86
3.1.2.4	Ins_Psw .....	93
3.1.2.4.1	Inserimento_password.....	95
3.1.2.5	Check_psw .....	99
3.1.2.6	Inserimento_porta.....	101
3.1.3	COM_PIC.....	105
3.1.3.1	Sender.....	109
3.1.3.2	Alarms.....	113
3.1.4	Tastierino .....	116
3.2	Altri blocchi.....	119
3.2.1	AliveCharacter .....	119
3.2.2	Check alive .....	120
3.2.3	Ck2ck.....	121
3.2.4	Counter .....	123
3.2.5	Debouncer .....	124
3.2.6	Dec2_4.....	126
3.2.7	Delay .....	127

3.2.8	FFJK .....	128
3.2.9	LCD .....	129
3.2.10	Mux4_1 .....	137
3.2.11	Packet_Transfer .....	138
3.2.12	RAM .....	143
3.2.13	Registro .....	146
3.2.14	Tast2Bin .....	148
3.2.15	Timer .....	150
3.2.16	Transition_Finder .....	152
3.2.17	UART .....	154
4	PIC .....	162
4.1	PIC Master .....	165
4.1.1	Main Master .....	165
4.1.2	Configuration bits .....	169
4.1.3	Funzione Initial_Reset() .....	172
4.1.4	Funzione general_settings() (Master) .....	179
4.1.5	CAN Configuration Mode .....	187
4.1.6	CAN Normal Mode + altre configurazioni .....	195
4.1.7	Funzione Initial_Test() (Lato Master) .....	197
4.1.8	EUSART_Alive_check .....	200
	4.1.8.1 Gestione dello switch: Switch_manager .....	201
	4.1.8.2 Gestione del carattere di ALIVE: EUSART_ImAlive .....	203
4.1.9	EUSART_Status_Display .....	205
4.1.10	EUSART_FSM .....	207
	4.1.10.1 Gestione degli identificativi: Junk_manager .....	209
	4.1.10.2 Gestione dei dati: Data_manager .....	211
4.1.11	Routine di ricezione CAN (Master) .....	214
4.1.12	Routine di Alive CAN .....	218
4.1.13	CAN Errors Handling (Master) .....	232
4.2	PIC SLAVE .....	239
4.2.1	Main Slave i-esimo .....	239
4.2.2	General Settings (Slave) .....	243
4.2.3	Routine di ricezione CAN (Slave) .....	246
4.2.4	Routine di accesso .....	251
4.2.5	Routine di gestione stato porta .....	273
4.2.6	CAN Errors Handling (Slave) .....	277
4.2.7	PIC porta principale .....	278
5	Conclusioni e considerazioni di progetto .....	279
Appendice A:	File supplementari per il PIC Master .....	281
Appendice B:	File supplementari per i PIC Slave .....	285

# 1 Introduzione

## 1.1 Idea originale

Sempre più aziende investono nella sicurezza in senso lato, dalla sicurezza informatica agli accessi delle aree riservate. Questo progetto si pone come obiettivo quello di creare un sistema di gestione di accessi informatizzato; in particolare, il sistema è pensato per un laboratorio di ricerca, nel quale gli accessi alle diverse aree sono talvolta limitati ad un personale ristretto.

Il sistema dovrà agire gestendo il blocco e lo sblocco di porte all'interno di un edificio, le quali conducono ad aree diverse all'interno dell'edificio stesso. Le credenziali delle persone a cui è permesso accedere saranno salvate in un apposito database (contenuto in un dispositivo master di sistema), la cui modifica (aggiunta/rimozione di utenti; promozione/declassamento degli stessi) è consentita solo da utenti qualificati come amministratori.

Per poter usufruire delle funzionalità del sistema, l'utente dovrà prima autenticarsi con ID e password, che sono chiavi di autenticazione formate rispettivamente da 6 e 4 cifre numeriche. Ci sono due tipi di utenti:

- **L'utente standard**, il quale potrà soltanto selezionare un'area (simboleggiata da una porta) a cui accedere.
- **L'amministratore**, che potrà anche selezionare alcune opzioni relative alla gestione del database, come accennato precedentemente.

Dopo aver selezionato la porta desiderata, viene sbloccata la porta principale di accesso, oltre la quale si trovano tutte le altre porte. Questo, tuttavia, non avviene nei casi in cui:

- Una porta dovesse rimanere aperta per troppo tempo;
- Sia intercorso un lasso di tempo troppo breve tra due richieste d'accesso sulla medesima porta.

L'utente dovrà quindi recarsi nei pressi della porta interessata, dotata a sua volta di un'interfaccia utente, in cui dovrà digitare la propria password una seconda volta. Quando questo accade, la porta in questione verrà sbloccata, consentendo quindi l'accesso. In questa fase l'utente avrà a disposizione tre tentativi.

La particolarità di questo progetto è data dal fatto che è stata prevista una **modalità di funzionamento ridotta**: anche in caso di guasto del master di sistema, si continuerà a garantire la possibilità di accedere alle aree richieste. Pertanto, al master di sistema vero e proprio se ne affiancherà un altro, che dovrà gestire il sistema in modalità ridotta quando l'unità che governa l'interfaccia non è collegata. Per modalità ridotta si intende la capacità del sistema di continuare ad operare normalmente nel riconoscimento degli utenti, ma senza la possibilità di apportare modifiche al database. Non sarà quindi possibile aggiungere o rimuovere utenti o modificare i permessi associati agli stessi.

## 1.2 Layout componenti

A livello funzionale si possono identificare diverse parti:

- Un dispositivo master di sistema principale, responsabile della gestione del database e di tutte le funzionalità principali del sistema;
- Un dispositivo master di sistema ausiliario, che entra in funzione soltanto in caso di modalità di funzionamento ridotta. Per questo motivo, nel progetto dovrà essere previsto anche un sistema di comunicazione point-to-point tra i due master di sistema, che all'occorrenza possa garantire il trasferimento dell'intero database;
- Un dispositivo che possa gestire le singole porte (uno per porta);
- Specifici sensori e attuatori per il controllo di una porta, opportunamente connessi al relativo dispositivo di gestione;
- Un dispositivo intermedio, che funga da tramite tra master di sistema e singole porte. In particolare:
  - Dovrà sempre comunicare direttamente con un solo master di sistema alla volta, ragion per cui bisognerà contemplare al più due reti di comunicazione distinte point-to-point, che funzionino in modo complementare;
  - Dovrà comunicare contemporaneamente con tutte le porte per poterle gestire correttamente, facendo le veci di un supervisore locale. Per questo motivo occorrerà una linea di comunicazione multi-point;
- Un'adeguata interfaccia utente, che possa gestire le autenticazioni, opportunamente collegata ai master di sistema.

Una volta definiti questi blocchi, è stato possibile scegliere i componenti e i protocolli di comunicazione associati:

- L'unità master di sistema primaria è un Raspberry Pi, mentre l'unità usata per la modalità ridotta è una scheda FPGA;
- Il dispositivo intermedio è un microcontrollore, definito d'ora in poi PIC Master;
- I dispositivi di gestione delle porte sono a loro volta dei microcontrollori, chiamati PIC Slave (si precisa che tutti i dispositivi PIC del sistema sono modello 18F4580);
- La comunicazione tra i vari microcontrollori avviene tramite protocollo CAN;
- La comunicazione tra Raspberry Pi e PIC Master, tra FPGA e PIC Master e tra Raspberry pi e FPGA avviene tramite protocollo RS-232. Raspberry Pi e FPGA comunicano con il PIC Master alternativamente tramite un multiplexer;
- L'interfaccia utente per l'autenticazione è formata da un monitor dedicato più una tastiera e un mouse (per il funzionamento completo), oppure direttamente dallo schermo LCD in dotazione all'FPGA più un tastierino numerico (per il funzionamento ridotto). La segnalazione di eventi importanti (ad esempio, eventuali errori) è assegnata a diversi LED.

L'architettura del sistema vede dunque il PIC Master come dispositivo centrale di collegamento. Esso si interfaccia con il Raspberry Pi o con l'FPGA, a seconda di chi è collegato, attraverso il multiplexer. La scelta del canale di comunicazione è gestita dal PIC stesso.

Il PIC Master è a sua volta inserito all'interno di una rete CAN, di cui è il supervisore. A questa rete sono connessi altri cinque nodi corrispondenti ad altrettanti PIC Slave, uno per ogni porta.

I PIC Slave comandano a loro volta alcuni componenti:

- L'interfaccia utente della singola porta è composta da due LED di segnalazione (verde e rosso) e un tastierino numerico. La porta principale ha solamente i due LED, non il tastierino;
- Una serratura che viene aperta su richiesta, altrimenti normalmente chiusa;
- Un sensore magnetico che rileva lo stato della porta.

La figura sottostante (Fig. 1.2-1) schematizza quanto detto:

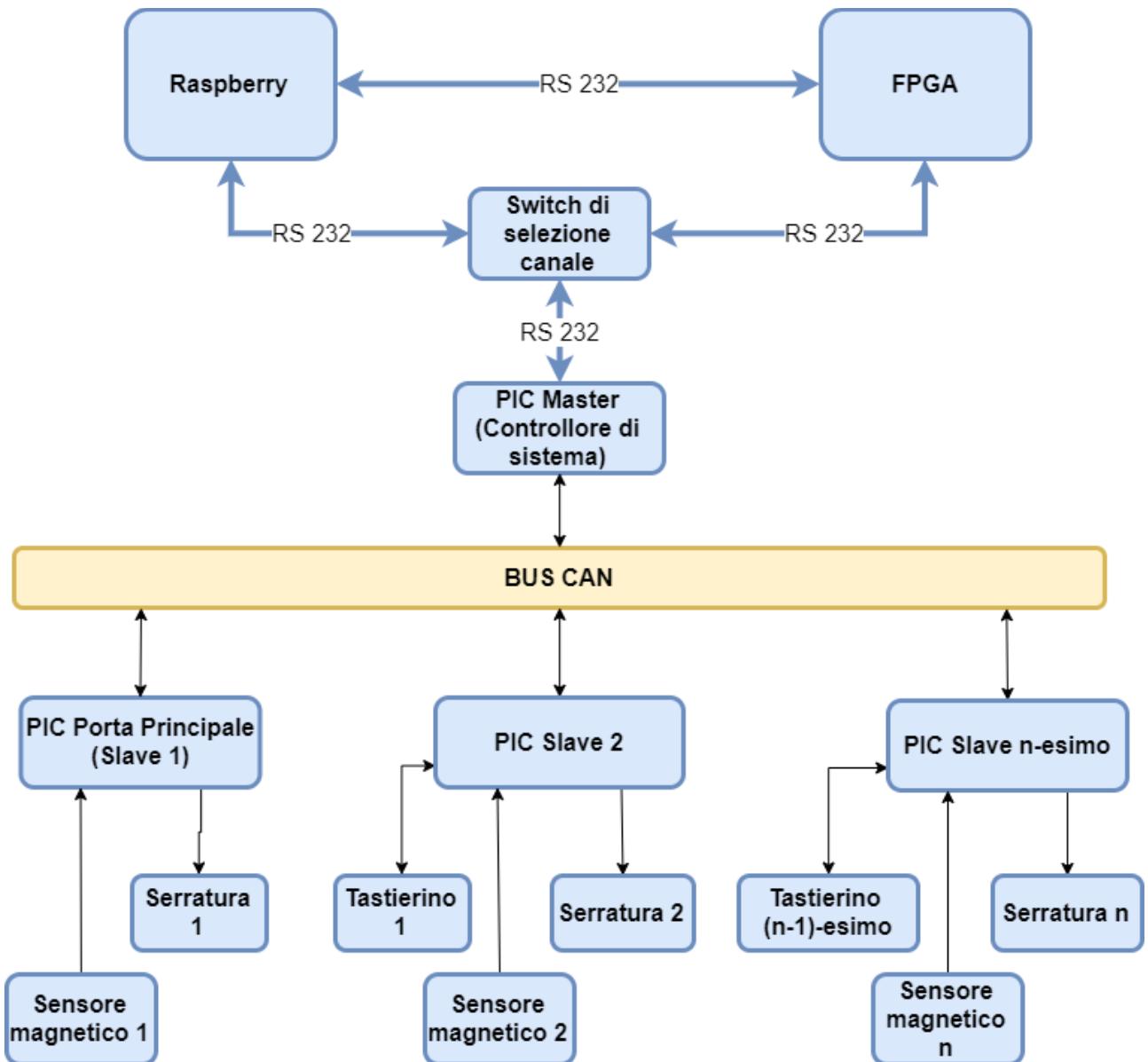


Fig. 1.2-1 Schema concettuale dei componenti del sistema

### 1.3 Descrizione funzionale

Per implementare al meglio la funzione principale descritta nel *paragrafo 1.1*, il sistema proposto dovrà essere in grado di svolgere alcune ulteriori sotto-funzioni/funzioni accessorie:

- Gestione degli utenti, intesa come la modifica dei dati presenti del database (funzionalità riservata ai soli amministratori);
- Trasferimento del database da Raspberry Pi a scheda FPGA, per garantire la continuità di funzionamento in caso di guasto del master di sistema primario;
- Trasferimento della password alla porta d'interesse;
- Validazione della password tramite matching (massimo tre tentativi);
- Apertura delle porte d'accesso;
- Rilevazione dell'effettiva chiusura delle porte e conseguente controllo degli accessi (funzione necessaria per evitare fallo nella sicurezza);
- Controllo sul funzionamento delle reti di comunicazione (con il meccanismo del carattere di Alive);
- Segnalazione di eventuali guasti/errori e generazione/gestione dei messaggi d'errore;
- Si preannuncia fin da ora che il sistema è stato pensato per prevedere, nel caso di applicazione reale, un sistema meccanico per l'apertura delle porte dall'interno verso l'esterno (es: maniglioni antipanico) che per ragioni economiche non è stato modellato nel prototipo (questo è anche il motivo per cui le uscite del personale dall'interno verso l'esterno non rientrano tra i casi contemplati dal nostro sistema);

I seguenti diagrammi a blocchi illustrano quanto detto (Fig. 1.3-1, 1.3-2 e 1.3-3):

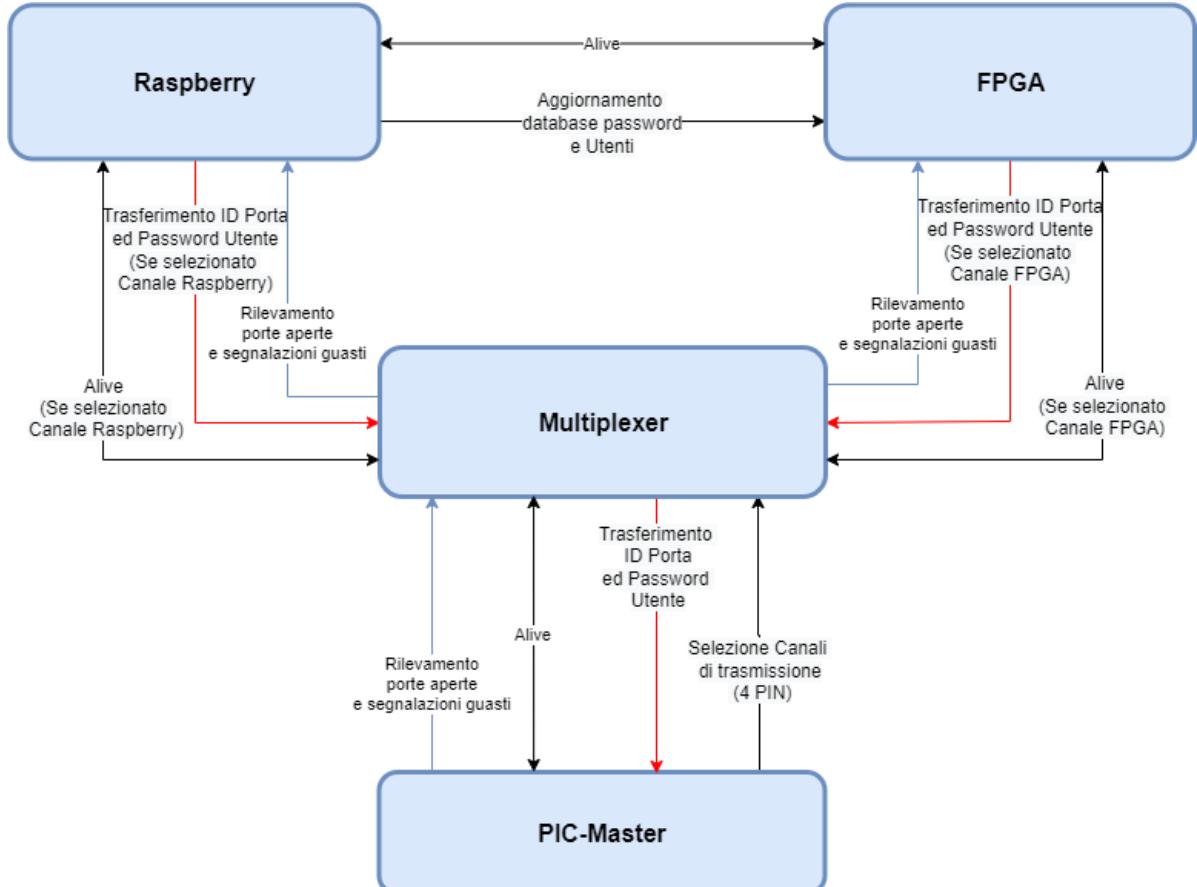


Fig. 1.3-1 Schema a blocchi del sistema (lato master di sistema)

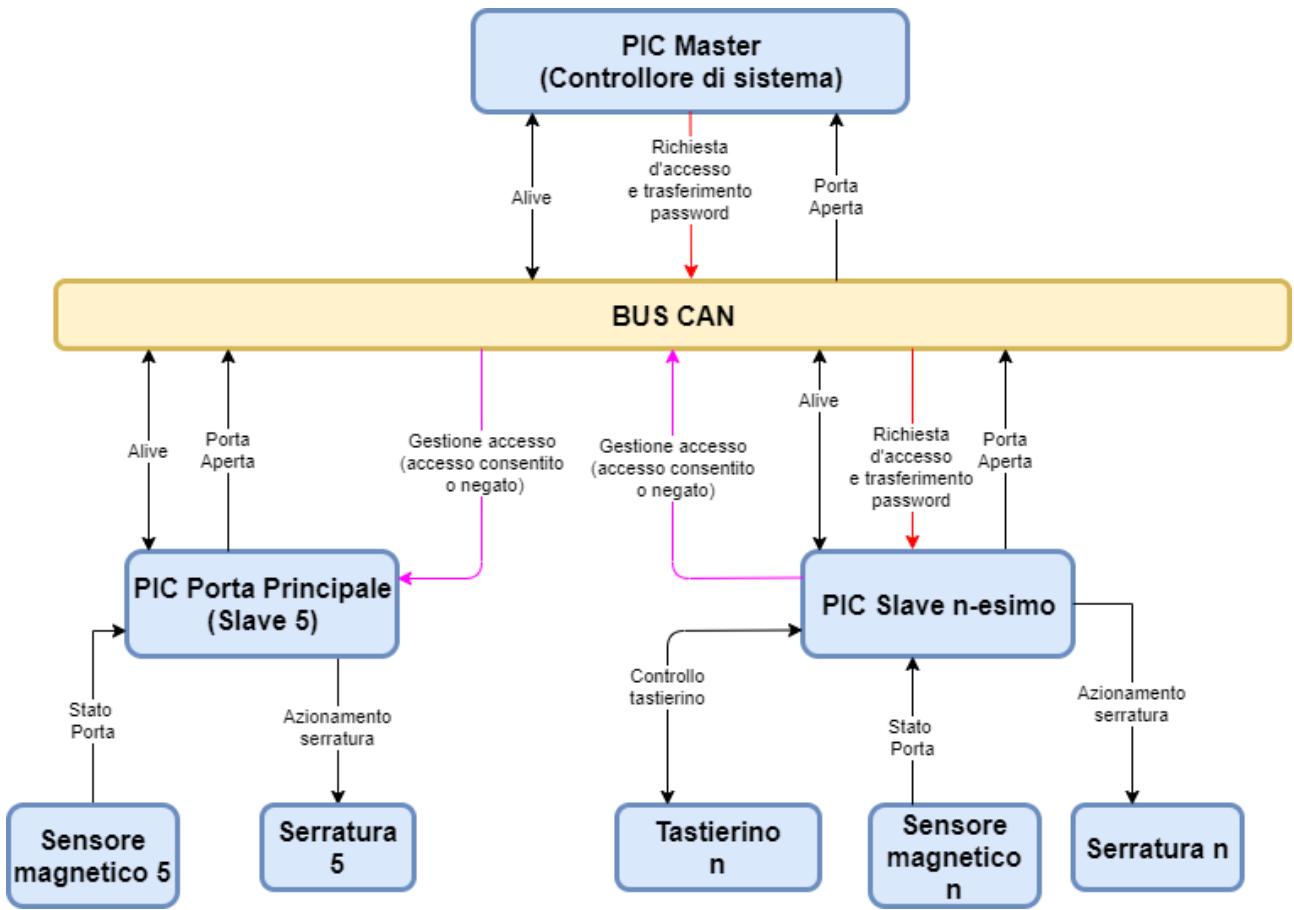


Fig. 1.3-2 Schema a blocchi del sistema (lato porte)

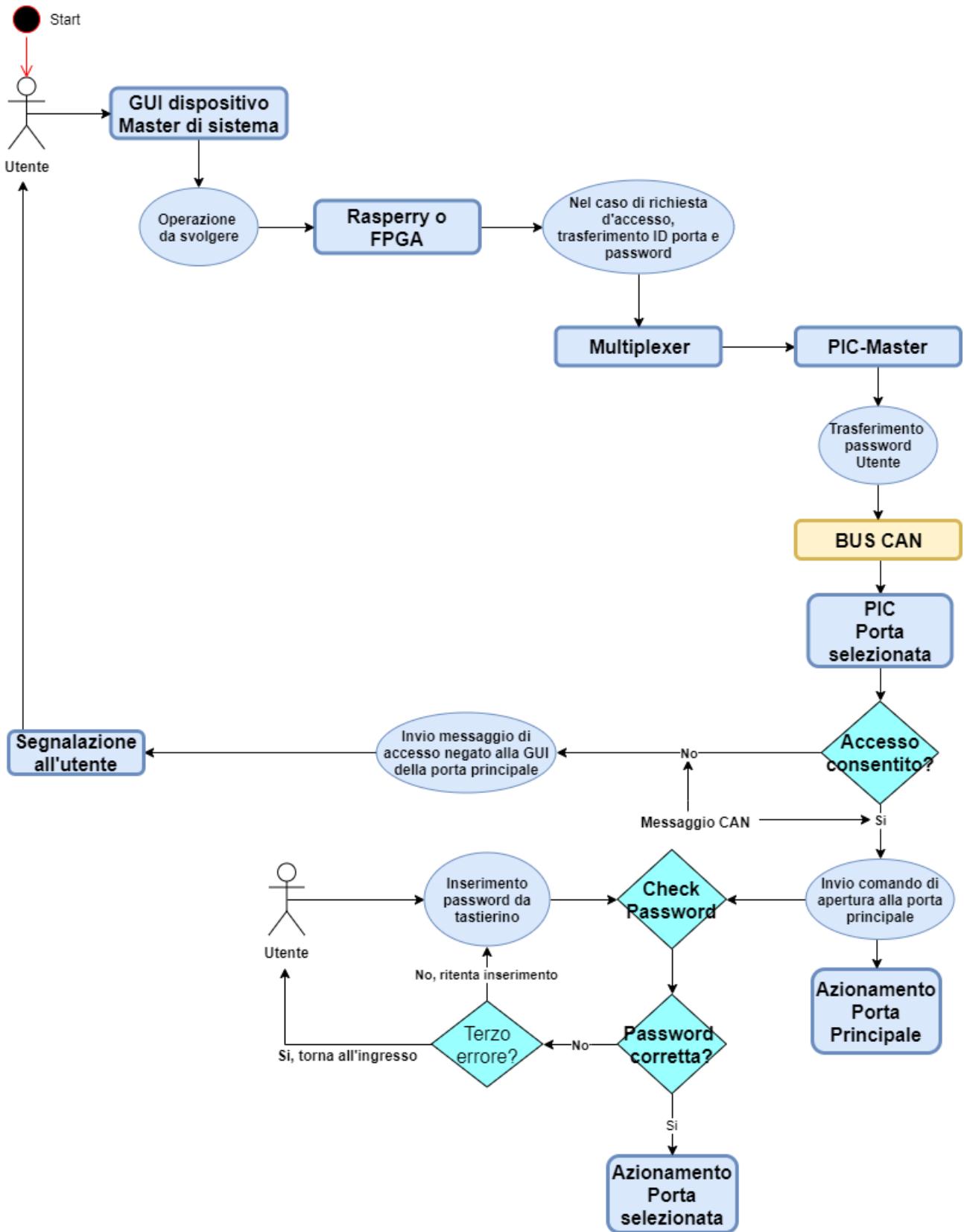


Fig. 1.3-3 Diagramma funzionale generale relativo all'accesso alla porta

## 1.4 Descrizione Hardware

Per validare il corretto funzionamento del sistema si è costruito un modello di legno costituito da quattro porte allineate sul lato lungo (in giallo in Fig. 1.4-1) che, idealmente, danno accesso ad altrettante stanze e una porta (in blu), situata sul lato corto, che consente l'accesso a un corridoio da cui è possibile accedere alle altre stanze. La porzione di piano della stanza comune funge da supporto per i tastierini che per lavorare correttamente necessitano di una superficie rigida di appoggio.

Le porte sono sagomate in modo da favorirne il montaggio nella posizione corretta e per potersi aprire in un unico verso. In questo modo si favorisce la sicurezza della struttura contro eventuali tentativi di accesso non autorizzato. Nella fig. 1.4-1 sono inoltre mostrate le posizioni predefinite per serrature e sensori. Da notare che non sono presenti divisorie tra le stanze, in quanto non necessarie alla valutazione del sistema.

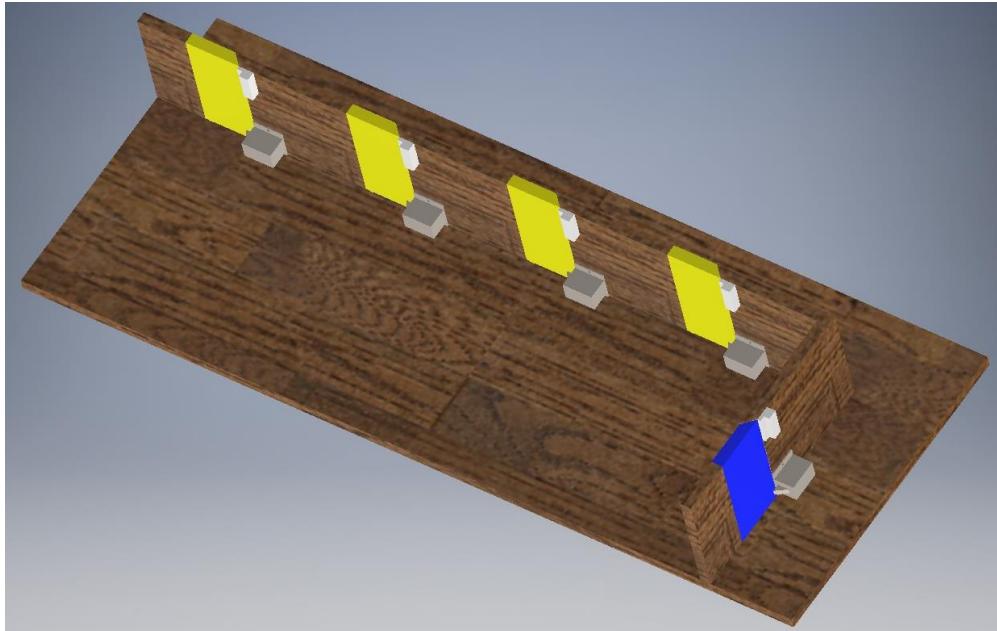


Fig. 1.4-1 Prototipo

I microprocessori sono saldati su millefori (una per PIC) così come lo sono i cavi che costituiscono le linee della rete CAN. Il multiplexer è stato saldato sulla stessa millefori del PIC Master, al fine di ridurre la lunghezza dei cavi e in conseguenza i disturbi. All'interno del modello i PIC Slave si trovano in prossimità delle porte mentre il PIC Master è l'unico che potrebbe, idealmente, trovarsi in qualsiasi punto del sistema, visto il proprio ruolo di mero supervisore e coordinatore.

Per fornire un feedback all'utente sulla corretta o errata scrittura della password si è deciso di integrare nel sistema due LED per ogni porta/millefori: uno verde per il caso di password corretta, uno rosso per il caso di password errata. Un sistema di amplificazione con transistor è stato posto a monte di ciascuna serratura, che altrimenti non sarebbe in grado di restare aperta a causa della bassa corrente erogata dal pin del PIC Slave a cui è collegata.

Le linee di alimentazione e ground non connettono le millefori tra loro, bensì queste sono direttamente connesse all'alimentatore tramite diversi capicorda. In questo modo è stata garantita una rete per l'alimentazione in parallelo delle diverse millefori.

FPGA e Raspberry Pi hanno alimentazioni separate e indipendenti, si è preferito infatti lasciare questi ultimi svincolati dal resto del sistema, per ridurre le dimensioni stesse della base su cui poggia il modello e per favorirne il trasporto. In un contesto applicativo più realistico, sarebbero posti all'esterno del corridoio nelle vicinanze della porta principale.

#### 1.4.1 Raspberry Pi

Il Raspberry è una scheda elettronica che funge da computer, il cui sistema operativo, Raspian, è salvato su una scheda SD. Il dispositivo non dispone di altre unità di memorizzazione non volatili. È facilmente utilizzabile collegandovi schermo, tastiera e mouse. Per il nostro progetto è stata utilizzata anche la porta USB, di cui il Raspberry dispone, per permettere al dispositivo di comunicare tramite protocollo RS-232 con il multiplexer del PIC Master (utilizzando un cavo USB-seriale). Il dispositivo è provvisto anche di diversi pin general purpose, tra questi, alcuni sono stati utilizzati per l'accensione di alcuni LED, altri due specifici pin sono dedicati alla comunicazione seriale e sono quindi stati usati per implementare una comunicazione di tipo seriale (protocollo RS-232) tra scheda FPGA e Raspberry. La figura sottostante (Fig. 1.4.1-1) mostra lo schema dettagliato dei componenti collegati al Raspberry, sapendo che la tensione di uscita di ciascun pin GPIO può variare tra 0 e 3,3 V.

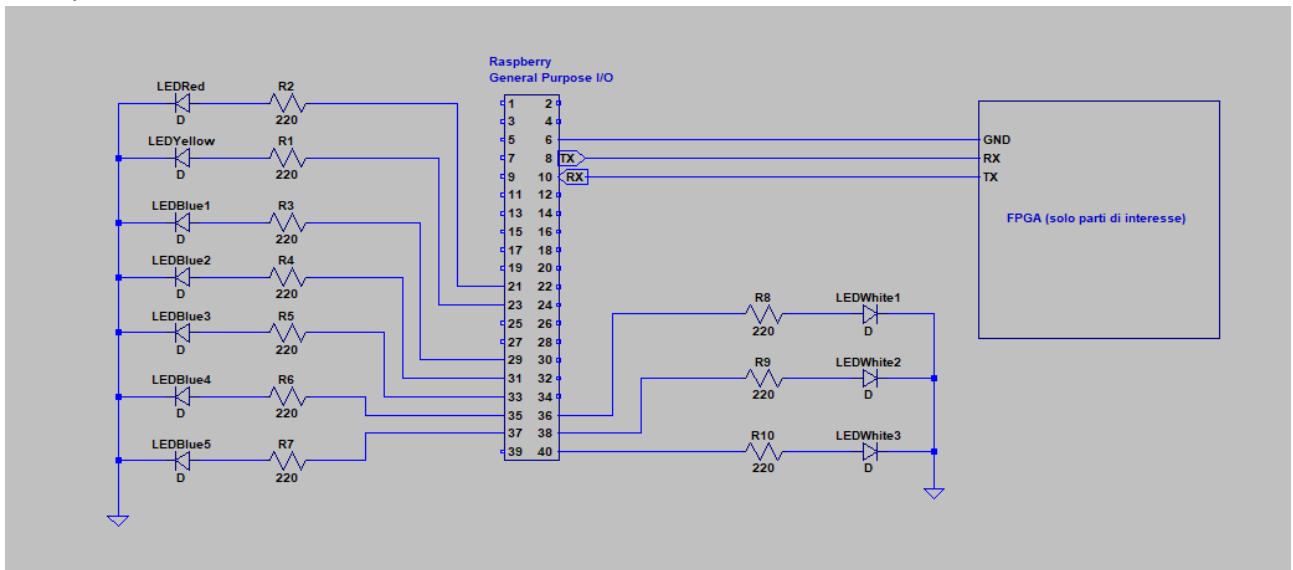


Fig. 1.4.1-1 Schema elettrico Raspberry

La scelta di un Raspberry Pi è dovuta anche al maggior quantitativo di memoria che possiede rispetto ai PIC18F4580 e a FPGA. Questa scelta deriva, in ottica futura, dalla possibilità (tutt'altro che remota) che il sistema si trovi a dover gestire un quantitativo di utenti ben superiore a quello ipotizzato in fase di prototipazione. Inoltre, l'uso del Raspberry consente una più facile e veloce implementazione di eventuali aggiornamenti al sistema e/o alla sua interfaccia utente.

Si rammenta comunque che questo dispositivo è l'unico elemento/componente del sistema che può essere rimosso senza pregiudicarne il funzionamento. Infatti, in sua assenza subentrerà la scheda FPGA come controllore degli accessi.

### 1.4.2 FPGA

L'FPGA è un dispositivo a logica programmabile che contiene una serie di risorse logiche immerso in una matrice di interconnessioni configurabili. Le risorse logiche sono delle piccole memorie che realizzano tabelle di verità per fare reti combinatorie. L'FPGA è installato nella board “DE2-115” fornita da Altera. Questo package include, in particolare:

- L'FPGA “EP4CE115F29C7” della famiglia “Cyclone IV E”;
- Una periferica per la comunicazione seriale RS-232;
- Un modulo LCD 16x2;
- Un oscillatore al quarzo con frequenza pari a 50 MHz;
- Slide switch e push-buttons;
- LED (rossi e verdi);
- Pin general-purpose di input/output;

Alla strumentazione già presente sulla board è stato aggiunto un tastierino numerico (differenti da quelli che verranno introdotti nel seguito) collegato tramite i pin di general purpose.

L'insieme di questi elementi permetterà di implementare la modalità ridotta, in caso di disconnessione del Raspberry.

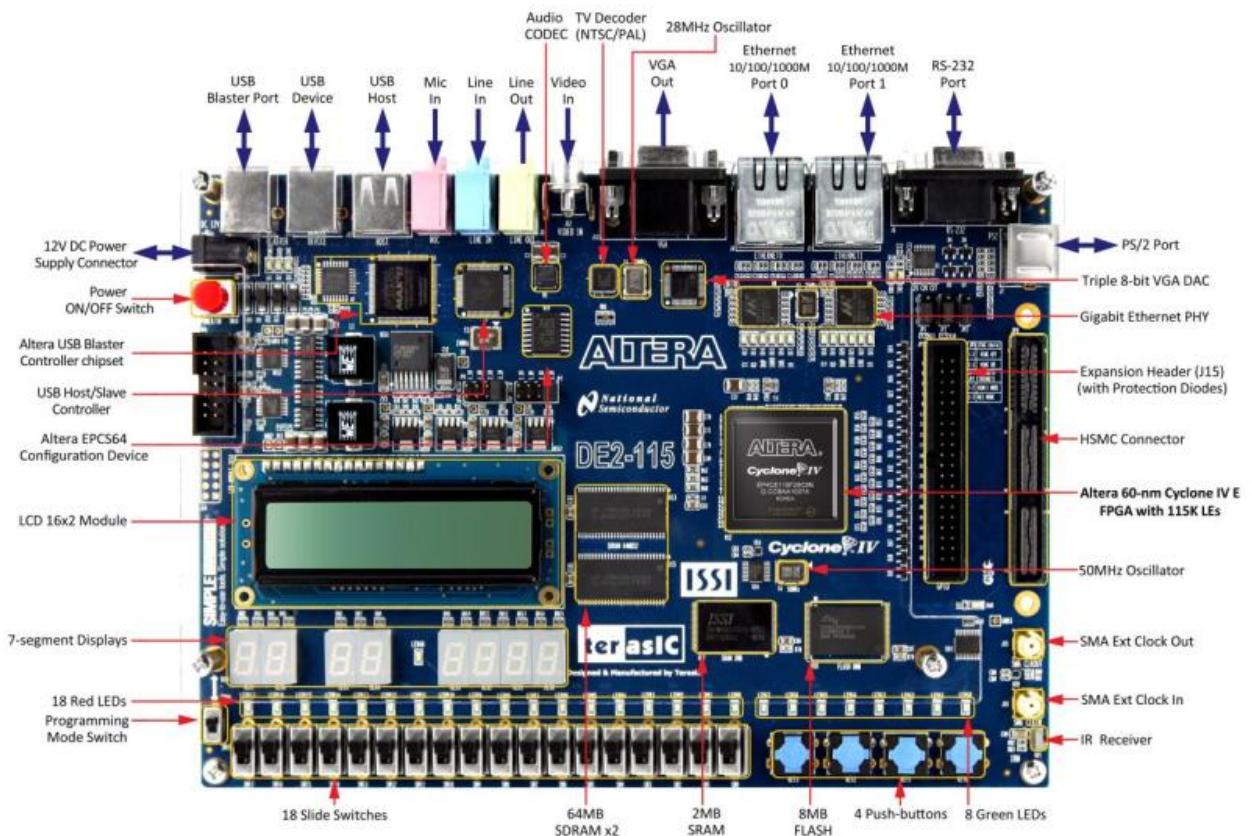


Figura 1.4.2-1 Schema della board DE2-115

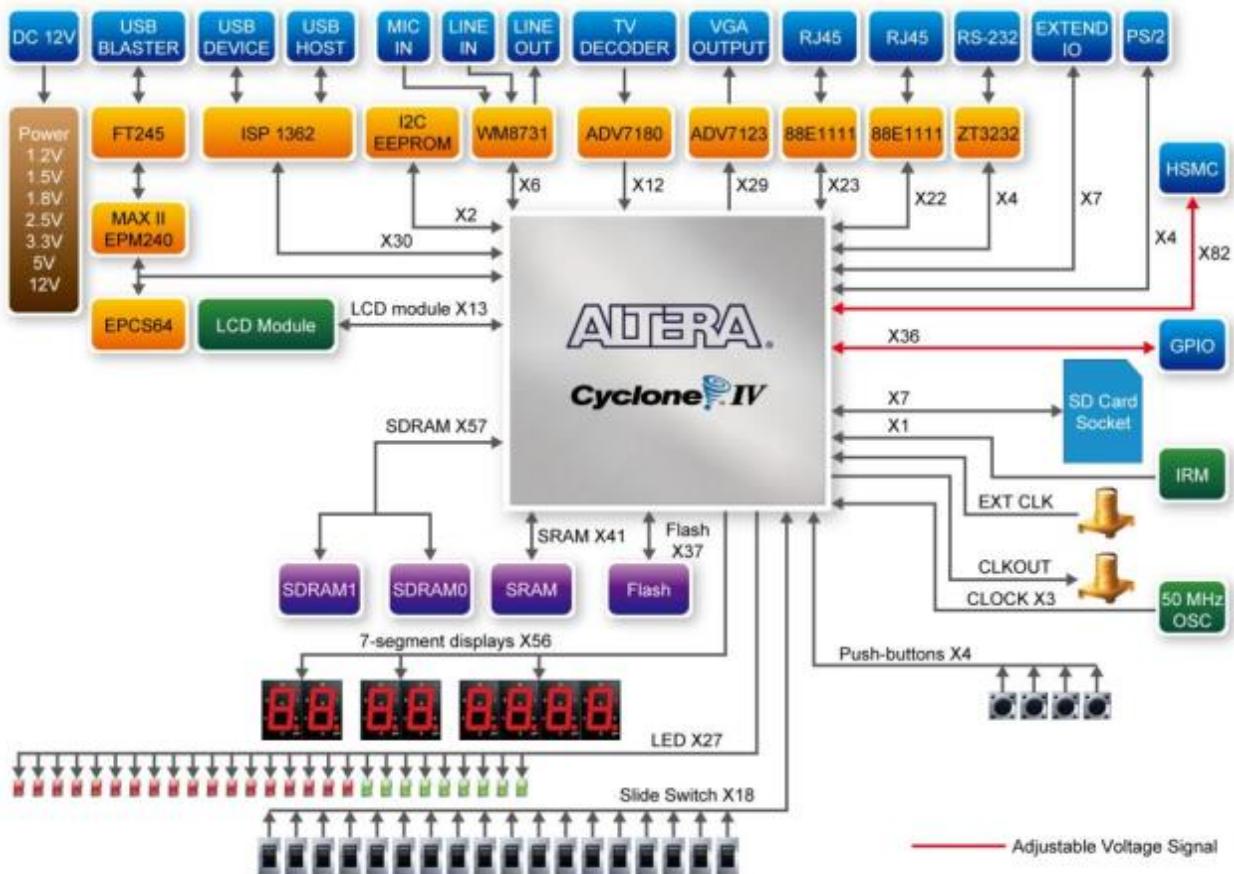


Figura 1.4.2-2 Diagramma a blocchi dell'FPGA

Si riporta nella figura seguente lo scherma di massima dei collegamenti con le periferiche della scheda FPGA.

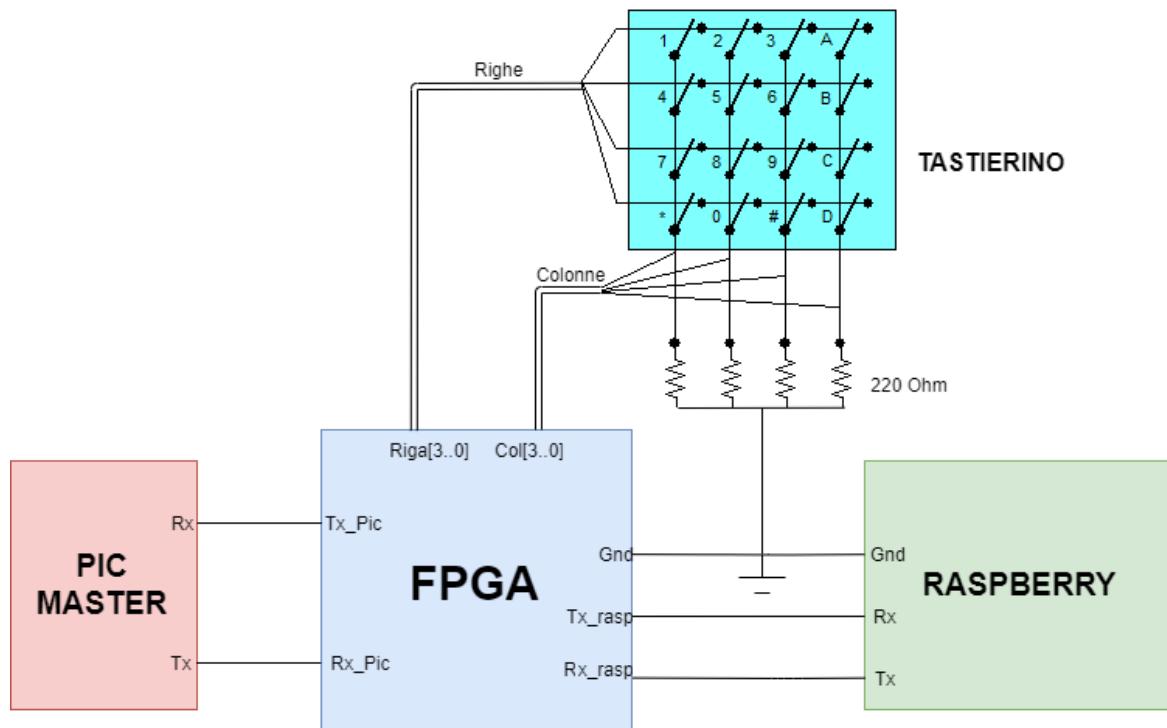


Figura 1.4.2-3 Schema collegamenti elettrici FPGA e altre parti del sistema

### 1.4.3 Microcontrollori

Come detto in precedenza i microcontrollori utilizzati sono tutti dello stesso modello (PIC18F4580 Microchip). Per la realizzazione del sistema sono stati inoltre utilizzati:

- degli High-Speed CAN Transceiver MCP255 Microchip, per la gestione della rete CAN;
- Multiplexer CD74HC4053 Texas Instruments, per veicolare l'informazione tra PIC Master e Raspberry Pi o tra PIC Master e FPGA.

La figura seguente mostra le connessioni del PIC Master verso l'esterno.

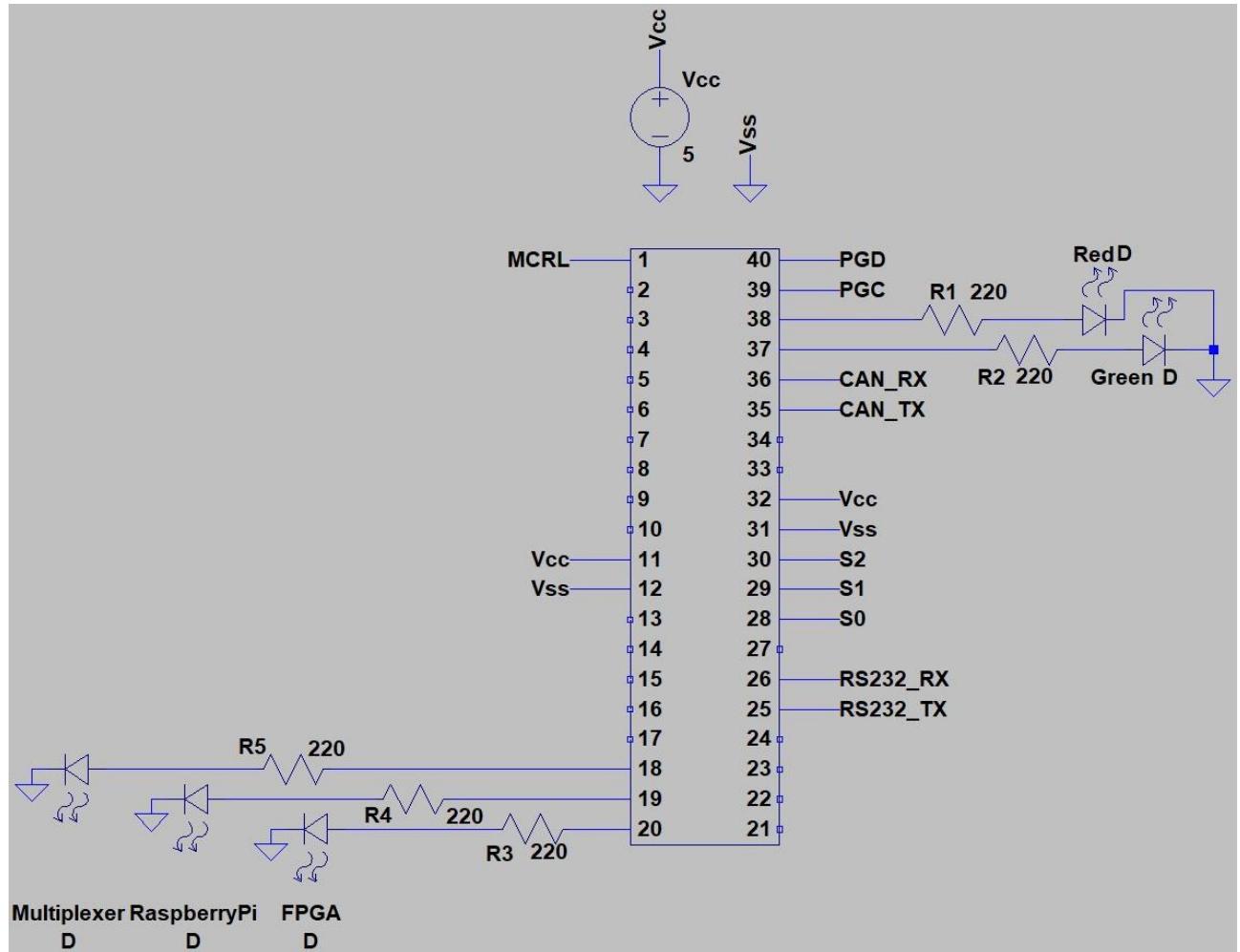


Fig. 1.4.3-1 Schema elettrico PIC Master

Ogni PIC Slave è connesso a sua volta con tre periferiche:

1. Serratura

Per simulare una porta con serratura automatica abbiamo utilizzato una serratura elettrica a solenoide (Fig. 1.4.3-2):

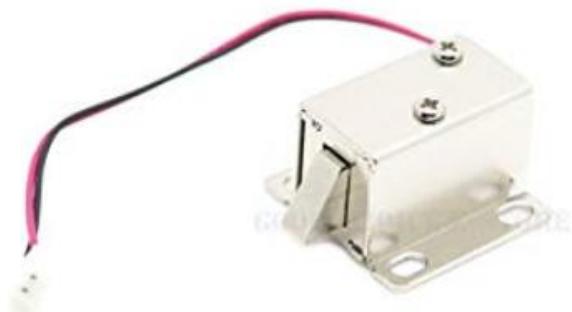


Fig.1.4.3-2 Serratura elettrica a solenoide

Come si evince dal nome, il componente funziona grazie ad un solenoide che, quando attraversato dalla corrente, genera un campo magnetico. La serratura, posta all'estremo di una molla, è normalmente chiusa (molla estesa e perno all'esterno della sede) e viene trattenuta dal campo magnetico generato dalla corrente, che bilancia la forza elastica della molla facendola ritrarre verso l'interno quando alimentata. Se invece non viene inviata corrente al solenoide, la molla respingerà la serratura all'esterno della carcassa metallica, impedendo l'apertura della porta.

Poiché i PIC18F4580 possono fornire una corrente massima in uscita dai pin di 200 mA, mentre la serratura per funzionare correttamente necessita di almeno 700 mA di corrente per generare un campo magnetico sufficiente, abbiamo utilizzato un transistor NPN S8050 con caratteristiche adeguate all'applicazione, collegato e utilizzato come "inseguitore di emettitore" (Fig.1.4.3-3):

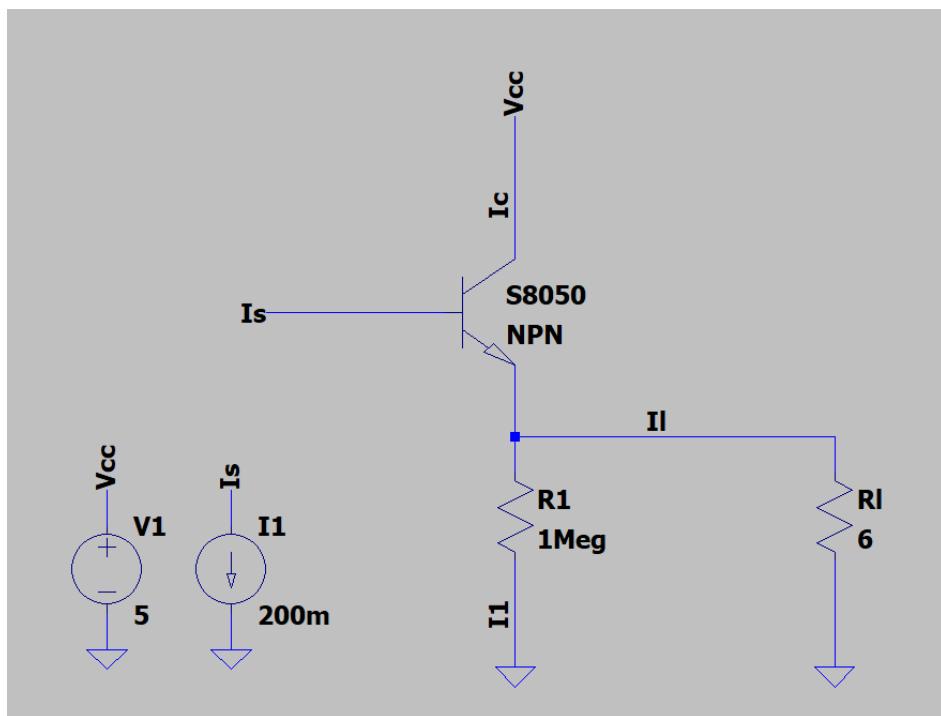


Fig.1.4.3-3 Schema elettrico del collegamento del transistor

Quando il PIC non invia corrente alla base del transistor ( $I_S=0$  A), questo funziona in modalità di interdizione, non permettendo passaggio di corrente dal collettore (connesso ad alimentazione) all'emettitore (connesso al carico, cioè la serratura); si avrà quindi  $I_C = 0$  A.

Quando invece il PIC invia corrente alla base ( $I_S \approx 200$  mA), il transistor entra in modalità di saturazione, funzionando come un interruttore chiuso e permettendo il passaggio della corrente dal collettore all'emettitore.

Dai dati messi a disposizione dal costruttore sappiamo che, quando la serratura è alimentata a 5 V la corrente di lavoro sarà pari a circa 0,83 A, mentre quando è alimentata a 6 V sarà di circa 1 A. Dalla legge di Ohm ricaviamo quindi una stima della resistenza opposta dal carico:

$$V = R * I$$

$$V_A = 5 \text{ V}$$

$$V_B = 6 \text{ V}$$

$$I_A = 0,83 \text{ A}$$

$$I_B = 1 \text{ A}$$

$$R_l \cong \frac{V_A}{I_A} = \frac{5 \text{ V}}{0,83 \text{ A}} \cong \frac{V_B}{I_B} = \frac{6 \text{ V}}{1 \text{ A}} = 6 \Omega$$

Sempre dal costruttore sappiamo inoltre che, con una corrente di collettore di 500 mA, la tensione di saturazione tra il collettore e l'emettitore del transistor è di 0.5 V, dunque il carico sarà soggetto ad una d.d.p. tra alimentazione e massa di circa 4.5 V.

Avendo collegato in parallelo al carico una resistenza da  $1 \text{ M}\Omega$  è lecito fare le seguenti approssimazioni:

$$I_1 \cong 0 \text{ A}$$

$$I_l \cong I_c$$

Quindi, la corrente di collettore è praticamente la stessa di quella inviata al carico:

$$I_c \cong \frac{V_l}{R_l} = \frac{4.5 \text{ V}}{6 \Omega} = 0,75 \text{ A.}$$

## 2. Tastierino (ad esclusione dello Slave della porta principale)

I tastierini utilizzati sono del classico tipo “a membrana”, ossia formati da due membrane separate da uno strato vuoto (come mostra la Fig. 1.2.3-4). Lo strato inferiore di membrana è formato da due zone conduttrive separate da una di isolante, per cui se il tasto non viene premuto non vi è passaggio di corrente tra le due zone. Se invece il tasto viene premuto, la membrana superiore entra a contatto con quella inferiore, permettendo il passaggio della corrente da una zona all'altra della membrana inferiore.

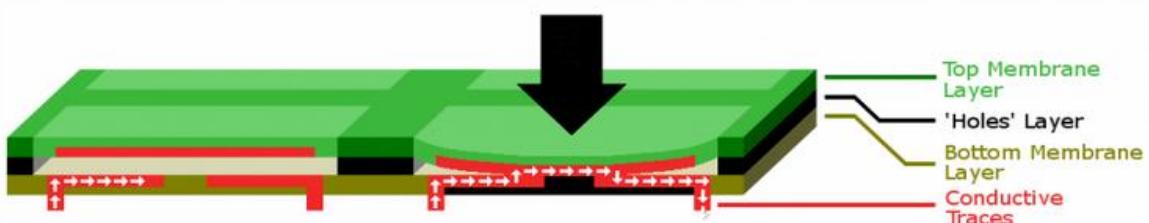


Fig. 1.4.3-4 Tastierino a membrana pt.1

Per meglio chiarire quanto detto, nella successiva figura (Fig. 1.4.3-5) è riportato il tastierino con uno schema approssimativo della piedinatura (come si può notare il tastierino non è alfanumerico, solo numerico):

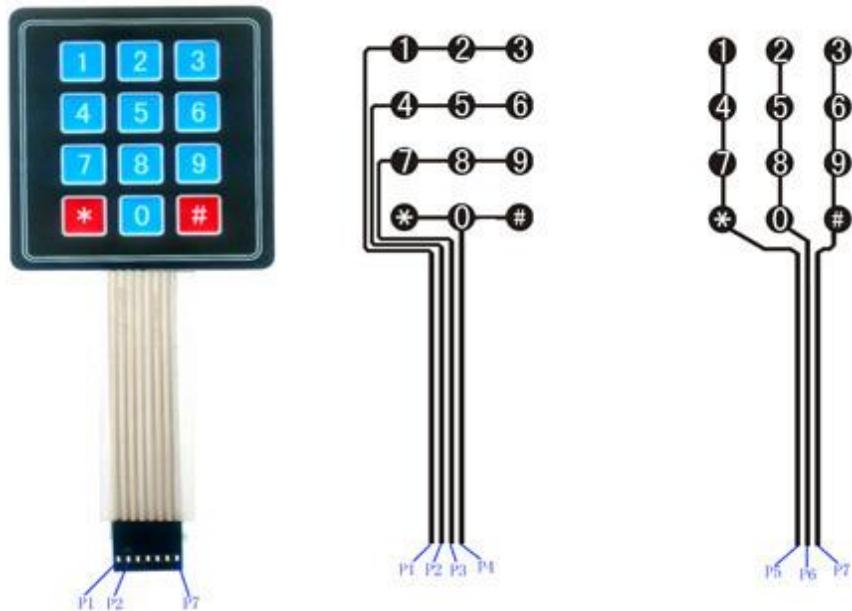


Fig. 2.4.3-5 Tastierino a membrana pt.2

### 3. Sensore magnetico

Il sensore magnetico, necessario per monitorare lo stato della porta associata, è composto da due magneti, che una volta a contatto annullano i rispettivi campi magnetici.

La parte di sensore fissata alla parete, è connessa all'alimentazione e ad un pin del PIC; l'altra è fissata alla porta, in modo che quando quest'ultima è chiusa i due magneti siano a contatto e si allontanino alla sua apertura, come mostrato nella figura seguente (Fig. 1.4.3-6):

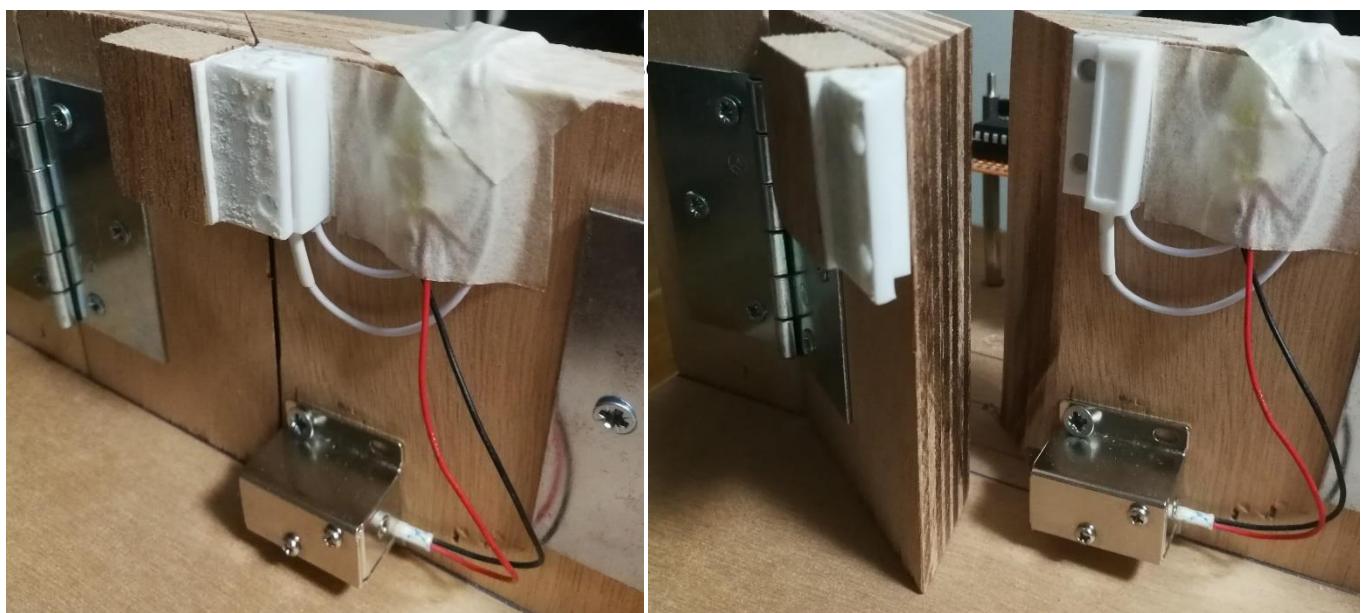


Fig. 1.4.3-6 Sensore magnetico a contatto (porta chiusa, a sinistra), e sensore non a contatto (porta aperta, a destra)

Si riporta di seguito uno schema elettrico delle connessioni dei dispositivi appena descritti al generico PIC Slave (ad esclusione di quello della porta principale che non dispone del tastierino):

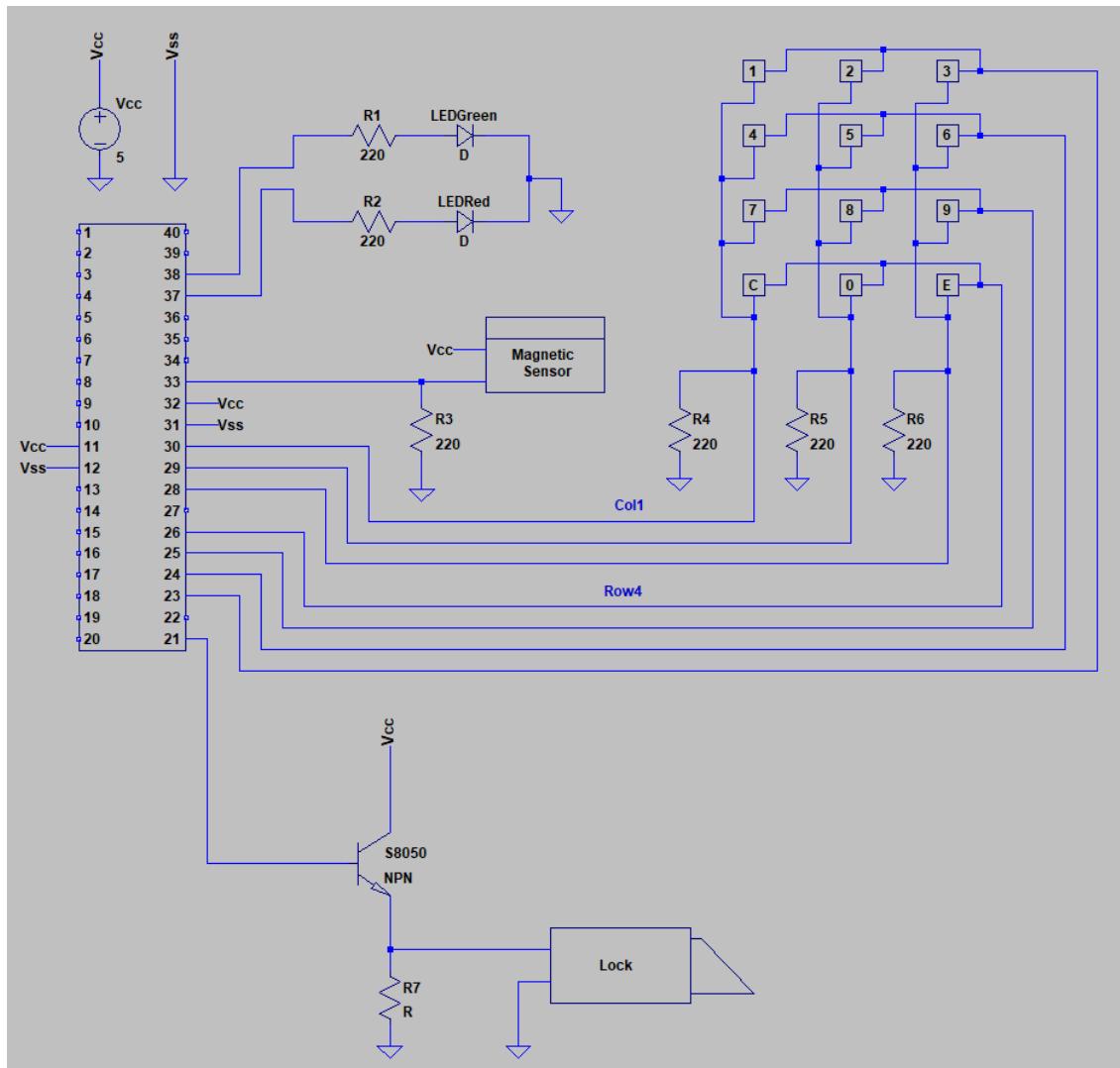


Fig. 1.4.3-7 Schema elettrico PIC Slave

#### 1.4.4 Altri componenti

Di non secondaria importanza per il corretto funzionamento del sistema sono alcuni altri componenti che verranno brevemente introdotti e descritti sommariamente:

- Per l'alimentazione complessiva del sistema è stato scelto un alimentatore volutamente sovradimensionato, il quale eroga una tensione DC costante di 5 V, e una corrente variabile in base alla richiesta, fino a raggiungere un massimo di 60 W di potenza (quindi eroga una corrente massima di 12 A). In allegato al materiale è presente il datasheet di tale dispositivo (Alimentatore RS-75-5). se ne riporta comunque un'immagine nel seguito:



Fig. 1.4.4-1 Alimentatore serie RS-75 variante 5

- Per l'implementazione corretta della connessione seriale (tramite protocollo RS-232) tra PIC Master e dispositivo master di sistema è stato necessario fare uso di opportuni moduli per l'adattamento dei livelli di tensione. Tali moduli (nel seguito a cui si farà riferimento come level-shifter) sono riportati nella successiva Fig.1.4.4-2.

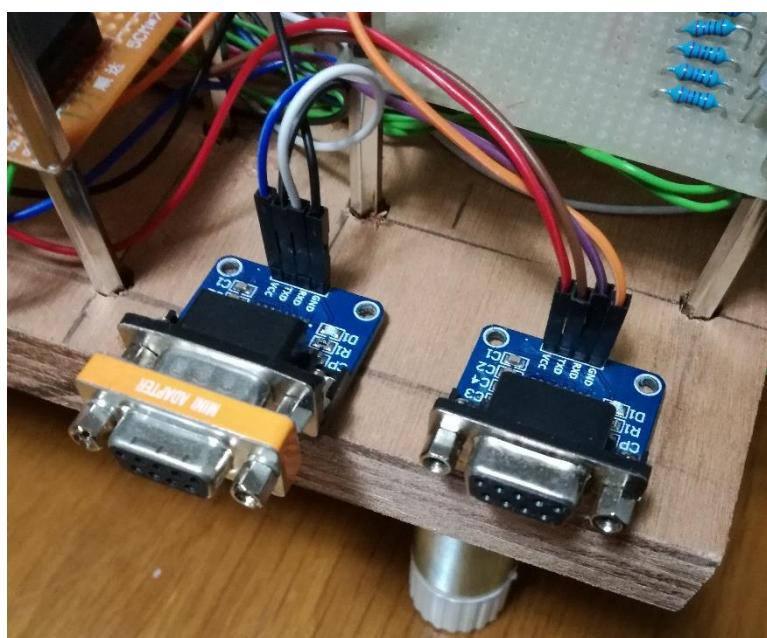
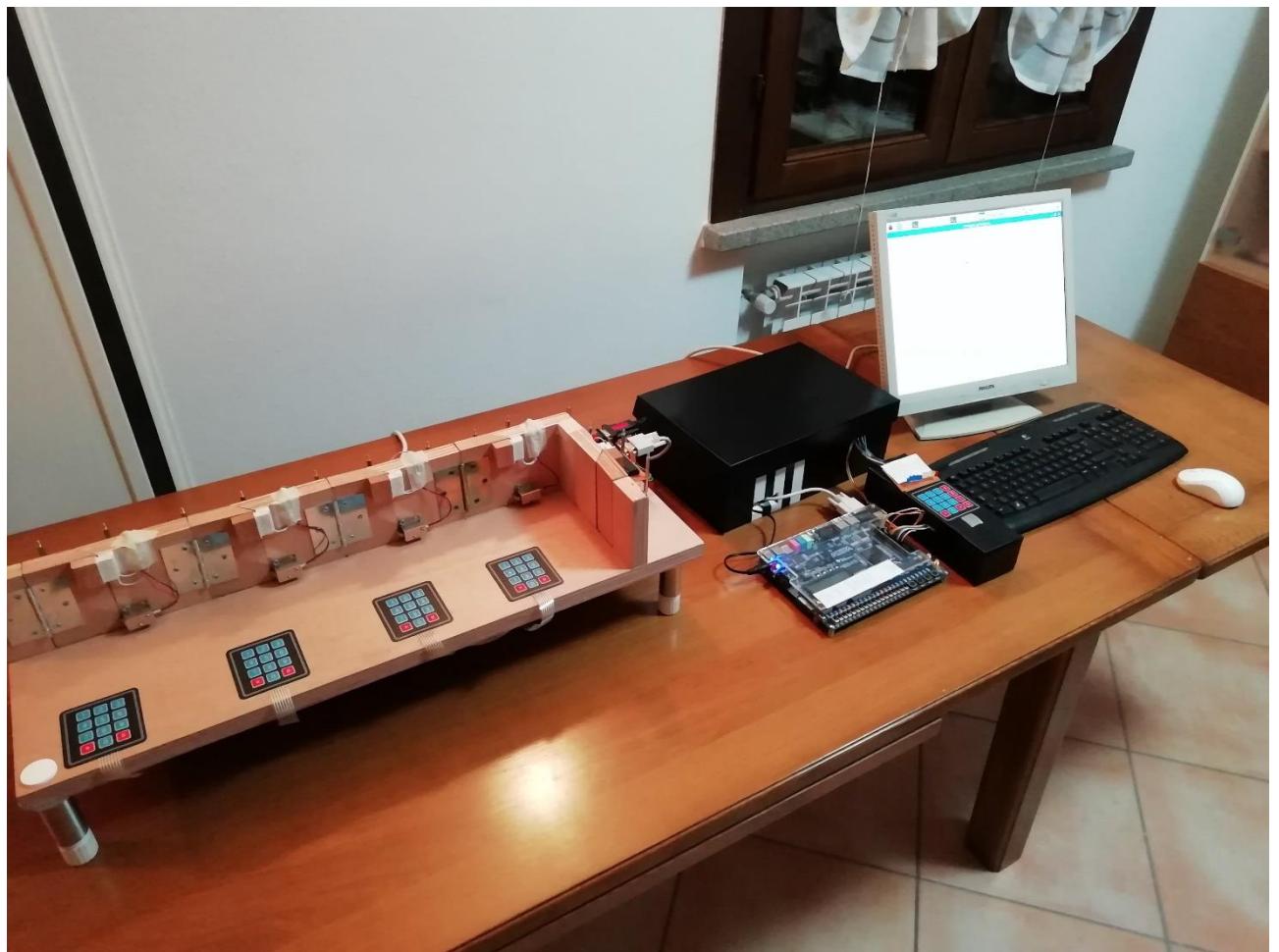


Fig. 1.4.4-2 Sulla destra il modulo level-shifter per la comunicazione con il Raspberry, sulla sinistra lo stesso modulo con in aggiunta un Null-Modem per la comunicazione con l'FPGA

Si noti che i moduli appaiono diversi, in realtà questa apparente diversità è dovuta a quanto segue:

- Il modulo sulla destra della figura serve per la comunicazione seriale (tramite cavo USB-seriale) tra PIC Master e Raspberry;
- Il modulo sulla sinistra serve per lo stesso scopo ma per collegare il PIC Master all'FPGA, in questo caso però il cavo seriale utilizzato non effettua internamente l'inversione dei terminali TX ed RX (il TX di un dispositivo deve essere collegato all'RX dell'altro). Pertanto, è stato necessario aggiungere al level-shifter un ulteriore adattatore Null-Modem (dispositivo con striscia arancione in figura), il quale ha proprio lo scopo di effettuare questa inversione dei terminali.

Nelle figure seguenti (1.4.4-3, 1.4.4-4 e 1.4.4-5) viene mostrato il prototipo così ottenuto:



*Fig. 1.4.4-3 Panoramica complessiva del prototipo*

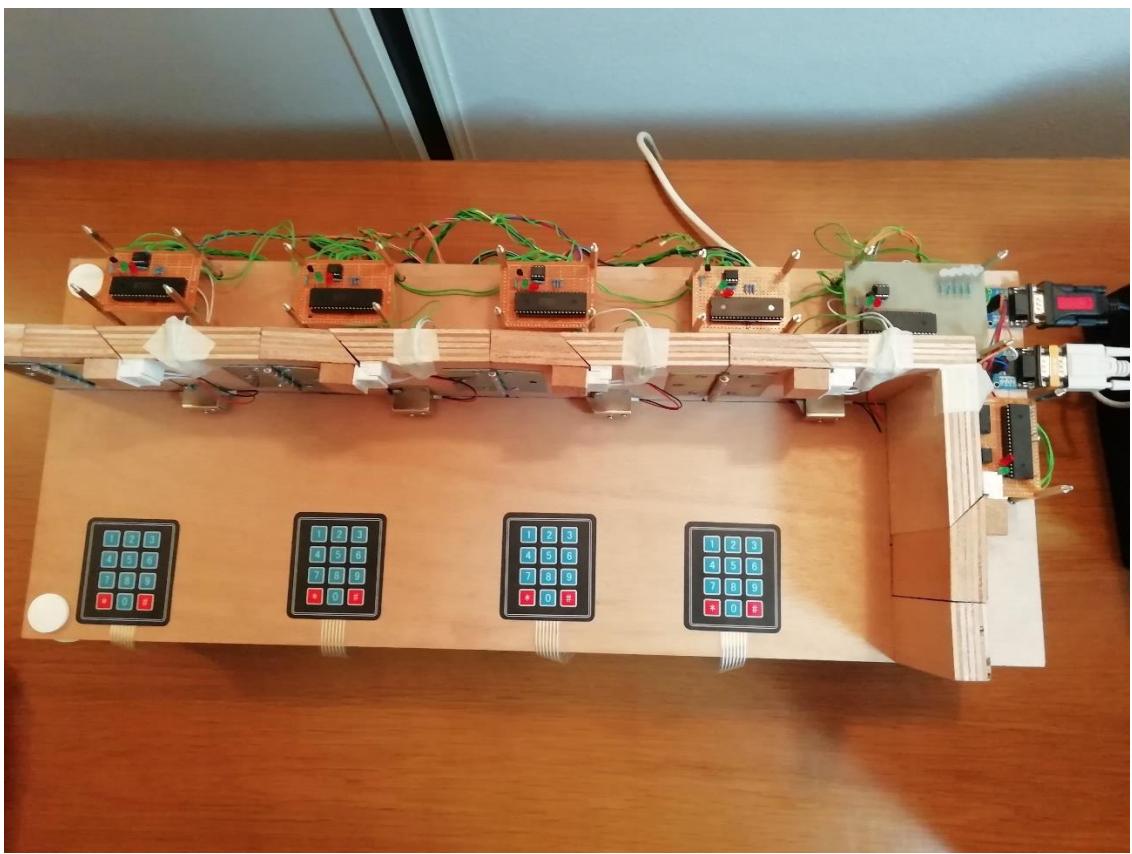


Fig. 1.4.3-4 Dettaglio porte. In evidenza, le millefori con PIC e transceiver; le porte, con sensori e serrature; tastierini numerici

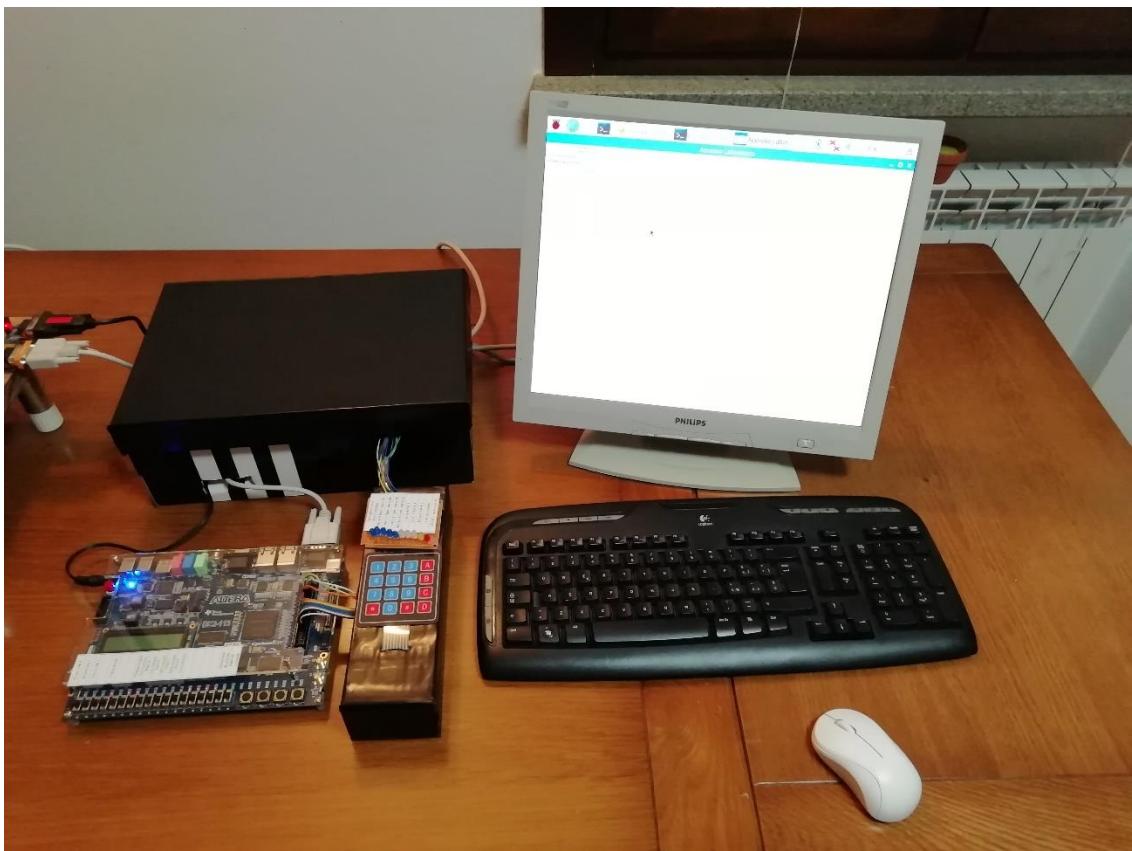


Fig. 1.4.4-5 Dettaglio master di sistema. In evidenza, la scheda FPGA e l'interfaccia utente del Raspberry

Nei prossimi capitoli verranno spiegate nel dettaglio tutte le parti del sistema e il loro funzionamento partendo dal master di sistema principale, ovvero il Raspberry (*capitolo 2*), proseguendo con la scheda FPGA, il master di sistema ausiliario (*capitolo 3*), e terminando con la descrizione dei PIC, facendo distinzione tra PIC Master e PIC Slave (*capitolo 4*).

Infine (*capitolo 5*), verrà svolto un approfondimento sugli eventuali sviluppi futuri del sistema in esame.

## 2 Raspberry

Il Raspberry è l'elemento a comando di tutto il sistema di gestione, in caso di guasto o malfunzionamento viene sostituito dalla scheda FPGA.

Esso gestisce tutto il database degli utenti, ne salva le informazioni personali: nome, cognome, ID e password e ne consente l'accesso ai laboratori. Gli ID e le password sono formati esclusivamente da cifre numeriche da zero a nove, l'ID è composto da sei cifre, la password da quattro, il nome e il cognome invece solo salvati in caratteri alfabetici normali.

Esistono due tipi di utenti, amministratori o utenti semplici. Gli utilizzatori del sistema dovranno identificarsi e in base al loro status avranno accesso a diverse funzioni del programma. Gli amministratori possono modificare il database, inserire nuovi utenti, nominare nuovi admin o declassarli e possono sempre visualizzare l'elenco completo delle persone che hanno accesso ai laboratori.

Tutti gli utenti hanno accesso ai laboratori. Dopo essersi identificati, i loro dati necessari all'accesso, cioè la porta a cui vogliono accedere e la password personale con cui dovranno identificarsi davanti alla porta, vengono inviati al PIC Master che a sua volta elabora queste informazioni e le invia ai successivi microcontrollori che agiscono di conseguenza per permettere alla persona di accedere al laboratorio desiderato.

Questo dispositivo ha anche la funzione di mostrare all'utente eventuali errori che si sono verificati nell'impianto, per esempio se qualche porta non è funzionante o se c'è già un altro utente in fase di accesso, in questo caso viene impedito a nuovi utenti di accedere. In caso di totale mancanza di segnale da parte del PIC Master, avviserà l'utente che non è possibile utilizzare il sistema. Viene indicato anche il caso in cui non venga rilevata la presenza del dispositivo ausiliario.

Per garantire l'utilizzo anche in caso di guasto del Raspberry, esso invia all'FPGA parte dei dati degli utenti, all'avvio del sistema e ogni volta che il database viene modificato. Vengono inviati solo gli ID e le password, in questo modo gli utilizzatori si potranno identificare allo stesso modo, ma non è possibile visualizzare il database o modificarlo.



Fig. 2-1 Raspberry

Il Raspberry è l'elemento che coordina e controlla tutto il sistema di accesso. Le sue funzioni sono:

- Inviare e ricevere un segnale di alive dall'FPGA
- Inviare e ricevere un segnale di alive dal PIC Master
- Ricevere, identificare e mostrare all'utente vari tipi di errori e messaggi
- Permettere all'utente di identificarsi e selezionare una porta da aprire
- Per gli utenti admin, fornire la possibilità di modificare il database
- Inviare i messaggi necessari all'apertura delle porte al PIC Master
- Inviare le informazioni degli utenti all'FPGA

Sono stati scritti due processi in python. Uno si occupa dell'invio e ricezione di alive e altri messaggi, l'altro della visualizzazione dell'interfaccia grafica che consente all'utente di utilizzare il sistema.

Entrambi utilizzano le porte seriali in maniera parallela, questo può comportare degli errori con conseguente interruzione del sistema se i due programmi si trovano a scrivere o a leggere contemporaneamente sulla stessa porta. Tale problema al momento non è stato risolto ma sicuramente in uno sviluppo futuro potrebbe esserlo facendo comunicare tra loro i due programmi in modo da non sovrapporsi, per esempio durante l'invio dei dati del database all'FPGA l'invio e la ricezione di alive e altri messaggi dovrebbe essere interrotto.

## 2.1 Invio e ricezione carattere di alive e altri messaggi

Lo scopo di questo processo è verificare che l'FPGA e il PIC siano collegati e rispondano all'alive. Ha anche la funzione di ricevere dal PIC Master messaggi di errore relativi al sistema e agire di conseguenza.

I codici che vengono inviati come alive e che vengono ricevuti come messaggi di errore sono i seguenti (Tabella 2.1-1), è riportato il loro equivalente in numeri decimali perché nell'invio vengono convertiti in codice ASCII.

Messaggio di errore o segnale ricevuto	Codice errore (hex)	Valore decimale	Conseguenza
Mancanza di alive dal PIC per oltre 12 secondi	FF	255	Visualizzazione di una schermata di allarme per 3 secondi.
Mancanza di alive dall'FPGA per oltre 4 secondi	CC	204	Accensione LED rosso fino alla ricezione di un nuovo segnale di alive.
Segnale di porta aperta	DD	221	Visualizzazione di una schermata che impedisce l'utilizzo del sistema per 3 secondi.
Segnale di malfunzionamento della porta 1	D1	209	Accensione LED blu 1 per 150 secondi
Segnale di malfunzionamento della porta 2	D2	210	Accensione LED blu 2 per 150 secondi
Segnale di malfunzionamento della porta 3	D3	211	Accensione LED blu 3 per 150 secondi
Segnale di malfunzionamento della porta 4	D4	212	Accensione LED blu 4 per 150 secondi
Segnale di malfunzionamento della porta principale	D5	213	Accensione LED blu 5 per 150 secondi
Segnale di errore comunicazione CAN tipo 1	E1	225	Accensione LED bianco 1 fino al reset manuale del sistema
Segnale di errore comunicazione CAN tipo 2	E2	226	Accensione LED bianco 2 fino al reset manuale del sistema
Segnale di errore comunicazione CAN tipo 3	E3	227	Accensione LED bianco 3 fino al reset manuale del sistema
Segnale di errore comunicazione seriale	EE	238	Accensione LED giallo per 150 secondi

Tabella 2.1-1 Messaggi di errore

I segnali di errori CAN sono degli errori che si possono verificare nella comunicazione tra i PIC, essi verranno descritti nelle sezioni 4.1.13 e 4.2.6.

Il segnale di errore seriale indica che il messaggio per l'apertura della porta non è stato inviato correttamente, se si verifica questo errore non è possibile accedere, verrà illustrato nel dettaglio nella sezione 4.1.10.

Per prima cosa, all'avvio del programma vengono inviati i due segnali di alive al PIC e all'FPGA e viene salvato nelle variabili *tempoPic* e *tempoFPGA* il tempo corrente.

Dopo di che inizia il ciclo infinito, ecco le azioni che esegue:

- Ogni 2 secondi invia i caratteri di alive sia all'FPGA che al PIC.
- Controlla se e cosa è stato ricevuto dal PIC, per ogni messaggio ricevuto scrive il suo significato su schermo, in un'implementazione futura, i dati potrebbero essere salvati su file e questo potrebbe essere usato come storico del funzionamento, per tenere traccia degli errori ma anche dei movimenti delle persone all'interno del sistema.

I messaggi ricevuti possono essere:

- segnale di alive FF: la variabile che controlla l'attività del PIC *tempoPic* viene aggiornata con il tempo attuale.
- segnale di porta aperta: viene attivata una schermata che impedisce l'utilizzo del programma di accesso, infatti nel caso ci fosse una porta aperta nessun'altra persona può entrare nello stesso momento, questa schermata rimane visibile per 3 secondi. Mentre questa schermata è aperta il programma rimane su questa istruzione, la lettura dei dati in ingresso non è in atto, perciò si è dovuto scegliere il tempo più corto possibile. Non è previsto un segnale di "porta chiusa" ma il PIC invia il messaggio di porta aperta ogni 6 secondi, perciò si è scelto che la schermata rimanesse visibile per soli 3 secondi, in modo da poter ricevere il messaggio successivo, nel caso non venga più ricevuto significa che la porta è stata richiusa.
- Segnale di porta fuori uso (da 1 a 5): viene acceso un LED corrispondente alla porta fuori servizio, il valore corrente del tempo viene salvato in una variabile apposita (es. *tmp\_1*) e una volta che saranno trascorsi 150 secondi da questo evento il LED verrà spento. Questo lasso di tempo è stato scelto perché non è stato previsto un segnale di "porta di nuovo in funzione" ma il segnale di porta non funzionante continua a essere inviato una volta ogni 2 minuti; quindi finché la porta sarà fuori uso ogni due minuti verrà ricevuto questo segnale, ma se non viene più ricevuto per 150 secondi allora significa che la porta è tornata in funzione.
- Segnali di errori CAN: anche se potrebbero non interrompere il funzionamento del sistema, essendo errori gravi vengono segnalati con LED che rimarranno accesi fino a che il sistema non verrà totalmente riavviato da un tecnico. Questi segnali sono usati per far capire al tecnico manutentore che tipo di errore si è verificato.
- Segnale di errore seriale: se viene ricevuto questo messaggio significa che non è possibile accedere per un errore di invio del messaggio, si è scelto di segnalarlo con LED che rimane acceso per 150 secondi per permettere all'utente di accorgersi dell'errore. Se questo errore si verifica spesso dovrà richiedere l'intervento di un tecnico per verificare che cosa lo crea.
- Controlla se viene ricevuto il segnale di alive, 55, dall'FPGA (si ricorda che quello inviato, invece, è CC). Ogni volta che lo riceve aggiorna la variabile relativa all'FPGA *tempoFpga*.
- Mostra una schermata se non è stato ricevuto il segnale di alive dal PIC per oltre 12 secondi, questa schermata viene mostrata per 3 secondi per lo stesso motivo illustrato precedentemente. Se il PIC non risponde continuerà a essere riaperta, fino a che questo non invierà correttamente il messaggio di alive.
- Accende un LED nel caso in cui l'FPGA non invii più l'alive per oltre 4 secondi. Tale LED viene spento non appena viene ricevuto di nuovo tale segnale.

L'FPGA è collegata serialmente grazie ai due pin presenti sul Raspberry di TX e RX. Il PIC Master è connesso alla porta USB con un cavo USB-seriale e la connessione seriale viene aperta tramite la porta USB, questo fa sì che qualora i cavi seriali venissero scollegati questo processo si arresti perché viene rilevato che la comunicazione si è interrotta. Uno sviluppo futuro sarebbe quello di gestire questo errore, in modo che il

programma non si interrompa e mostrare una schermata per informare l'utente di questo problema. Una volta riparato il malfunzionamento, il programma dovrebbe essere in grado di ripartire con il normale funzionamento.

Un'altra implementazione futura riguarda le schermate sopracitate di avviso in caso di impianto non funzionante, che al momento sono mostrate per pochi secondi perché interferiscono con la lettura dei messaggi seriali, il problema potrebbe essere gestito creando più processi comunicanti tra loro in modo che la ricezione dei messaggi non venga mai interrotta.

### Codice Python:

```
#importazione di moduli utili, in questo caso viene utilizzato il tempo, la comunicazione seriale, i message
#box e i pin general purpose per accendere LED.
#La variabile Act, definita sempre vera serve per attivare un ciclo while infinito.

import time
import serial
from struct import *
import tkMessageBox
from Tkinter import *
import RPi.GPIO as GPIO

Act = True
portFPGA = serial.Serial("/dev/ttyS0", baudrate = 9600, timeout = 3.0)
portPic = serial.Serial("/dev/ttyUSB0", baudrate = 9600, timeout = 3.0)

GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)

GPIO.setup(36, GPIO.OUT) #errori CAN
GPIO.setup(38, GPIO.OUT)
GPIO.setup(40, GPIO.OUT)

GPIO.setup(21, GPIO.OUT) #mancanza FPGA
GPIO.setup(23, GPIO.OUT) #errore seriale

GPIO.setup(29, GPIO.OUT) #porte fuori servizio
GPIO.setup(31, GPIO.OUT)
GPIO.setup(33, GPIO.OUT)
GPIO.setup(35, GPIO.OUT)
GPIO.setup(37, GPIO.OUT)

alivePic = 255      #FF    inviato al PIC per funzione di alive
aliveFPGA = 204     #CC    inviato all'FPGA per funzione di alive
open_door = 221     #DD    ricevuto dal PIC per segnalare che una porta è aperta
broken_1 = 209      #D1    ricevuto dal PIC per segnalare la porta 1 fuori servizio
broken_2 = 210      #D2    porta 2 fuori servizio
broken_3 = 211      #D3    porta 3 fuori servizio
broken_4 = 212      #D4    porta 4 fuori servizio
broken_5 = 213      #D5    porta principale fuori servizio
err_1 = 225         #E1    ricevuto dal PIC per segnalare un errore CAN di tipo 1
err_2 = 226         #E2    ricevuto dal PIC per segnalare un errore CAN di tipo 2
```

```

err_3 = 227           #E3      ricevuto dal PIC per segnalare un errore CAN di tipo 3
err_ser = 238         #EE      ricevuto dal PIC per segnalare un errore seriale

# Inizializzazione delle variabili che tengono traccia del tempo al quale avvengono gli eventi
tmp_1 = 0
tmp_2 = 0
tmp_3 = 0
tmp_4 = 0
tmp_5 = 0
tmp_ser = 0
tmp_invio= 0

#spegnimento di tutti i LED eventualmente accessi
for i in [21, 23, 29, 31, 33, 35, 37, 36, 38, 40]:
    GPIO.output(i, GPIO.LOW)

#invio iniziale alive
dato = pack('B', alivePic)      #La funzione pack converte il numero decimale nel suo corrispettivo ASCII
portPic.write(dato)

dato = pack('B', aliveFPGA)
portFPGA.write(dato)

# La funzione time.time() riporta il tempo assoluto trascorso dal 1 gennaio 1970, il suo valore viene salvato
# in questo momento per essere utilizzato come misura del tempo trascorso.

tmp_invio = time.time()
tempoPic = time.time()
tempoFPGA = time.time()

while Act:
    #invio alive ogni 2 secondi
    if time.time() > tmp_invio + 2:
        dato = pack('B', alivePic)
        portPic.write(dato)
        dato = pack('B', aliveFPGA)
        portFPGA.write(dato)
        tmp_invio = time.time()

    #mancanza di alive del PIC per oltre 12 secondi
    if time.time() > tempoPic + 12 :
        print('impianto non collegato')
        root = Tk()
        root.title("Accesso Laboratorio")
        root.geometry("1900x1000")
        messaggio = Label(root, text = "\n Attenzione! \n Impianto non funzionante", font=(40))
        messaggio.pack()
        root.after(3000, root.destroy)
        root.mainloop()

    #mancanza alive dall'FPGA per oltre 4 secondi

```

```

if time.time() > tempoFPGA + 4 :
    print('FPGA non collegata')
    GPIO.output(21, GPIO.HIGH)

#controllo ricezione dal PIC
rcv = (portPic.read(1))
if rcv != '':
    if rcv == pack('B', 255):
        tempoPic = time.time()
        print('alive dal PIC')
    if rcv == pack('B', open_door):
        print('porta aperta')
        root = Tk()
        root.title("Accesso Laboratorio")
        root.geometry("1900x1000")
        messaggio = Label(root, text = " \n Attenzione! \n Accesso momentaneamente non
disponibile", font=(40))
        messaggio.pack()
        root.after(4000, root.destroy)
        root.mainloop()
    if rcv == pack('B', broken_1):
        print('porta 1 fuori uso')
        tmp_1 = time.time()
        GPIO.output(29, GPIO.HIGH)
    if rcv == pack('B', broken_2):
        print('porta 2 fuori uso')
        tmp_2 = time.time()
        GPIO.output(31, GPIO.HIGH)
    if rcv == pack('B', broken_3):
        print('porta 3 fuori uso')
        tmp_3 = time.time()
        GPIO.output(33, GPIO.HIGH)
    if rcv == pack('B', broken_4):
        print('porta 4 fuori uso')
        tmp_4 = time.time()
        GPIO.output(35, GPIO.HIGH)
    if rcv == pack('B', broken_5):
        print('porta 5 fuori uso')
        tmp_5 = time.time()
        GPIO.output(37, GPIO.HIGH)
    if rcv == pack('B', err_ser):
        print('errore seriale')
        tmp_ser = time.time()
        GPIO.output(23, GPIO.HIGH)
    if rcv == pack('B', err_1):
        print('errore CAN 1')
        tmp_ser = time.time()
        GPIO.output(36, GPIO.HIGH)
    if rcv == pack('B', err_2):
        print('errore CAN 2')
        tmp_ser = time.time()
        GPIO.output(38, GPIO.HIGH)

```

```

if rcv == pack('B', err_3):
    print('errore CAN 3')
    tmp_ser = time.time()
    GPIO.output(40, GPIO.HIGH)

#controllo ricezione dall'FPGA
rcvFPGA = portFPGA.read(1)
if rcvFPGA == 'U':           # carattere ascii corrispondente al numero esadecimale 55
    tempoFPGA = time.time()
    GPIO.output(21, GPIO.LOW)

#controllo dei LED che si devono spegnere dopo un certo tempo
if tmp_1 != 0:
    if time.time() > tmp_1 + 150:
        GPIO.output(29, GPIO.LOW)
    tmp_1 = 0

if tmp_2 != 0:
    if time.time() > tmp_2 + 150:
        GPIO.output(31, GPIO.LOW)
    tmp_2 = 0

if tmp_3 != 0:
    if time.time() > tmp_3 + 150:
        GPIO.output(33, GPIO.LOW)
    tmp_3 = 0

if tmp_4 != 0:
    if time.time() > tmp_4 + 150:
        GPIO.output(35, GPIO.LOW)
    tmp_4 = 0

if tmp_5 != 0:
    if time.time() > tmp_5 + 150:
        GPIO.output(37, GPIO.LOW)
    tmp_5 = 0

if tmp_ser != 0:
    if time.time() > tmp_ser + 150:
        GPIO.output(23, GPIO.LOW)
    tmp_ser = 0

```

## 2.2 Interfaccia grafica

Il secondo processo, realizzato attraverso interfaccia grafica, permette all'utente di sfruttare il sistema in modo molto intuitivo.

Per prima cosa è richiesto all'utente di identificarsi, tranne nel caso particolare in cui non sia presente nessun amministratore in cui viene richiesto di inserire i dati di un admin.

Le funzioni che l'utente è in grado di usare variano in base al suo status: gli utenti semplici possono solamente richiedere l'accesso a un laboratorio, gli admin invece possono modificare il database degli utenti, possono aggiungere o eliminare utenti semplici e possono convertire un utente semplice in admin o viceversa.

Lo scopo principale del sistema di gestione è quello di poter selezionare una porta da aprire, dopo essersi identificati.

Nel diagramma di flusso (Fig. 2.2.-1) nella pagina seguente si possono vedere tutte le opzioni disponibili e tutti gli errori gestiti.

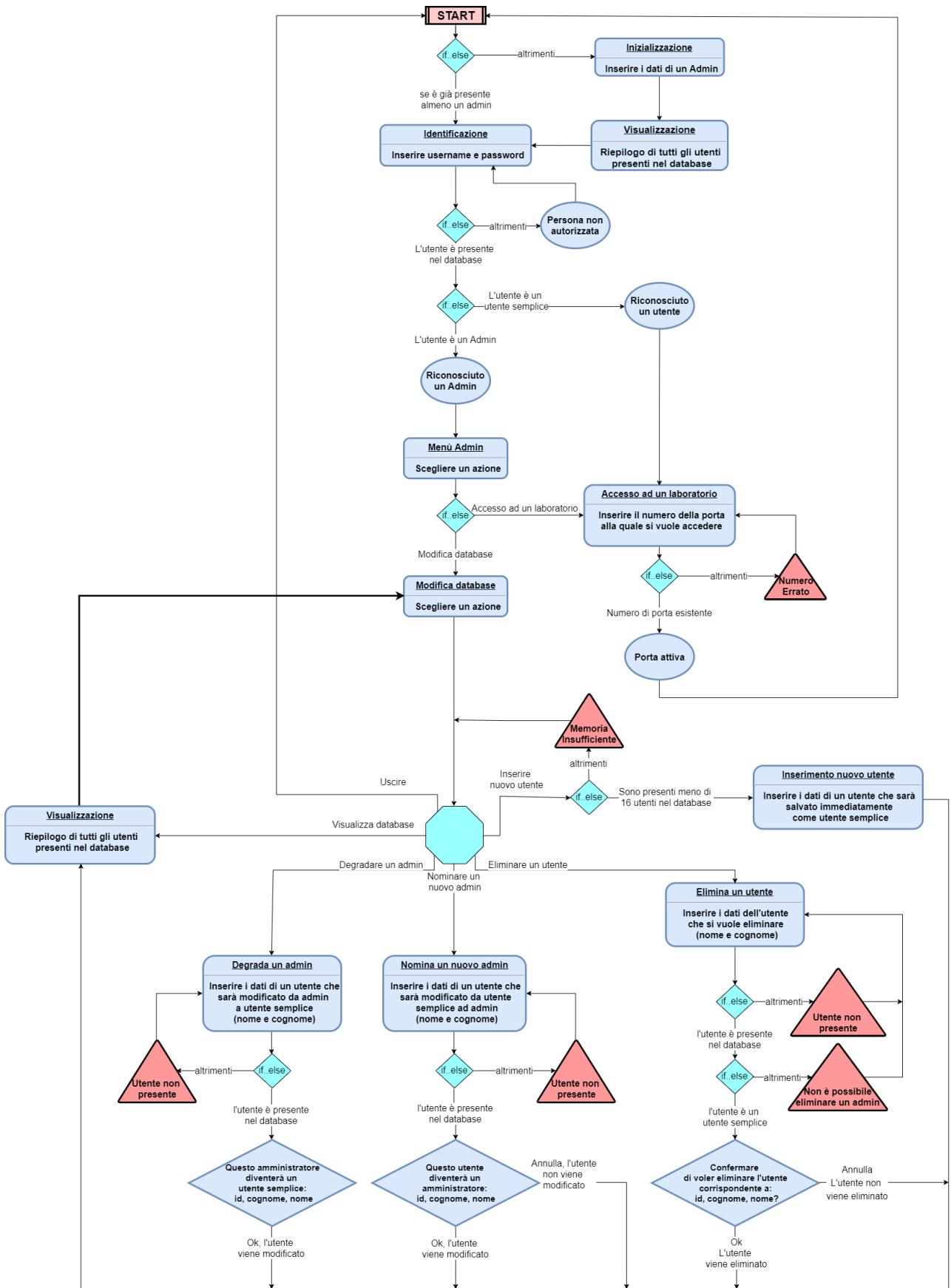


Fig. 2.2-1 Diagramma di flusso interfaccia grafica

Verranno illustrate ora le varie schermate realizzate.

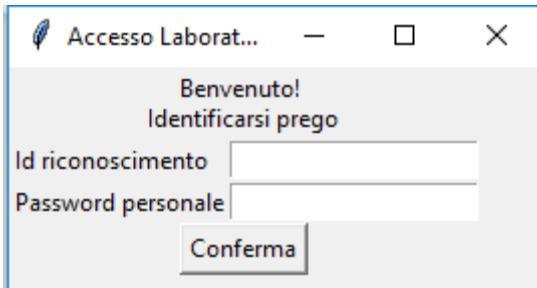


Fig. 2.2-2 Schermata Iniziale nel normale funzionamento

Questa è la schermata di normale utilizzo del sistema (Fig. 2.2-2), durante l'utilizzo però può verificarsi l'evento di non avere nessun admin nel database, in questo caso si viene indirizzati sulla schermata di inserimento iniziale di un admin (Fig. 2.2-3).

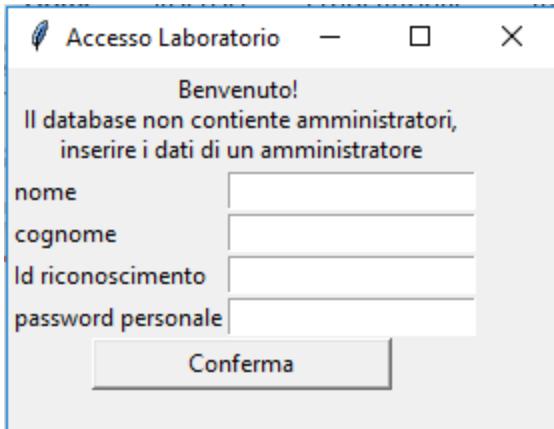


Fig. 2.2-3 Schermata iniziale in mancanza di admin

Nel caso di funzionamento normale è richiesto all'utente di inserire le proprie credenziali. Comparirà all'utente un messagebox che segnala l'esito del confronto (Fig. 2.2-4).

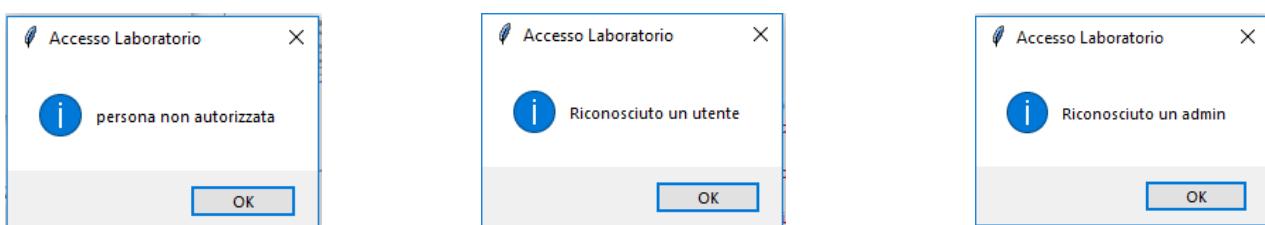


Fig. 2.2-4 Pop-up di informazione esito confronto ID e password

Nel caso di una persona non autorizzata, alla chiusura del messaggio si torna alla schermata di accesso. Se viene riconosciuto un utente semplice viene mostrata la schermata di accesso ai laboratori (Fig. 2.2-5).

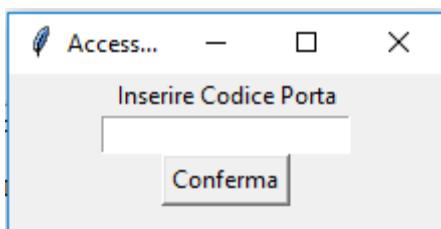


Fig. 2.2-5 Schermata di accesso ai laboratori

Una volta inserito il numero della porta, compare il messaggio di porta aperta se tale numero è corretto, altrimenti compare un messaggio di errore (Fig. 2.2-6) e viene richiesto di inserire nuovamente il numero della porta.

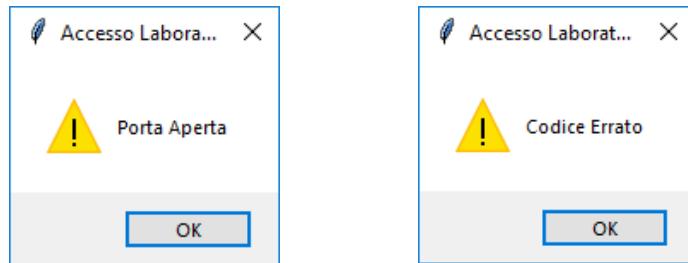


Fig. 2.2-6 Pop-Up Esito inserimento codice porta

Se invece viene riconosciuto un admin, egli ha la possibilità di scegliere le azioni da compiere (Fig. 2.2-7).

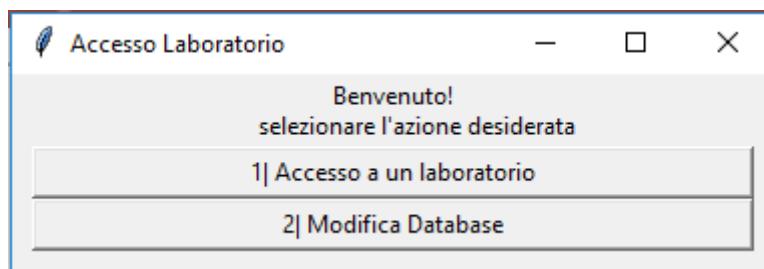


Fig. 2.2-7 Schermata di benvenuto per utenti admin

Se sceglie l'accesso a un laboratorio gli verrà presentata la stessa schermata di apertura porta vista precedentemente (Fig. 2.2-5).

Il menù per la modifica del database, invece, si presenta in questo modo (Fig. 2.2-8).

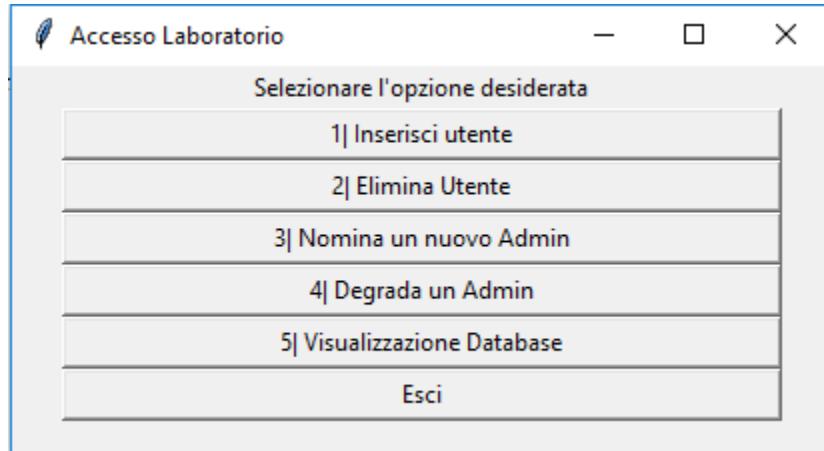


Fig. 2.2-8 Schermata Menù di modifica database

Cliccando sulla prima opzione appare un form (Fig. 2.2-9) in cui inserire i dati dell’utente da registrare.

The screenshot shows a standard Windows-style dialog box with a title bar 'Modifica database'. Inside, there's a header 'Inserisci i dati del nuovo utente'. Below it are four text input fields: 'nome', 'cognome', 'Id riconoscimento', and 'password personale'. At the bottom is a large 'Conferma' button.

Fig. 2.2-9 Schermata per l’inserimento dati di un nuovo utente

Se sono già presenti 16 utenti, compare la schermata di avviso di memoria insufficiente (Fig. 2.2-10) e non è possibile inserire nuovi utenti, la schermata di inserimento non verrà mostrata. Il motivo per cui 16 utenti è la capienza massima viene spiegato nel *capitolo 2.4*.

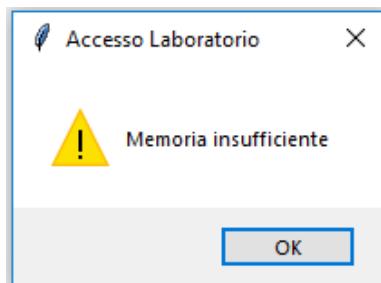


Fig. 2.2-10 Pop-up di errore, non è possibile inserire nuovi utenti

Al termine dell’inserimento per mostrare il risultato dell’operazione viene mostrata la schermata di visualizzazione del database (Fig. 2.2-11).

The screenshot shows a table titled 'Riepilogo Database'. It has columns 'Id', 'Cognome', 'Nome', and 'Tipo'. Two rows are shown: '123456 fantuzzi denise Admin' and '987654 bartolotta miriam Semplice'. At the bottom is a 'Conferma' button.

Riepilogo Database:			
Id	Cognome	Nome	Tipo
123456	fantuzzi	denise	Admin
987654	bartolotta	miriam	Semplice

Fig. 2.2-11 Schermata di visualizzazione database

Per eliminare un utente vengono richiesti il nome e il cognome dell’utente da eliminare (Fig. 2.2-12). Non è possibile eliminare un admin, in quel caso si avrà un messaggio di errore, come per il caso in cui l’utente non sia presente (Fig. 2.2-13), altrimenti verrà richiesta una seconda conferma dopo la quale l’utente sarà eliminato (Fig. 2.2-14). Per completezza viene mostrato di nuovo l’intero database con la schermata di visualizzazione vista in precedenza.

The screenshot shows a form with two text input fields: 'Cognome' and 'Nome'. At the bottom is a 'Conferma' button.

Fig. 2.2-12 Schermata per l’inserimento dati di un utente da modificare

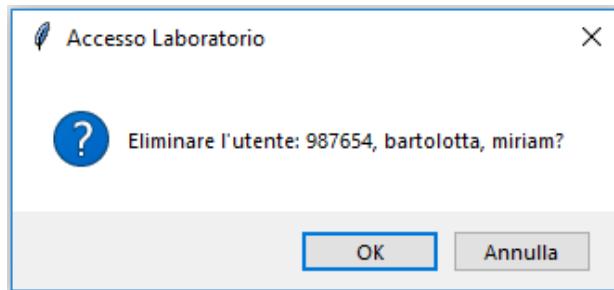


Fig. 2.2-14 Richiesta conferma azione

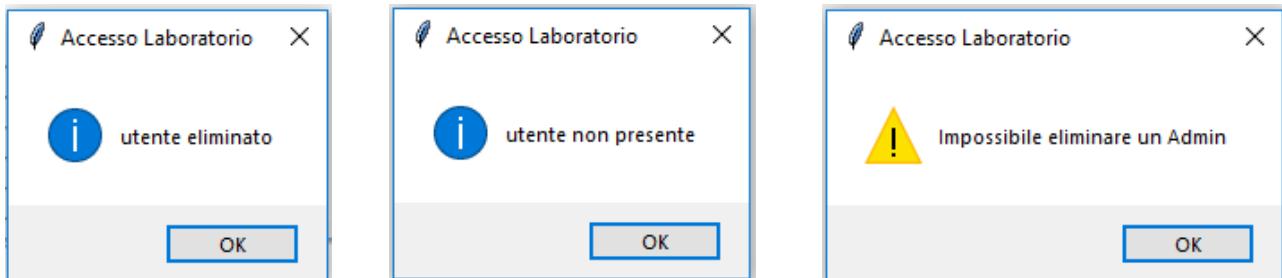


Fig. 2.2-13 Pop-up Esito eliminazione utente

Per le opzioni di nominare un nuovo admin o declassarne uno, vengono richiesti il nome e cognome dell’utente da modificare (con una schermata del tutto simile a Fig. 2.2-12); se tale utente non è presente, verrà mostrato un messaggio di errore (Fig. 2.2-14b), altrimenti verrà richiesta una seconda conferma (Fig. 2.2-15a e 2.2-16a) e dopo sarà effettuata la modifica. Ancora una volta viene mostrato il riepilogo del database (Fig. 2.2-15b e 2.2-16b).

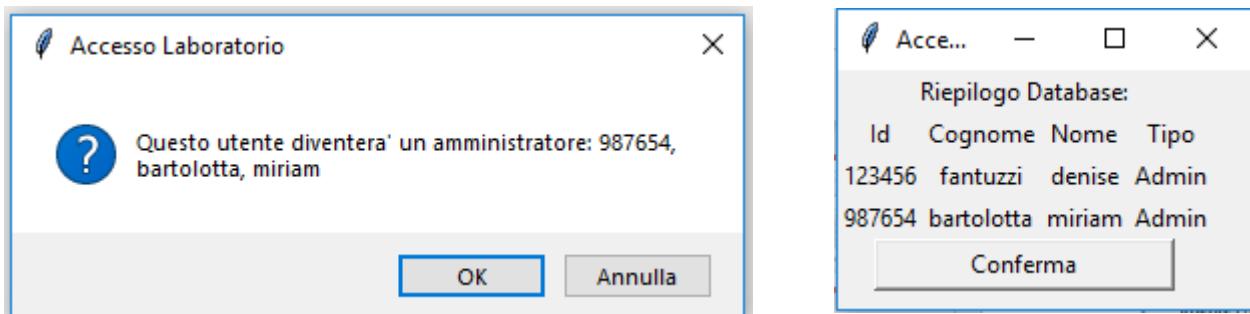


Fig. 2.2-16 Schermate relative all'upgrade di un utente

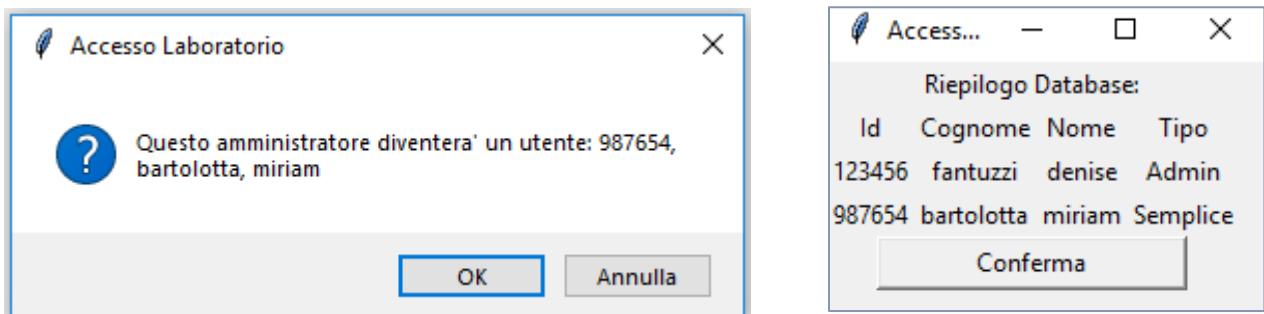


Fig. 2.2-16 Schermate relative al downgrade di un utente

L’ultima opzione del menù consente di vedere il database attuale senza apportare nessuna modifica. Cliccando sul tasto “Esci” si torna alla schermata di benvenuto.

Il database è stato realizzato usando il modulo *sqlite3* di Python. Questo modulo integrativo permette di creare database a matrice, in cui ogni riga è associata ad un oggetto. Ogni oggetto può a sua volta avere diversi attributi, che vengono elencati nelle diverse colonne.

Il database viene salvato in una posizione definita dal programmatore o casuale, in ogni caso è identificato da un nome che permette al dispositivo utilizzato di trovarne la posizione quando necessario.

Nel nostro caso il database non ha una posizione specificata ed è composto da cinque colonne:

1. ID
2. Cognome
3. Nome
4. Tipo di utente
5. Password

La colonna ID serve ad identificare l'oggetto e quindi la riga del database, le altre colonne servono ad inserire gli attributi, che sono appunto: nome e cognome dell'utente, tipo di utente (user o admin) e password personale.

Le due funzioni più importanti per il sistema sono quelle che avvengono quando si raggiunge la schermata di “porta aperta” e quando, dopo aver modificato il database, si clicca su “Esci”. Nel primo caso viene inviato al PIC Master un messaggio con le informazioni per l'apertura delle porte, nel secondo viene inviato l'intero database all'FPGA codificato in modo speciale.

Ora verranno illustrate queste due funzioni, in coda verrà descritto in modo più dettagliato il codice dell'interfaccia grafica.

## 2.3 Invio codice di apertura porta al PIC Master

La funzione per inviare numero di porta e password al PIC Master, prende in ingresso il dato composto da cinque cifre: numero porta + password, e segue questo protocollo per l'invio (cifre in esadecimale): vengono inviati due pacchetti di sei byte ciascuno. I primi due byte di entrambi sono AA e 55, per il primo pacchetto, il terzo è 0B, gli ultimi tre sono la cifra della porta e le prime due cifre della password; per il secondo pacchetto il terzo è 0C, seguono poi due byte per le due cifre restanti della password e l'ultimo byte della trasmissione è 0D. In Tabella 2.3-1 si riassume quanto detto.

Primo pacchetto:	Secondo pacchetto:
AA	AA
55	55
0B	0C
0000 + numero porta (4 bit)	0000 + terza cifra ID
0000 + prima cifra ID	0000 + quarta cifra ID
0000 + seconda cifra ID	0D

Tabella 2.3-1 Messaggio di apertura porta

### Codice Python:

(questa parte è all'interno del codice scritto nel *paragrafo 2.5*)

```
# funzione per l'invio info porta al PIC
def InviaPorta(dataIn):
    codice = str(dataIn)
    parola = 170 #'AA'
    dato = pack('B', parola) # funzione pack è utilizzata per convertire il numero da decimale a binario
    InviaPic(dato) # funzione per l'invio verso il PIC
    parola = 85 #'55'
    dato = pack('B', parola)
    InviaPic(dato)
    parola = 11 #'0B'
    dato = pack('B', parola)
    InviaPic(dato)
    for i in [0,1,2]:
        cifra = int(codice[i])
        cifra = pack('B', cifra)
        InviaPic(cifra)

    parola = 170 #'AA'
    dato = pack('B', parola)
    InviaPic(dato)
    parola = 85 #'55'
    dato = pack('B', parola)
    InviaPic(dato)
    parola = 12 #'0C'
    dato = pack('B', parola)
    InviaPic(dato)
    for i in [3,4]:
        cifra = int(codice[i])
        cifra = pack('B', cifra)
        InviaPic(cifra)
    parola = 13 #'0D"
```

```
dato = pack('B', parola)
InviaPic(dato)

# La funzione che effettua la comunicazione seriale con il PIC
def InviaPic(dataIn):
    global portPic
    portPic.write(dataIn)
```

## 2.4 Invio database all'FPGA

La funzione per inviare il database completo all'FPGA viene chiamata all'inizio dell'esecuzione del database e ogni volta che il database viene modificato.

La memoria a disposizione sull'FPGA è composta da 256 indirizzi, ognuno contenente un byte di dati.

Ogni utente è memorizzato in un blocco formato da 16 byte di memoria (utilizzandone però cinque) in modo che le prime quattro cifre significative dell'indirizzo definiscano univocamente la posizione di ogni utente, cioè tutti gli indirizzi che iniziano con la stessa cifra fanno riferimento ad uno stesso utente. Le quattro cifre meno significative rappresentano invece la posizione in memoria dei singoli byte in cui sono contenute le informazioni dell'utente, i primi cinque byte di ogni blocco sono utilizzati, i restanti sono lasciati vuoti. In questo modo gran parte della memoria rimane inutilizzata ed è possibile inserire solo 16 utenti, tale scelta è volta al solo scopo di semplificare il progetto.

I cinque byte di dati relativi ad un utente sono: sei cifre (da 0 a 9) per il nome utente e quattro cifre (sempre da 0 a 9) per la password personale. Le cifre vengono convertite in BCD perciò ognuna occupa 4 bit. In un byte vengono salvate due cifre, per un totale di cinque byte.

Esempio di dato memorizzato:

Indirizzo: E

ID: 268457

Password: 1035

Si può vedere come viene salvato nella memoria RAM questo utente in Tabella 2.4-1

Indirizzo (Hex)	Dato (Hex)
00	//
...	
E0	26
E1	84
E2	57
E3	10
E4	35
E5	//
...	
EF	//
...	
FF	//

Tabella 2.4-1 Esempio di utilizzo della memoria RAM della scheda FPGA

Questa conversione da numeri interi a BCD viene effettuata dalla funzione “ConvToHex”.

Essa riceve in ingresso, nel nostro caso, un numero da 0 a 99 (ma il metodo funziona per qualsiasi numero) e lo converte nel formato appena descritto.

Esempio: le prime due cifre dell'ID di un utente sono il numero 23, che in binario sarebbe “10111”. Esso sarà convertito in “00100011” cioè 23 in esadecimale e 35 in decimale, che viene salvato al posto del precedente 23 decimale.

La funzione “InviaData” per prima cosa salva gli ID e le password di tutti gli utenti presenti nel database e li unisce per formare il “codice utente” cioè le dieci cifre che lo identificano.

Dopo di che inizia l'invio del database, un utente alla volta (tutti i numeri elencati di seguito sono in esadecimale).

Non c'è un carattere di inizio invio database, ogni utente viene inviato singolarmente.

Il primo byte inviato è sempre A, seguito dalla cifra dell'indirizzo in cui verrà salvato l'utente; “A0”, “A1”...“AF”.

Come detto precedentemente gli utenti vengono salvati solo a partire dagli indirizzi "00", "10", "20"..."E0", "F0". Perciò la cifra inviata indica proprio in quale di questi 16 slot inserire l'utente.

I cinque byte inviati subito dopo sono la conversione delle cifre identificative dell'utente.

L'ultimo byte è BB, che segnala la fine dell'invio dei dati dell'utente.

Questa sequenza viene ripetuta 16 volte, se gli utenti effettivamente presenti sono meno, i restanti indirizzi vengono sovrascritti con zeri.

I numeri inseriti sono in decimale, attraverso la funzione pack vengono convertiti in binario.

In Tabella 2.4-2 si può vedere il messaggio trasmesso da Raspberry a FPGA per trasferire l'utente dell'esempio precedente.

Invio utente	
AE (esadecimale)	Carattere di inizio (A) e indirizzo di riferimento (E)
26 (= 0010 0110 cifre convertite in BCD)	Sei cifre per l'ID dell'utente
84 (= 1000 0100)	
57 (= 0101 0111)	
10 (= 0001 0000)	Quattro cifre per la password dell'utente
35 (= 0011 0101)	
BB (esadecimale)	Carattere di fine trasmissione utente

Tabella 2.4-2 Messaggio di trasmissione dati di un utente

### Codice Python:

(questa parte è all'interno del codice scritto nel paragrafo 2.5)

```
# funzione per la conversione di un numero intero in BCD
def convToHex(intero):
    intero = int(intero)
    ris = intero
    resto = intero
    resti = []
    numero = 0
    while resto > 0 :
        resto = resto % 10
        resto = resto // 10
        resti.append(resto)
    for i in range(len(resti)):
        numero = numero + resti[i] * (16 ** i)
    return numero

# funzione per l'invio dell'intero database all'FPGA
def InviaData():
    ids = []
    psws = []
    codici = []
    for row in c.execute('SELECT ID FROM Utenti'): #seleziono tutti gli ID
        row = str(row)
        ids.append(row[1:7])
    for row in c.execute('SELECT password FROM Utenti'): #seleziono tutte le password
        row = str(row)
        psws.append(row[1:5])
    for i in range(len(ids)): #formo i "codici utente"
        codice = ids[i] + psws[i]
        codici.append(codice)
```

```

for k in range(16):
    parola = 160 + k           #A0, A1, A2...
    dato = pack('B', parola)
    InviaFPGA(dato)

    if k < len(codici):
        for j in [0,2,4,6,8]:
            coppia = codici[k][j:j+2]
            dato = convToHex(coppia)
            dato = pack('B', dato)
            InviaFPGA(dato)

    else:
        for j in [0,2,4,6,8]:
            dato = 0
            dato = pack('B', dato)
            InviaFPGA(dato)

    parola = 187                #BB
    dato = pack('B', parola)
    InviaFPGA(dato)

#La funzione che effettua la comunicazione seriale con l'FPGA, un byte alla volta
def InviaFPGA(dataln):
    global portFPGA
    portFPGA.write(dataln)

```

## 2.5 Approfondimento codice dell’interfaccia grafica e del database

In questo capitolo verrà illustrato il codice dell’interfaccia grafica, che è stata già descritta funzionalmente in precedenza. Le funzioni per l’invio delle informazioni riguardanti l’apertura della porta e l’invio del database all’FPGA si trovano in questo programma ma non verranno descritte di nuovo.

Come prima cosa, nell’intestazione del codice si trovano gli import dei moduli usati per creare l’interfaccia (*Tkinter* e *tkMessageBox*), il database (*sqlite3*), la comunicazione seriale (*serial*) e la conversione dei dati (*struct*) e le variabili globali: la porta selezionata, la password inserita, il numero di porte del sistema, la risoluzione dello schermo, le porte seriali per FPGA e PIC Master. L’uso di queste variabili verrà definito in seguito.

```
from Tkinter import *
import tkMessageBox as messagebox
import sqlite3
import serial
import time
from struct import *
PortaSelezionata = None
password = 0
NMaxPorte = 4
definizione = "1900x1000"
portPic = serial.Serial("/dev/ttyUSB0", baudrate = 9600, timeout = 3.0)
portFPGA = serial.Serial("/dev/ttyS0", baudrate = 9600, timeout = 3.0)
```

In seguito, viene definito il database Utenti che ha le seguenti caratteristiche: ID (numero matricola), cognome, nome, tipo (Semplice o Admin), password (scelta dall’utente).

```
database = sqlite3.connect('data.db')
c = database.cursor() #creo un database in una certa posizione e collogo
                      #un cursore (istanza della classe sqlite3)
                      #per muovermi al suo interno ed effettuare
                      #le operazioni
c.execute("""CREATE TABLE IF NOT EXISTS Utenti
            (ID INTEGER,
             cognome TEXT,
             nome TEXT,
             tipo TEXT,
             password INTEGER)""")
```

L’ordine di definizione degli attributi è lo stesso di quello di inserimento di un nuovo utente.

Le uniche funzioni che vengono attivate all’interno di questo processo sono prima l’invio del database all’FPGA e poi la funzione di Inizializzazione che richiama a catena tutte le altre funzioni che rendono possibile visualizzare le schermate dell’interfaccia.

```
InviaData()
Inizializzazione()
```

Di seguito sono illustrate tutte le funzioni implementate.

**Funzione per il lancio dell'interfaccia:** controlla se è presente un admin, se non c'è fa partire l'inizializzazione, altrimenti avvia la schermata principale del database. Questa è la prima funzione che viene attivata all'avvio del programma e successivamente per tornare alla schermata di avvio.

```
def Inizializzazione():
    c.execute('SELECT * FROM Utenti WHERE tipo = ?', ("Admin",))
    data = c.fetchone()
    if data is None:
        root = Tk()
        finestra0 = Init(root)
        root.mainloop()
    else:
        root = Tk()
        finestra = Start(root)
        root.mainloop()
```

**Funzione per visualizzare il database:** Come tutte le funzioni che attivano una schermata, crea un'istanza della classe di riferimento che in questo caso si tratta della classe Visual per visualizzare il database. È una schermata che simula una tabella e mostra ID, nome e cognome degli utenti, non mostra le password per ovvie ragioni di sicurezza.

```
def VisualizzazioneDatabase():
    root = Tk()
    tabella = Visual(root)
```

La classe è dichiarata in modo indipendente dalla funzione che ne crea un'istanza.

**Definizione classe Visual:** nella definizione di questa classe si può vedere che viene fissata la grandezza della finestra pari a quella definita come variabile globale, viene settato il titolo della finestra “Accesso Laboratorio”, una label posta in alto con il titolo della schermata “Riepilogo database” e 4 label poste sulla stessa riga (“ID”, “Cognome”, “Nome” e “Tipo”) che formano le intestazioni delle colonne di questa tabella che schematizza il database. Le righe della tabella sono formate con i dati degli utenti inseriti, mostrati su varie label. Per chiudere la schermata c'è un pulsante posto al di sotto della tabella. I vari elementi della schermata sono differenziati per la loro lettera iniziale (W per window, B per button, L per label, E per entry) seguita dal dato che contengono o mostrano.

```
class Visual:
    def __init__(self, WVisual):
        self.WVisual = WVisual
        global definizione
        WVisual.geometry(definizione)
        WVisual.title("Accesso Laboratorio")
        self.LTitolo = Label(WVisual, text = "Riepilogo Database:")
        self.LTitolo.grid(row=0, columnspan = 4)
        self.LNome = Label(WVisual, text = "ID")
        self.LNome.grid(row = 1, column = 0)
        self.LCognome = Label(WVisual, text = "Cognome")
        self.LCognome.grid(row = 1, column = 1)
        self.LNome = Label(WVisual, text = "Nome")
        self.LNome.grid(row = 1, column = 2)
        self.LTipo = Label(WVisual, text = "Tipo")
        self.LTipo.grid(row = 1, column = 3)
```

```

data = []
for row in c.execute('SELECT * FROM Utenti'):
    data.append(list(row))
Nrow = len(data)
for i in range(Nrow):
    for j in range(4):
        self.LData0 = Label(WVisual, text = data[i][j])
        self.LData0.grid(row = i+2, column = j)

self.BOK = Button(WVisual, text = 'Conferma', width = 20, command = self.WVisual.destroy)
self.BOK.grid(row=i + 3, columnspan = 4)

```

**Definizione classe Init:** la classe per la finestra di inizializzazione in cui vengono richiesti tutti i dati di un admin che sarà il primo utente registrato nel database. In questa finestra viene richiesto all'utente di inserire in alcuni campi entry, segnalati da apposite label il proprio nome, cognome, ID e password. La classe ha anche una funzione “InserisciAdminZero” che salva i dati inseriti nel database. Per mostrare l'esito positivo dell'inserimento viene subito visualizzato il database attraverso la funzione “VisualizzazioneDatabase” illustrata precedentemente, contemporaneamente la finestra di inserimento viene chiusa e viene aperta la schermata iniziale del sistema di gestione degli accessi.

```

class Init:
    def __init__(self, WInit):
        self.WInit = WInit
        global definizione
        WInit.geometry(definizione)
        WInit.title("Accesso Laboratorio")
        self.LInit = Label(WInit, text = "Benvenuto! \n Il database non contiene amministratori, \n inserire i dati di un amministratore")
        self.LInit.grid(row=0, columnspan = 2)
        self.LNome = Label(WInit, text = "nome")
        self.LNome.grid(row = 1, column = 0, sticky = 'W')

        self.ENome = Entry(WInit)
        self.ENome.grid(row = 1, column = 1)
        self.LCognome = Label(WInit, text = "cognome")
        self.LCognome.grid(row = 2, column = 0, sticky = 'W')

        self.ECognome = Entry(WInit)
        self.ECognome.grid(row = 2, column = 1)
        self.LId = Label(WInit, text = "ID riconoscimento")
        self.LId.grid(row = 3, column = 0, sticky = 'W')

        self.EId = Entry(WInit)
        self.EId.grid(row = 3, column = 1)
        self.LPsw = Label(WInit, text = "password personale")
        self.LPsw.grid(row = 4, column = 0, sticky = 'W')

        self.EPsw = Entry(WInit)
        self.EPsw.grid(row = 4, column = 1)
        self.BOK = Button(WInit, text = 'Conferma', width = 20, command = self.InserisciAdminZero)
        self.BOK.grid(row=6, columnspan = 2)

```

```

def InserisciAdminZero(self):
    cogn = str(self.ECognome.get())
    nome = str(self.ENome.get())
    psw = int(self.EPsw.get())
    ID = int(self.EId.get())
    c.execute("""INSERT INTO Utenti VALUES (?,?,?,?,?) """, (ID, cogn, nome, "Admin", psw))
    database.commit()
    VisualizzazioneDatabase()
    self.WInit.destroy()
    root = Tk()
    finestra = Start(root)

```

**Definizione classe Start:** La classe per la finestra di start, questa finestra è il funzionamento normale dell'interfaccia. Vengono richiesti ID e password per accedere alle funzioni. La pressione del pulsante “Conferma” richiama la funzione “Riconoscimento” anch’essa definita all’interno di questa classe.

```

class Start:
    def __init__(self, WId):
        self.WId = WId
        global definizione
        WId.geometry(definizione)
        WId.title("Accesso Laboratorio")
        self.LBen = Label(WId, text = "Benvenuto! \n Identificarsi prego")
        self.LBen.grid(row=0, columnspan = 2)
        self.LId = Label(WId, text = "ID riconoscimento")
        self.LId.grid(row = 1, column = 0, sticky = 'W')

        self.EId = Entry(WId)
        self.EId.grid(row = 1, column = 1)
        self.LPsw = Label(WId, text = "Password personale")
        self.LPsw.grid(row = 2, column = 0, sticky = 'W')

        self.EPsw = Entry(WId)
        self.EPsw.grid(row = 2, column = 1)
        self.BOk = Button(WId, text = 'Conferma', command = self.Riconoscimento)
        self.BOk.grid(row = 3, columnspan = 2)
        self.psw = None
        self.ID = None

```

**Funzione Riconoscimento:** per riconoscere se le credenziali inserite sono di un admin, di un utente o di un estraneo. Avendo come input l’ID e la password dell’utente effettua una ricerca nel database; se non c’è nessuna corrispondenza viene mostrato un message box per informare l’utente che le credenziali inserite non sono presenti nel database. Se invece l’utente è presente la sua password viene salvata come variabile globale, perché nel caso egli volesse aprire un laboratorio questo dato dovrebbe essere utilizzato da altre funzioni per essere inviato agli altri componenti del sistema. Successivamente viene effettuato il controllo sul tipo di utente; se è un utente semplice, viene chiamata la funzione che mostrerà la schermata per l’apertura porta, se è un admin verrà aperto il menù generale di modifica del database e apertura porte. In entrambi i casi viene comunicato tramite message box l’esito del confronto.

```

def Riconoscimento(self):
    self.psw = str(self.EPsw.get())

```

```

        self.ID = int(self.Eld.get())
        c.execute("SELECT * FROM Utenti WHERE ID=:ID AND password=:psw", {"ID": self.ID, "psw": self.psw})
        data = c.fetchone()

        if data is None:
            stringa = 'persona non autorizzata'
            messagebox.showinfo("Accesso Laboratorio", stringa)
            self.WId.destroy()
            Inizializzazione()

        else:
            global password
            password = int(data[4])
            if str(data[3]) == 'Semplice':
                stringa = 'Riconosciuto un utente'
                messagebox.showinfo("Accesso Laboratorio", stringa)
                self.WId.destroy()
                root = Tk()
                finestra2 = AprirePorta(root)
            else: #admin
                stringa = 'Riconosciuto un admin'
                messagebox.showinfo("Accesso Laboratorio", stringa)
                self.WId.destroy()
                root= Tk()
                finestra2 = BenvenutoAdmin(root)

```

**Definizione classe BenvenutoAdmin:** la classe per la finestra riservata agli admin, consente di scegliere se aprire una porta o modificare il database, in base al tasto premuto viene richiamata la funzione apposita “SelPorta” per aprire la schermata di apertura porta e “ModData” per aprire la schermata relativa alla modifica del database.

```

class BenvenutoAdmin:

    def __init__(self, WStart):
        self.WStart = WStart
        global definizione
        WStart.geometry(definizione)
        WStart.title("Accesso Laboratorio")
        self.LWelcome = Label(WStart, text = """Benvenuto!
selezionare l'azione desiderata""")
        self.LWelcome.pack()
        self.BSelPorta = Button(WStart, text = '1| Accesso a un laboratorio', width = 50, command=
self.SelPorta) #cliccando questo pulsante viene chiamata la funzione SelPorta
        self.BSelPorta.pack()
        self.BModData = Button(WStart, text = '2| Modifica Database', width =50, command =
self.ModData)
        self.BModData.pack()

    def SelPorta(self):
        self.WStart.destroy()
        root = Tk()

```

```

finestra3 = AprirePorta(root)

def ModData(self):
    self.WStart.destroy()
    root = Tk()
    finestra3 = Menu(root)

```

**Definizione classe AprirePorta:** la classe per la finestra dell'apertura porta definisce il tipo di schermata utilizzata per l'apertura della porta. È presente una casella in cui inserire il numero della porta e un tasto per dare la conferma. Se il numero della porta è superiore al numero di porte esistenti viene mostrato un messaggio di errore. Se invece è corretto, tale dato viene unito con la variabile *password* precedentemente inserita e inviato al PIC Master tramite un'altra funzione “InviaPorta” che è stata illustrata nel paragrafo precedente. Al termine dell'invio la schermata viene chiusa e viene richiamata la schermata iniziale “Start” con la funzione “Inizializzazione” già incontrata in precedenza.

*class AprirePorta:*

```

def __init__(self, WSelPorta):
    self.WSelPorta= WSelPorta
    global definizione
    WSelPorta.geometry(definizione)
    WSelPorta.title("Accesso Laboratorio")
    self.LSelPorta = Label(WSelPorta, text = 'Inserire Codice Porta')
    self.LSelPorta.pack()
    self.ESelPorta = Entry(WSelPorta)
    self.ESelPorta.pack()
    self.BOkPorta = Button(WSelPorta, text = 'Conferma', command = self.NumPorta)
    self.BOkPorta.pack()

def NumPorta(self)
    if str(self.ESelPorta.get()) == "":
        messagebox.showwarning("Accesso Laboratorio", "Inserire un numero")
    elif int(self.ESelPorta.get()) > NMaxPorte
        messagebox.showwarning("Accesso Laboratorio", "Codice Errato")
        self.ESelPorta.delete(0,5)
    else:
        PortaSelezionata = str(self.ESelPorta.get())
        global password
        psw = str(password)
        codice = int(PortaSelezionata + psw)
        codice = str(codice)
        InviaPorta(codice)
        self.WSelPorta.destroy()
        Inizializzazione()

```

**Definizione classe Menu:** la classe per la finestra del menu della modifica database. Qui vengono mostrate le varie opzioni tra cui scegliere per la modifica del database, cliccando sul pulsante corrispondente si richiama la funzione che apre la schermata apposita. Saranno illustrate ora, per ogni opzione, la classe per la schermata e le opzioni di ciascuna. Il tasto “Esci” chiude la schermata e torna alla pagina iniziale ma richiama anche la funzione “InviaData” illustrata precedentemente, che invia il database all’FPGA.

```

class Menu:
    def __init__(self, WMenu):
        self.WMenu = WMenu
        global definizione
        WMenu.geometry(definizione)
        WMenu.title("Accesso Laboratorio")
        self.LMenu = Label(WMenu, text = "scegliere l'opzione desiderata")
        self.LMenu.pack()
        self.BInsUtente = Button(WMenu, text = '1| Inserisci utente', width = 50, command=
self.InsUtente)
        self.BInsUtente.pack()
        self.BDelUtente = Button(WMenu, text = '2| Elimina Utente', width =50, command =
self.DelUtente)
        self.BDelUtente.pack()
        self.BUpgrade = Button(WMenu, text = '3| Nomina un nuovo Admin', width =50, command
= self.Upgrade)
        self.BUpgrade.pack()
        self.BDowngrade = Button(WMenu, text = '4| Degrada un Admin', width =50, command =
self.Downgrade)
        self.BDowngrade.pack()
        self.BDowngrade = Button(WMenu, text = '5| Visualizzazione Database', width =50,
command = VisualizzazioneDatabase)
        self.BDowngrade.pack()
        self.BDowngrade = Button(WMenu, text = 'Esci', width =50, command = self.Chiudi)
        self.BDowngrade.pack()

    def InsUtente(self):
        c.execute('SELECT * FROM Utenti')
        if len(c.fetchall()) < 16:
            root = Tk()
            finestra4 = InserireUtente(root)
        else:
            messagebox.showwarning("Accesso Laboratorio", "Memoria insufficiente")

    def DelUtente(self):
        root = Tk()
        finestra4 = CancellareUtente(root)

    def Upgrade(self):
        root = Tk()
        finestra4 = RendiAdmin(root)

    def Downgrade(self):
        root = Tk()
        finestra4 = RendiUtente(root)

```

```

def Chiudi(self):
    InviaData()
    self.WMenu.destroy()
    Inizializzazione()

```

**Definizione classe InserireUtente:** la classe per la finestra di inserimento di un nuovo utente, vengono richieste tutte le info dell'utente attraverso text box. Alla pressione del tasto “Ok” la funzione “InserisciNuovoUtente” salverà i dati nel database. Al termine del salvataggio viene visualizzato il database completo.

`class InserireUtente:`

```

def __init__(self, WIns):
    self.WIns = WIns
    global definizione
    WIns.geometry(definizione)
    WIns.title("Modifica database")
    self.LTitolo = Label(WIns, text = "Inserisci i dati del nuovo utente")
    self.LTitolo.grid(row=0, columnspan = 2)
    self.LNome = Label(WIns, text = "nome")
    self.LNome.grid(row = 1, column = 0, sticky = 'W')

    self.ENome = Entry(WIns)
    self.ENome.grid(row = 1, column = 1)
    self.LCognome = Label(WIns, text = "cognome")
    self.LCognome.grid(row = 2, column = 0, sticky = 'W')

    self.ECognome = Entry(WIns)
    self.ECognome.grid(row = 2, column = 1)
    self.LId = Label(WIns, text = "ID riconoscimento")
    self.LId.grid(row = 3, column = 0, sticky = 'W')

    self.EId = Entry(WIns)
    self.EId.grid(row = 3, column = 1)
    self.LPsw = Label(WIns, text = "password personale")
    self.LPsw.grid(row = 4, column = 0, sticky = 'W')

    self.EPsw = Entry(WIns)
    self.EPsw.grid(row = 4, column = 1)
    self.BOK = Button(WIns, text = 'Conferma', width = 20, command =
self.InserisciNuovoUtente)
    self.BOK.grid(row = 5, columnspan = 2)

```

`def InserisciNuovoUtente(self):`

```

cogn = str(self.ECognome.get())
nome = str(self.ENome.get())
psw = int(self.EPsw.get())
ID = int(self.EId.get())
c.execute("""INSERT INTO Utenti VALUES (?,?,?,?,?) """, (ID, cogn, nome, "Semplice", psw))
database.commit()
VisualizzazioneDatabase()

```

```
self.WIns.destroy()
```

**Definizione classe Cancellareutente:** la classe per la finestra della cancellazione di un utente in cui vengono richiesti nome e cognome dell'utente da eliminare. La funzione “CancellaUtente” attivata dal pulsante “Ok” controlla se nel database è presente l’utente. In caso negativo mostra un errore; in caso positivo, ma se l’utente è un admin, mostra un altro tipo di errore, perché non è possibile eliminare un admin; altrimenti vengono mostrati di nuovo i dati dell’utente per chiedere la conferma della scelta, e se viene premuto il tasto “Ok” l’utente viene definitivamente cancellato dal database. Per completezza viene mostrato di nuovo il database completo.

```
class CancellareUtente:  
    def __init__(self, WCanc):  
        self.WCanc = WCanc  
        global definizione  
        WCanc.geometry(definizione)  
        WCanc.title("Modifica database")  
        self.LCogn = Label(WCanc, text = "Cognome")  
        self.LCogn.grid(row = 0, column = 0, sticky = 'W')  
  
        self.ECogn = Entry(WCanc)  
        self.ECogn.grid(row = 0, column = 1)  
        self.LNome = Label(WCanc, text = "Nome")  
        self.LNome.grid(row = 1, column = 0, sticky = 'W')  
  
        self.ENome = Entry(WCanc)  
        self.ENome.grid(row = 1, column = 1)  
  
        self.BOK = Button(WCanc, text = 'Conferma', width =20, command = self.CancellaUtente)  
        self.BOK.grid(row=2, columnspan= 2)  
  
    def CancellaUtente(self):  
        cogn = str(self.ECogn.get())  
        nome = str(self.ENome.get())  
  
        c.execute("SELECT * FROM Utenti WHERE cognome=:sur AND nome=:name", {"sur": cogn,  
"name": nome})  
        data= []  
        data = c.fetchone()  
        if data is None:  
            messagebox.showinfo("Accesso Laboratorio", 'utente non presente')  
            self.ECogn.delete(0,20)  
            self.ENome.delete(0,20)  
        else:  
            if str(data[3]) == "Admin":  
                messagebox.showwarning("Accesso Laboratorio", "Impossibile eliminare un  
Admin")  
                self.WCanc.destroy()  
            else:  
                risposta = messagebox.askokcancel("Accesso Laboratorio", "Eliminare  
l'utente?" + str(data[:3]))  
                if risposta:
```

```

        c.execute("DELETE FROM Utenti WHERE cognome=:sur AND
        nome=:name", {"sur": cogn, "name": nome})
                database.commit()
                messagebox.showinfo("AccessoLaboratorio", "utente eliminato")
                VisualizzazioneDatabase()
                self.WCanc.destroy()

```

**Definizione classe RendiAdmin:** La classe per rendere admin un utente, in cui sono richiesti nome e cognome dell'utente. Come nel caso dell'eliminazione compare un messaggio di errore se l'utente non è presente, se invece è presente viene richiesta una seconda conferma e al termine viene mostrato il database.

```

class RendiAdmin:
    def __init__(self, WAdm):
        self.WAdm = WAdm
        global definizione
        WAdm.geometry(definizione)
        WAdm.title("Modifica database")
        self.LCogn = Label(WAdm, text = "Cognome")
        self.LCogn.grid(row = 0, column = 0, sticky = 'W')

        self.ECogn = Entry(WAdm)
        self.ECogn.grid(row = 0, column = 1)
        self.LNome = Label(WAdm, text = "Nome")
        self.LNome.grid(row = 1, column = 0, sticky = 'W')

        self.ENome = Entry(WAdm)
        self.ENome.grid(row = 1, column = 1)
        self.BOK = Button(WAdm, text = 'Conferma', width = 20, command =
self.RendiAmministratore)
        self.BOK.grid(row=2, columnspan = 2)

    def RendiAmministratore(self):
        cogn = str(self.ECogn.get())
        nome = str(self.ENome.get())

        c.execute("SELECT * FROM Utenti WHERE cognome=:sur AND nome=:name", {"sur": cogn,
"nome": nome})
        data= []
        data =c.fetchone()
        if data is None:
            messagebox.showinfo("Accesso Laboratorio", 'utente non presente')
            self.ECogn.delete(0,20)
            self.ENome.delete(0,20)
        else:
            risposta = messagebox.askokcancel("Accesso Laboratorio", "Questo utente
diventerà un amministratore: " + str(data[:3]))
            if risposta:
                c.execute("UPDATE Utenti SET tipo = 'Admin' where cognome=:sur AND
nome=:name", {"sur": cogn, "name": nome})
                database.commit()
                self.WAdm.destroy()
                VisualizzazioneDatabase()

```

**Definizione classe RendiUtente:** la classe per la finestra del degradamento di un admin, sono richiesti nome e cognome dell'admin, uguale alla schermata precedente.

```

class RendiUtente:
    def __init__(self, WUt):
        self.WUt = WUt
        global definizione
        WUt.geometry(definizione)
        WUt.title("Modifica database")
        self.LCogn = Label(WUt, text = "Cognome")
        self.LCogn.grid(row = 0, column = 0, sticky = 'W')

        self.ECogn = Entry(WUt)
        self.ECogn.grid(row = 0, column = 1)
        self.LNome = Label(WUt, text = "Nome")
        self.LNome.grid(row = 1, column = 0, sticky = 'W')

        self.ENome = Entry(WUt)
        self.ENome.grid(row = 1, column = 1, sticky = 'W')
        self.BOK = Button(WUt, text = 'Conferma', width = 20, command = self.RendiUt)
        self.BOK.grid(row = 2, columnspan= 2)

    def RendiUt(self):
        cogn = str(self.ECogn.get())
        nome = str(self.ENome.get())

        c.execute("SELECT * FROM Utenti WHERE cognome=:sur AND nome=:name", {"sur": cogn,
        "name": nome})
        data = []
        data = c.fetchone()
        if data is None:
            messagebox.showinfo("Accesso Laboratorio", 'utente non presente')
            self.ECogn.delete(0,20)
            self.ENome.delete(0,20)
        else:
            risposta = messagebox.askokcancel("Accesso Laboratorio", "Questo amministratore
diventera' un utente: " + str(data[:3]))

            if risposta:
                c.execute("UPDATE Utenti SET tipo = 'semplice' where cognome=:sur AND
                nome=:name", {"sur": cogn, "name": nome})
                database.commit()

        self.WUt.destroy()
        VisualizzazioneDatabase()

```

### 3 FPGA

In questa sezione verrà descritto l'uso dell'FPGA nell'ambito del progetto realizzato partendo dalle funzioni che svolge ed entrando sempre più nel dettaglio del funzionamento dei singoli blocchi.

L'FPGA, in questo progetto, è utilizzata come soluzione di continuità in mancanza del Raspberry. È possibile suddividere il suo comportamento in due casi possibili; la distinzione tra questi due casi è fornita dal controllo perpetuo del segnale di alive del Raspberry.

- **Raspberry ON**

All'accensione del sistema, l'FPGA controlla immediatamente il segnale di alive dal Raspberry. In presenza di questo, la gestione degli utenti sarà garantita dal Raspberry e l'FPGA rimane in attesa. Quando viene aggiornato il database sul Raspberry, per esempio inserendo un nuovo utente, o alla prima accensione dello stesso, viene effettuato l'invio dei dati all'FPGA. Questi dati vengono salvati nella memoria RAM interna all'FPGA. Essendo le RAM memorie volatili a lettura e scrittura, quando la board dell'FPGA viene spenta l'intero database viene perso. L'utente deve assicurarsi che il Raspberry invii nuovamente il database per consentire il corretto funzionamento, per fare ciò è sufficiente accendere il Raspberry e collegarlo correttamente almeno per qualche istante.

- **Raspberry OFF**

Se per 4 secondi non viene ricevuto il segnale di alive, l'FPGA entra in Safe Mode. Questa modalità permette di utilizzare il sistema di gestione accessi anche in assenza del Raspberry. Passando dal funzionamento normale alla Safe mode, il monitor è sostituito dallo schermo LCD 16x2 presente sulla board, la tastiera e il mouse per l'inserimento dei dati sono sostituiti da un modulo matrix membrane keypad 4x4. In questa modalità le funzionalità sono ridotte, non è possibile modificare o visualizzare il database. L'FPGA si occuperà anche della comunicazione con il PIC Master tramite protocollo seriale RS-232 e anche della segnalazione di errori nel sistema tramite l'accensione di specifici LED sulla board. Se all'accensione del sistema, il Raspberry risulta non collegato, non sarà possibile utilizzare l'FPGA perché non è stato effettuato il caricamento del database.

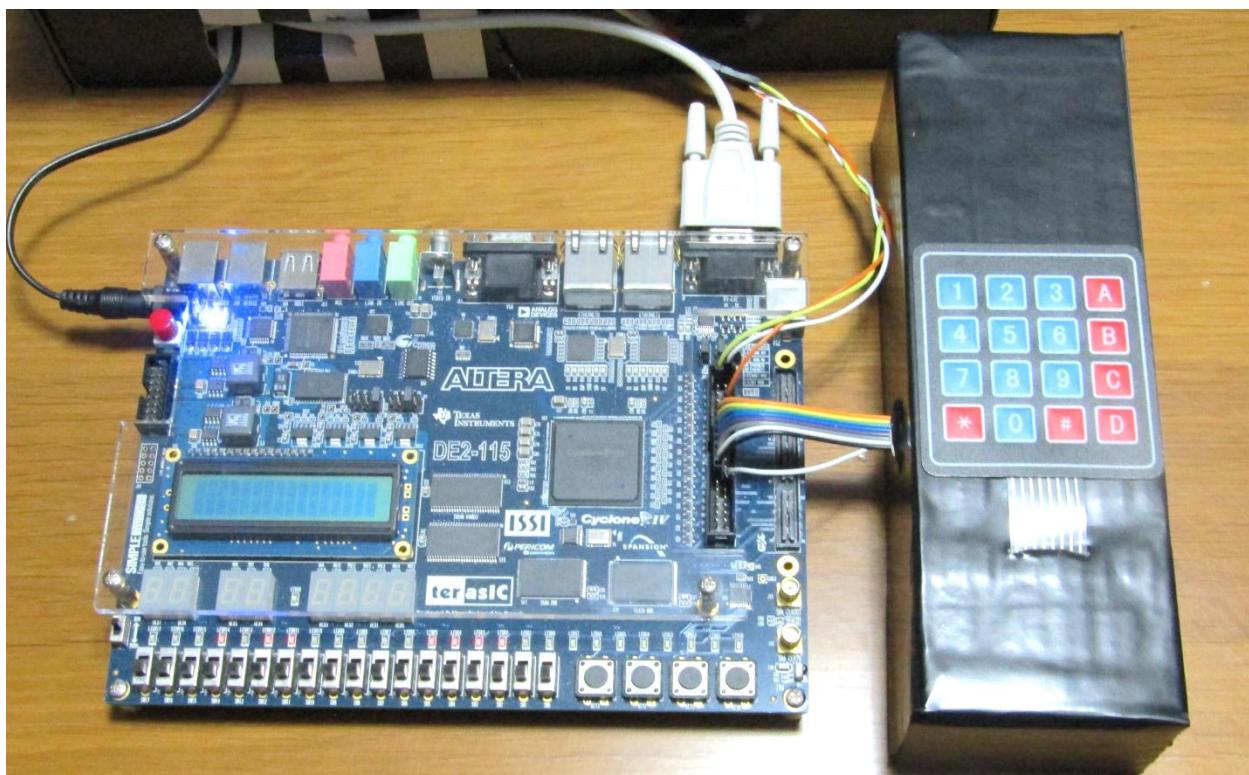


Fig. 3-1 Foto dell'utilizzo della scheda FPGA nel progetto

Per l'implementazione della parte che riguarda l'FPGA è stato usato il software Quartus II di Intel, che permette di creare programmi in vari linguaggi di programmazione per la descrizione dell'hardware e anche diagrammi a blocchi. I blocchi funzionali sono scritti in linguaggio VHDL e connessi tra loro da linee di segnale e porte logiche elementari a seconda delle esigenze.

Il software Quartus II si occupa della compilazione del codice VHDL, l'analisi e la sintesi, della progettazione dell'hardware (fitter and place), dell'assegnazione dei pin e della generazione dei file di programmazione. I file di output poi vengono caricati sulla scheda in fase di programmazione e ne rendono possibile l'utilizzo secondo quanto espresso dal codice.

Il progetto è stato strutturato sul Quartus II con una dipendenza ad albero, dove il file principale è posto come "Top Level Entity". Questa posizione è assunta dallo schema a blocchi principale chiamato "Gestione Accessi". Questo schema gestisce e individua il funzionamento degli input e degli output e di tutti i sottosistemi.

### 3.1 Gestione accessi

In Fig. 3.1-1. il simbolo del blocco Gestione Accessi:

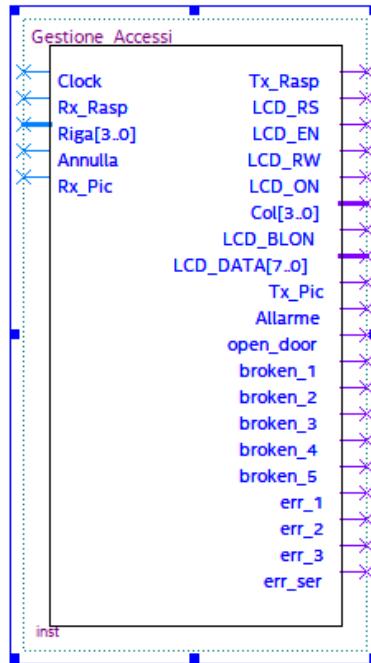


Fig. 3.1-1 Simbolo di Gestione Accessi

**Porte:** in Tabella 3.1-1. l'elenco degli input e output del blocco Gestione Accessi

Pin	I/O	bit	Descrizione	Nome Porte
clock	I	1	Clock collegato con l'oscillatore a 50 MHz della scheda	Clock_50
Rx_rasp	I	1	Pin Rx collegato al Raspberry per la comunicazione seriale	gpio[0]
Riga[3..0]	I	4	General purpose collegati alle 4 righe del tastierino	GPIO[10][12][14][16]
Annulla	I	1	Reset generale della safe mode, tipo switch	SW[0]
Rx_PIC	I	1	Pin Rx collegato al PIC Master	UART_RXD
Tx_Rasp	O	1	Pin Tx collegato al Raspberry per la comunicazione seriale	GPIO[1]
LCD_RS	O	1	Segnale di gestione LCD – Register Select	//
LCD_EN	O	1	Segnale di gestione LCD – Enable	//
LCD_RW	O	1	Segnale di gestione LCD – Read/Write	//
LCD_ON	O	1	Segnale di gestione LCD – On/Off	//
LCD_BLON	O	1	Segnale di gestione LCD – Backlight On	//
LCD_DATA	O	8	Segnale di gestione LCD – Carattere	//
Tx_PIC	O	1	Pin Tx collegato al PIC Master	UART_TXD
Allarme	O	1	Segnale di impianto non collegato (LED rosso)	LEDR[0]
Open_door	O	1	Segnale di porta aperta (LED rosso)	LEDR[1]
Broken_1	O	1	Segnale di malfunzionamento della porta 1 (LED rosso)	LEDR[6]
Broken_2	O	1	Segnale di malfunzionamento della porta 2 (LED rosso)	LEDR[7]
Broken_3	O	1	Segnale di malfunzionamento della porta 3 (LED rosso)	LEDR[8]
Broken_4	O	1	Segnale di malfunzionamento della porta 4 (LED rosso)	LEDR[9]
Broken_5	O	1	Segnale di malfunzionamento della porta 5 (LED rosso)	LEDR[10]
Err_1	O	1	Segnale di errore comunicazione CAN tipo 1 (LED rosso)	LEDR[15]
Err_2	O	1	Segnale di errore comunicazione CAN tipo 2 (LED rosso)	LEDR[16]
Err_3	O	1	Segnale di errore comunicazione CAN tipo 3 (LED rosso)	LEDR[17]
Err_ser	O	1	Segnale di errore comunicazione seriale (LED rosso)	LEDR[13]

Tabella 3.1-1 Input e output di Gestione Accessi

In Fig. 3.1-2 è mostrato il macro-blocco caricato sull'FPGA. In questa sezione verranno descritte le sue funzioni senza entrare nei dettagli, che verranno analizzati nei seguenti sotto paragrafi.

Le funzioni svolte sono:

- Ricevere e memorizzare il database di utenti dal Raspberry;
- Monitorare la sua presenza e in caso di necessità attivare il sistema di gestione “Safe Mode” che si occupa di
  - Permettere all’utente di identificarsi con ID e Password;
  - Verificare se tal utente ha diritto all’accesso;
  - Permettere la selezione della porta che si vuole aprire;
  - Inviare al PIC Master le informazioni per l’apertura della porta;
  - Inviare al PIC Master un segnale di alive;
- Inviare al Raspberry un segnale di alive;
- Segnalare all’utente diversi tipi di messaggi ed errori;

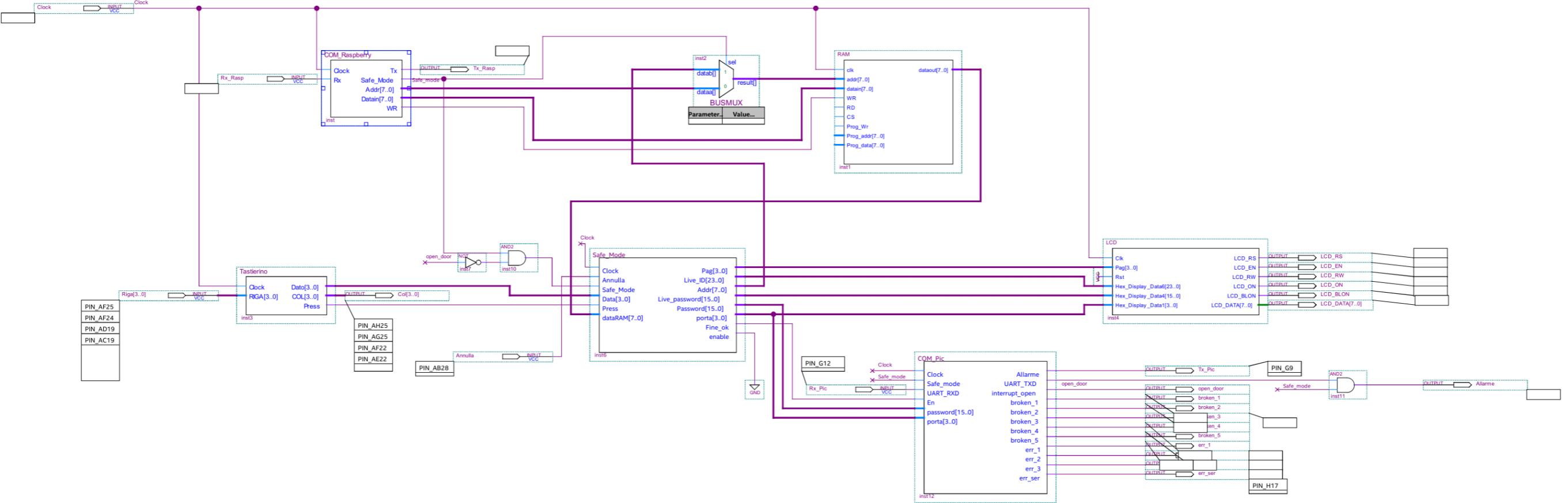


Fig. 3.1-2 Diagramma a blocchi di Gestione Accessi

### 3.1.1 COM\_Raspberry

In Fig. 3.1.1-1 il simbolo del blocco COM\_Raspberry:

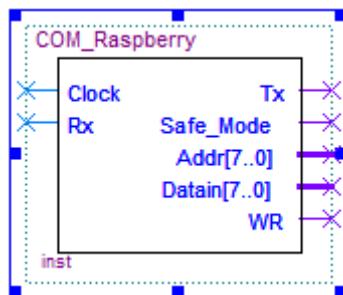


Fig. 3.1.1-1 Simbolo di Com\_Raspberry

**Porte:** in Tabella 3.1.1-1 l'elenco degli input e output del blocco COM\_Raspberry

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
Rx	I	1	Pin Rx collegato al Raspberry
Tx	O	1	Pin Tx collegato al Raspberry
Safe_Mode	O	1	Segnale di attivazione della Safe Mode
Addr[7..0]	O	8	Indirizzi della memoria RAM
DataIn[7..0]	O	8	Dati da memorizzare sulla RAM
WR	O	1	Segnale di enable per la scrittura su RAM

Tabella 3.1.1-1 Input e output di COM\_Raspberry

Questo blocco si occupa della comunicazione con il Raspberry. È collegato al blocco Safe Mode perché in caso di mancanza di segnale da parte dell'altro dispositivo deve essere attivato tale blocco. È collegato anche alla RAM perché deve salvare i dati degli utenti.

Le funzioni svolte sono:

- Inviare carattere di alive (55 esadec) ogni 3 secondi;
- Controllare di aver ricevuto il messaggio di alive negli ultimi 6 secondi;
- Analizzare i dati in arrivo e scriverli sulla RAM;

Di seguito (Fig. 3.1.1-2) il diagramma interno al blocco in esame:

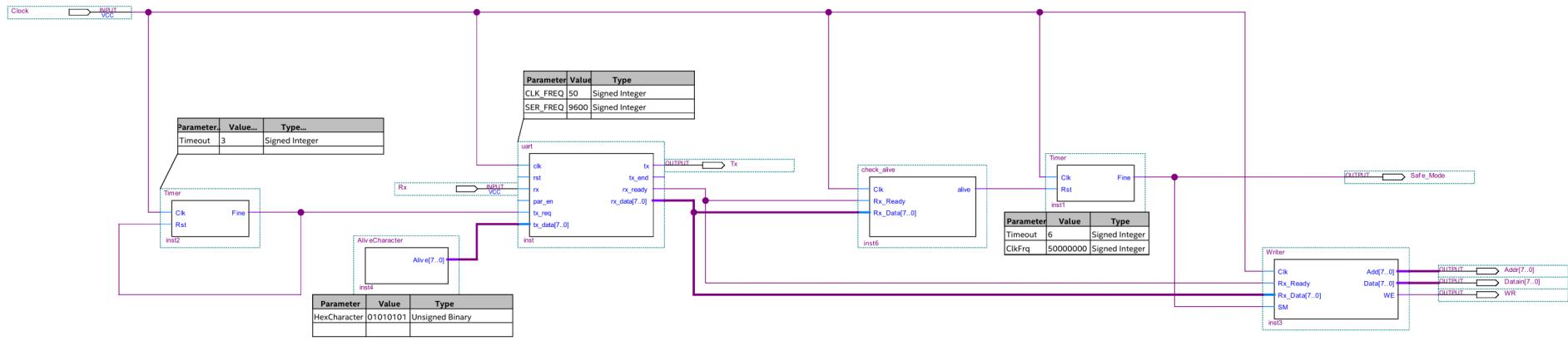


Fig. 3.1.1-2 Diagramma a blocchi di COM\_Raspberry

Come si può vedere dallo schematico allegato nella pagina precedente (Fig. 3.1.1-2), il primo timer scatta ogni 3 secondi, la sua uscita è collegata al suo reset perciò ogni volta che raggiunge il time out l'uscita alta lo fa resettare e riparte il conteggio.

L'uscita del timer è collegata al pin “Tx\_req” (richiesta di trasmissione) del blocco UART. Ogni volta che questo pin riceve un impulso alto effettua la trasmissione seriale del dato caricato nell'ingresso “Tx\_data”. In questo caso il dato è fornito dal blocco Alive\_Character che produce la costante 01010101 in uscita, cioè il carattere di alive (si noti che il carattere da inviare scelto è 55, quello ricevuto è invece CC).

Il blocco Check\_Alive, collegato al blocco UART, controlla ogni messaggio ricevuto e segnala quando uno di questi è un messaggio di alive, in tal caso l'uscita va alta per un ciclo di clock e resetta il secondo timer. Se non viene ricevuto tale messaggio per oltre 4 secondi, il timer setta alta la sua uscita, indicando che il Raspberry non sta funzionando correttamente; questo segnale viene mandato come output “Safe\_mode” che verrà successivamente utilizzato dagli altri blocchi per entrare nella modalità di funzionamento ridotta. Il blocco writer si occupa della gestione dei messaggi ricevuti, quando inizia la trasmissione dei dati degli utenti, analizza i dati e li salva nella RAM. Il suo funzionamento sarà descritto nel dettaglio nel *paragrafo successivo* (3.1.2).

### 3.1.1.1 Writer

In Fig. 3.1.1.1-1 il simbolo del blocco Writer:

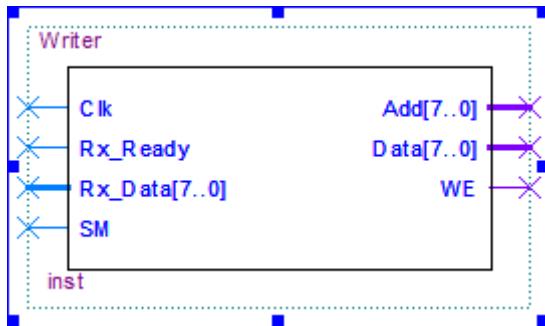


Fig. 3.1.1.1-1 Simbolo di Writer

**Porte:** in Tabella 3.1.1.1-1 l'elenco degli input e output del blocco Writer.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
Rx_Ready	I	1	Segnale di notifica di nuovi dati in ingresso
Rx_Data	I	8	Byte di dati in ingresso
SM	I	1	Safe Mode attiva (1) o disattiva (0)
Add	O	8	Indirizzo di memoria della RAM
Data	O	8	Dati da salvare nella RAM
WE	O	1	Write Enable, Abilitazione alla scrittura nella RAM

Tabella 3.1.1.1-1 Input e output di Writer

Questo blocco è una macchina a stati che permette di memorizzare dei dati a 8 bit sulla RAM interna dell'FPGA. I dati da scrivere provengono dal blocco UART per la comunicazione seriale, da cui derivano i segnali in ingresso "Rx\_Ready" e "Rx\_Data". Quando arriva un byte di informazioni dalla seriale, il segnale di "Rx\_Ready" rimane alto per un ciclo di clock mentre il dato in arrivo "Rx\_Data" si aggiorna allo stesso ciclo di clock e rimane invariato fino all'arrivo di un nuovo dato.

L'ingresso "SM" segnala che il sistema è entrato nella modalità ridotta; in questo caso, la funzione di scrittura si interrompe.

Le uscite "Add", "Data" e "WE" permettono al blocco di interagire con il blocco RAM che controlla la memoria volatile interna. In aggiunta si utilizzano due segnali interni che servono per gestire l'indirizzo in cui saranno salvati i dati in memoria.

Il protocollo di trasmissione del database dal Raspberry all'FPGA è stato illustrato nel *capitolo 2.4*. La macchina a stati di questo blocco si basa quindi su questo protocollo di trasmissione ripetuto per tutti i 16 spazi disponibili in memoria. In Tabella 3.1.1.1-2 è spiegato il sistema di codifica utilizzato per la trasmissione dei dati utente:

Invio utente	
AE (esadecimale)	Carattere di inizio (A) e indirizzo di riferimento (E)
26 (= 0010 0110 cifre convertite in BCD)	6 cifre per l'ID dell'utente
84 (= 1000 0100)	
57 (= 0101 0111)	
10 (= 0001 0000)	4 cifre per la password dell'utente
35 (= 0011 0101)	
BB (esadecimale)	Carattere di fine trasmissione utente

Tabella 3.1.1.1.2 Messaggio di trasmissione dati utente dal Raspberry

### **Elenco Stati:**

- **IDLE:**

Nello stato di IDLE, il blocco rimane in attesa di un nuovo dato in arrivo. Quando arriva un nuovo dato, si passa allo stato di WAIT4START. In questo blocco si azzerano i segnali “addr” e “count”.

- **WAIT4START**

In questo stato si controllano le prime 4 cifre significative del nuovo dato in arrivo. Se queste sono pari ad “A”, allora il carattere in arrivo è il byte di start e si salvano le 4 cifre meno significative nel segnale “addr” dato che individuano l’indirizzo in memoria dove salvare i dati successivi. In caso contrario, si ritorna in IDLE.

- **WAIT4DATA**

In questo stato, il blocco rimane in attesa di un nuovo dato in arrivo. Quando arriva un nuovo dato si passa allo stato START2WRITE.

- **START2WRITE**

Questo stato ha lo scopo di discriminare il tipo di dato in arrivo:

- Se il dato in arrivo è il carattere di alive, si ritorna in WAIT4DATA, ignorando così il carattere.
- Se il dato in arrivo è un esadecimale che contiene due numeri, bisogna scrivere il dato nella RAM:
  - All’indirizzo “Add” viene dato il valore del segnale “addr” per le 4 cifre più significative e “count” per le 4 cifre meno significative.
  - Alla porta “Data” viene assegnato il valore dei dati in ingresso “Rx\_Data”.
  - Si porta in alto il segnale “WE” per abilitare la scrittura della RAM.
  - Si passa allo stato END2WRITE
- Se il dato in arrivo è il carattere di Stop o di Start, si ritorna in IDLE, poiché o è terminata la comunicazione dell’utente, o è in arrivo un nuovo utente. Genericamente si torna in IDLE per qualsiasi dato in arrivo che non sia un dato o l’alive.

- **END2WRITE**

Questo stato disabilita la scrittura portando “WE” a ‘0’ e incrementa il segnale “count” in modo che se il prossimo dato in ingresso dovesse essere un’informazione, questa venga salvata nella locazione successiva della memoria. Il ciclo di clock successivo si torna allo stato WAIT4DATA.

- **SAFE\_MODE**

In qualsiasi stato, se l’ingresso “SM” è alto, ci si pone nello stato SAFE\_MODE. Si ritorna in IDLE quando lo stesso ingresso torna basso. Questo stato è necessario per permettere di terminare immediatamente il Writer quando l’FPGA deve lavorare in Safe Mode.

**Diagramma degli stati:** In Fig. 3.1.1.1-2 il diagramma di flusso del blocco Writer.

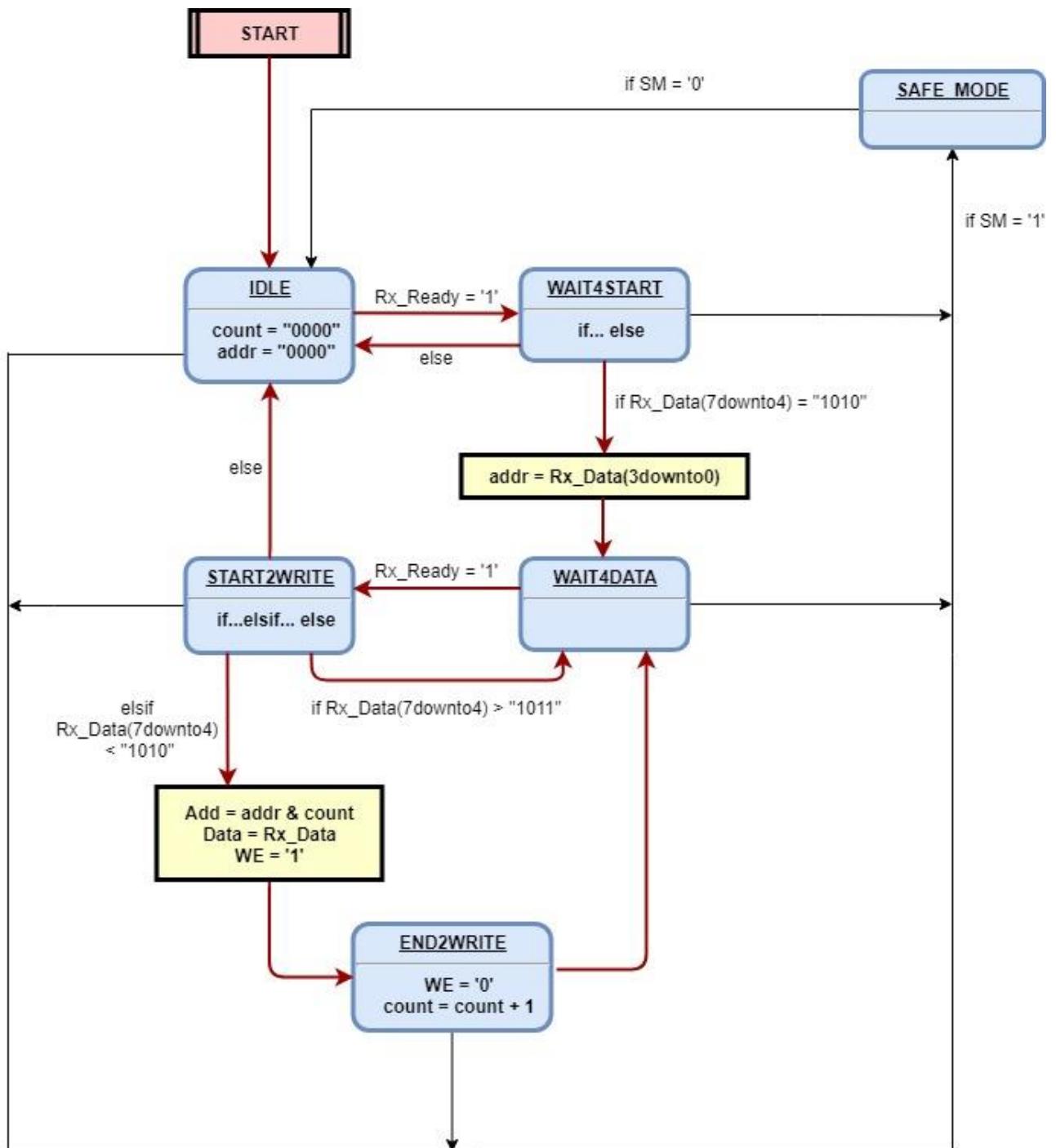


Fig. 3.1.1.1-2 Diagramma di flusso di Writer

## Codice VHDL:

```
library ieee;
use ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Questa macchina a stati preleva i dati dal blocco uart quando arriva il segnale di Rx_Ready.
-- Il Raspberry quando invia il database manda un utente alla volta.
-- I dati di ogni utente sono salvati sulla RAM interna e si è scelto di salvare al massimo 16 utenti.
-- La posizione in memoria di ogni utente è definito dalle 4 cifre più significative dell'indirizzo,
-- le 4 cifre meno significative identificano byte in memoria di informazioni diverse dello stesso utente.
-- I dati sono salvati in questo modo:
-- (I byte di dati e di indirizzi sono rappresentati in esadecimale (es: "A5"="10100101"), // <= Vuoto)
-- Utente 1: ID: "123456" Pass: "1234"
-- Utente 2: ID: "987654" Pass: "3210"
-- Memoria:
-- Add - Data
-- "00" - "12"
-- "01" - "34"
-- "02" - "56"
-- "03" - "12"
-- "04" - "34"
-- "05" - //
-- ...
-- "0F" - //
-- "10" - "98"
-- "11" - "76"
-- "12" - "54"
-- "13" - "32"
-- "14" - "10"
-- "15" - //
-- ...
-- I byte sono inviati seguendo questo protocollo:
-- "A0" "A" word di start. "0" rappresenta le 4 cifre più significative dell'indirizzo.
-- "12"
-- "34"
-- "56"
-- "12"
-- "34"
-- "BB" "B" word di fine utente. Si è scelto di ripeterlo 2 volte nello stesso byte.
-- "A1" "A" word di start. Inizio invio secondo utente. "1" è l'indirizzo del secondo utente.
-- "98"
-- "76"
-- "54"
-- "32"
-- "10"
-- "BB" Fine trasmissione utente.
-- 
-- A questo punto la macchina rimane in attesa di ulteriori ordini.
-- 
-- La macchina a stati ignora il carattere di alive "CC" inviato dal Raspberry ogni 2 secondi perché non fa parte dei dati.
```

-- In caso di disconnessione del Raspberry, si attiverà il segnale di Safe Mode ( $SM \leq 1$ )  
-- grazie ad un blocco di timer esterno a questo del writer che si attiva se per 5 secondi non riceve alcun dato.  
-- In questo caso, il writer si blocca e ritorna nello stato di attesa dati (Idle) quando il Raspberry viene riconnesso.

```

entity Writer is
    Port(
        Clk : in STD_LOGIC;                                     -- Clock
        Rx_Ready : in STD_LOGIC;                                -- Rx_Ready
        Rx_Data : in STD_LOGIC_VECTOR (7 downto 0);            -- Rx_Data
        SM : in STD_LOGIC;                                     -- Safe Mode. (SM <= 1 : Safe Mode attiva, SM <= 0 : Safe Mode off)
        Add : out STD_LOGIC_VECTOR (7 downto 0);                -- Address, Indirizzo di memoria della RAM
        Data : out STD_LOGIC_VECTOR (7 downto 0);                -- Data, Dati da salvare in RAM
        WE : out STD_LOGIC                                     -- Write Enable
    );
end;

begin
    process (Clk) is
    begin
        d74 <= Rx_Data(7 downto 4);

        if (Clk'event and Clk = '1') then
            case state is

                -- Idle: Rimane in attesa di un dato di start in arrivo.
                when Idle =>
                    count <= "0000"; -- Reset dell'indirizzo (...3-0)
                    addr <= "0000"; -- Reset dell'indirizzo (7-4...)
                    WE <= '0';
                    if Rx_Ready = '1' then
                        state <= Wait4Start;
                    end if;

                -- Ingresso in Safe Mode. Può avvenire in ogni stato.
                if SM = '1' then
                    state <= Safe_Mode;
                end if;

                -- Wait4Start: Se arriva la word di start, memorizza le 4 cifre più significative dell'indirizzo,
                -- che identificano
                -- la locazione in memoria dell'utente. (0-F, 16 utenti)
                -- Se non arriva la word di start, torna in Idle.
            end case;
        end if;
    end process;
end;

```

```

when Wait4Start =>
    if d74 = 10 then
        addr <= Rx_Data(3 downto 0);
        state <= Wait4Data;
    else state <= Idle;
    end if;

-- Safe Mode
if SM = '1' then
    state <= Safe_Mode;
end if;

-- Wait4Data: Attende un dato in arrivo.
when Wait4Data =>
    if Rx_Ready = '1' then
        state <= Start2Write;
    end if;

-- Safe Mode
if SM = '1' then
    state <= Safe_Mode;
end if;

-- Start2Write: Se il byte ricevuto è di Alive, torna in attesa di un dato.
-- Se il byte contiene informazioni (2 numeri), attiva il Write Enable e porta dati e indirizzo corretto
in uscita,
-- Il dato sarà scritto nella RAM nel prossimo ciclo di clock.
-- Se il byte ricevuto è di Fine Utente, torna in Idle.
when Start2Write =>
    if d74 > 11 then
        state <= Wait4Data;
    elsif d74 < 10 then
        Add <= addr & count;
        Data <= Rx_Data;
        WE <= '1';
        state <= End2Write;
    else
        state <= Idle;
    end if;

-- Safe Mode
if SM = '1' then
    state <= Safe_Mode;
end if;

-- End2Write: Porta a 0 il Write Enable e vai all'indirizzo successivo.
when End2Write =>
    WE <= '0';
    count <= count + 1;
    state <= Wait4Data;

-- Safe Mode

```

```

if SM = '1' then
    state <= Safe_Mode;
end if;

-- Safe Mode: Torna in Idle quando il segnale di Safe Mode è disattivato.
when Safe_Mode =>
    WE <= '0';
    if SM = '0' then
        state <= Idle;
    end if;

end case;
end if;
end process;

end beh;

```

### 3.1.2 Safe\_Mode

In Fig. 3.1.2-1 il simbolo del blocco Safe\_Mode:

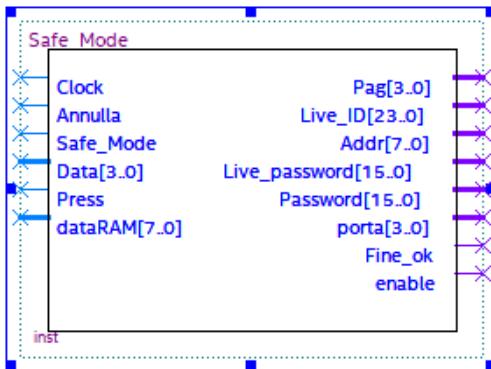


Fig. 3.1.2-1 Simbolo di Safe\_Mode

**Porte:** in Tabella 3.1.2-1 l'elenco degli input e output del blocco Safe\_Mode.

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
Annulla	I	1	Reset (switch)
Safe_mode	I	1	Safe Mode attiva (1) o disattiva (0)
Data	I	4	Cifra premuta dall'utente sul tastierino
Press	I	1	Segnale che indica la pressione di un tasto sul tastierino
DataRAM	I	8	Dati provenienti dalla RAM
Pag	O	4	Selettore delle pagine dell'LCD
Live_ID	O	24	ID in tempo reale (destinato all'LCD)
Live_password	O	16	Password in tempo reale per l'LCD
Password	O	16	Password inserita, destinata all'invio verso il PIC
Porta	O	4	Numero di porta inserito, verrà poi inviato al PIC
Fine_ok	O	1	Termine dell'inserimento dei dati, se è alto i dati salvati vengono inviati al PIC
Enable	O	1	Segnala l'utilizzo della memoria RAM
Addr	O	8	Indirizzi RAM

Tabella 3.1.2-1 Input e output di Safe\_Mode

Questo blocco gestisce il funzionamento quando il sistema è in modalità ridotta.

Le sue funzioni sono:

- Guidare l'utente nell'inserimento dei dati personali attraverso lo schermo LCD;
- Acquisire i dati inseriti dall'utente sul tastierino e mostrarli sul display;
- Effettuare la scansione del database per cercare corrispondenza dell'ID inserito;
- Confrontare la password inserita con quella salvata in memoria;
- Verificare che il numero di porta inserito sia corretto;
- Abilitare il segnale di invio verso il PIC Master per procedere all'apertura della porta.

Come si può osservare in Fig. 3.1.2-2 si può considerare il blocco State\_Machine\_Pagine come il coordinatore di questo macro-blocco, esso infatti attiva i vari blocchi, ne legge gli output e gestisce l'LCD. Il segnale di "Safe\_Mode" è quello che attiva la macchina a stati principale, quando il Raspberry risulta non funzionante, mentre sull'LCD viene mostrata la prima schermata: "Premere un tasto qualsiasi". Le diverse schermate sono salvate nel blocco LCD e vengono selezionate tramite il segnale di output "Pag". Al segnale di "press" viene mostrata la seconda schermata: "inserire ID"; ora la macchina a stati attiva il blocco ins\_ID che si occupa della

ricezione del dato inserito dall'utente. L'ID, che viene salvato appena ricevuto, viene utilizzato dal blocco Finder. Questo blocco è attivato da un flip flop jk che viene settato dal segnale di fine operazione del blocco di inserimento e viene resettato dal segnale di termine dello stesso blocco finder. Quest'ultimo si occupa di scorrere la memoria per trovare l'ID nel database: per farlo necessita di prendere il controllo della selezione degli indirizzi, il segnale di enable in output serve, a tale scopo, ad attivare il selettore descritto nel blocco Gestione\_Accessi. Se viene trovato l'ID, la macchina a stati riceve il segnale di "ID\_ok" e mostra la schermata di inserimento password, altrimenti mostra una schermata di errore. Il blocco Finder dopo aver trovato l'ID utente nel database, preleva la password ad esso associata nell'indirizzo di memoria adiacente e la invia al blocco Check\_psw. Durante l'inserimento password è attivo il blocco Ins\_psw, che riceve la password inserita e la salva. Al segnale di "End\_op" di questo blocco corrisponde l'attivazione del blocco Check\_psw, che restituisce alla macchina a stati generale il responso del confronto. Anche in questo caso ci sono due possibili schermate che seguono: quella di password errata o quella dell'inserimento porta; nel secondo caso viene attivato il blocco di inserimento porta che salva il numero della porta da aprire. Se il numero è corretto, quindi tutte le operazioni sono andate a buon fine viene settato il segnale di "Fine\_ok" per la durata di un ciclo di clock. Questo segnale andrà poi ad attivare il blocco COM\_PIC.

Nei prossimi sotto paragrafi verranno illustrati tutti i blocchi citati in dettaglio.

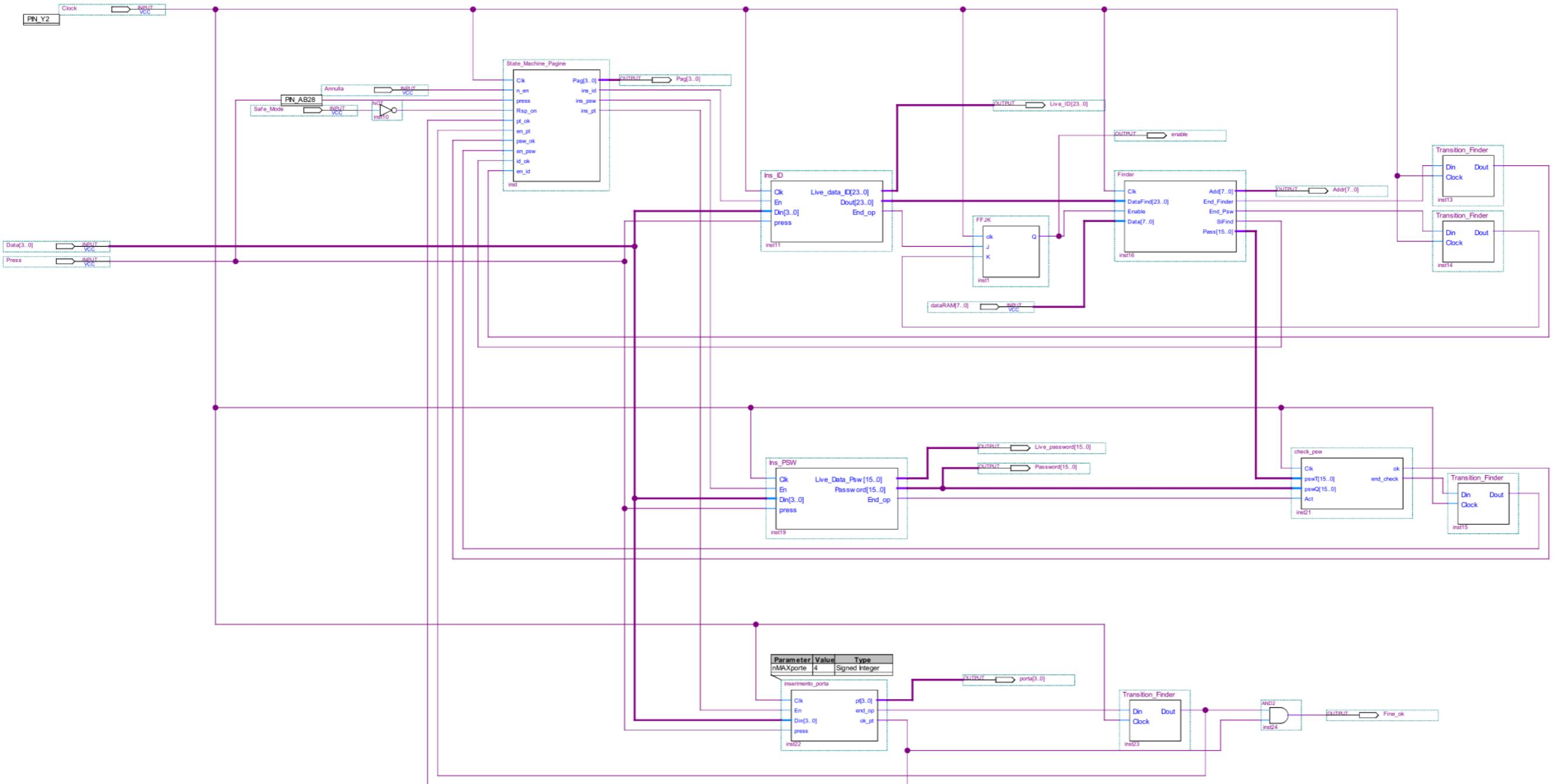


Fig. 3.1.2-2 Diagramma a blocchi di `Safe_Mode`

### 3.1.2.1 State\_Machine\_Pagine

In Fig. 3.1.2.1-1 il simbolo del blocco State\_Machine\_Pagine:

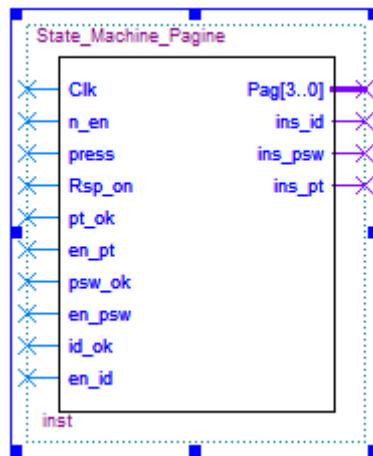


Fig. 3.1.2.1-1 Simbolo di State\_Machine\_pagine

**Porte:** in Tabella 3.1.2.1-1 l'elenco degli input e output del blocco State\_Machine\_Pagine.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
N_en	I	1	Reset collegato a uno switch (alto blocca il funzionamento)
Press	I	1	Segnale di pressione di un tasto
Rsp_on	I	1	Segnale di attivazione della Safe Mode, alto se il Raspberry è funzionante.
Pt_ok	I	1	Numero di porta inserito corretto
En_pt	I	1	Fine dell'operazione di controllo del numero porta
Psw_ok	I	1	Password inserita corretta
En_psw	I	1	Fine dell'operazione di confronto della password
ID_ok	I	1	ID inserito corretto
En_ID	I	1	Fine dell'operazione di ricerca ID
Pag[3..0]	O	4	Selettore numero di pagina da mostrare sull'LCD
Ins_ID	O	1	Segnale di attivazione blocco per l'inserimento ID
Ins_psw	O	1	Segnale di attivazione blocco per l'inserimento password
Ins_pt	O	1	Segnale di attivazione blocco per l'inserimento numero porta

Tabella 3.1.2.1-1 Input e Output di State\_Machine\_Pagine

Questa è la macchina a stati che fa scorrere le schermate e coordina l'utilizzo degli altri blocchi: prende in ingresso il segnale di pressione di un tasto dal tastierino e vari segnali da altri blocchi che conferiscono il risultato delle loro elaborazioni.

- **Fase 1 (Attivazione del blocco):** Il blocco viene attivato proprio dal segnale "Rsp on". Questo è il negato del segnale "Safe\_Mode" citato precedentemente. Questo perché quando l'FPGA è nella modalità ridotta ("Safe\_Mode" = '1') il Raspberry è off ("Rsp\_on" = '0') e viceversa. Una volta attivato il blocco è inizializzato sullo stato zero che fissa come uscita "Pag" il numero corrispondente alla schermata iniziale: "Premere un tasto per iniziare".
- **Fase 2 (Premere un tasto qualsiasi):** Al segnale di "press" passa allo stato uno in cui l'uscita "Pag" cambia e seleziona la schermata "Inserire ID", viene settato anche l'output che abilita il blocco Ins\_ID.

- **Fase 3 (Inserire ID):** Appena viene ricevuto alto per un ciclo di clock il segnale “en\_ID”, che segnala la fine dell’operazione di inserimento ID e ricerca di esso nel database, viene controllato l’esito della ricerca che è riportato nel segnale “ID\_ok”. Se questo segnale è basso si andrà nello stato 2 dove verrà selezionata la pagina “ID non riconosciuto”, per tornare allo stato 1 è necessario un segnale di pressione di un tasto. Se “ID\_ok” è alto si passa allo stato 3, dove viene settata la pagina “inserire password” e posta alta l’uscita “ins\_psw”, che attiva il blocco corrispondente.
- **Fase 4 (Inserire Password):** Quando l’input “en\_psw” è alto significa che l’inserimento password e il confronto sono terminati, quindi viene controllato l’ingresso “psw\_ok”, che identifica se la password inserita è corretta. Come nel caso precedente ci sono due possibili stati successivi: il 4 che seleziona la schermata “password non riconosciuta” (da cui si torna allo stato 3 con un segnale di “press”) oppure il 5 che prosegue con la schermata “inserire numero porta” e setta alto il segnale “ins\_pt”, il quale è collegato al blocco che si occupa di gestire l’inserimento del numero porta e verificare che sia corretto.
- **Fase 5 (Inserire numero porta):** Quando viene ricevuto il segnale “en\_pt” viene controllato l’ingresso “pt\_ok”, con la stessa logica dei casi precedenti: se è alto si passa allo stato 7, che fa mostrare la schermata “porta attiva”, se è basso si viene indirizzati allo stato 6 “porta errata”. In entrambi i casi si cambia di stato con il segnale di “press”: dallo stato 6 si torna al 5 per consentire l’inserimento di un nuovo numero porta, dal 7 si torna allo zero.
- **Fase 6 (Reset):** In qualsiasi momento la sequenza può essere interrotta utilizzando l’input “n\_en”, che quando è alto riporta la macchina allo stato zero, in modo tale che quando torna basso la sequenza riparta da capo. È associato a uno switch per scelta progettuale ma poteva essere associato forse più realisticamente a un pulsante (con associati i blocchi di debouncer e transition finder).

In qualsiasi momento torni attivo il Raspberry lo stato della macchina viene resettato a zero ma non viene attivato e l’uscita per la selezione della pagina è impostata sullo schermo completamente vuoto.

#### **Schema di funzionamento:**

##### **Condizione iniziale – RESET:**

Se il segnale di “n\_en” è alto lo stato viene resettato a zero, la macchina a stati è bloccata finché l’n\_en” non è posto a 0.

##### **Condizione OFF:**

Finché il segnale “Rsp\_on” rimane alto la macchina a stati rimane disattivata, il segnale di output “pag” rimane impostato a “1000”, che rappresenta lo schermo vuoto (Fig. 3.1.2.1-2). Solo quando il segnale “Rsp\_on” si resetta, si attiva la macchina a stati.

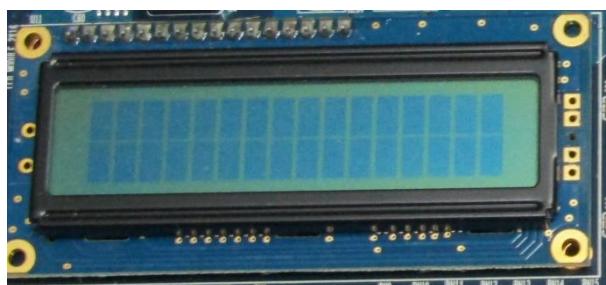


Fig. 3.1.2.1-2 Foto schermo LCD durante stato OFF

## Elenco Stati:

- **STATE 0**

In questo stato vengono resettati i segnali in uscita: "Pag", "ins\_ID", "ins\_psw" e "ins\_pt". Il cambiamento di stato avviene appena viene premuto un tasto, questo è segnalato dall'ingresso "Press". La schermata che vede l'utente è quella di benvenuto con l'indicazione "premi un tasto qualsiasi" (Fig. 3.1.2.1-3).



Fig. 3.1.2.1-3 Foto schermo LCD in stato di attesa

- **STATE 1**

Viene visualizzata la schermata: "inserire ID" (Fig. 3.1.2.1-4) e viene attivata l'uscita "Ins\_ID". Il cambiamento di stato avviene quando viene settato alto l'ingresso "en\_ID": se il segnale "ID\_ok" è alto la macchina passa allo stato 3, se è basso passa allo stato 2.

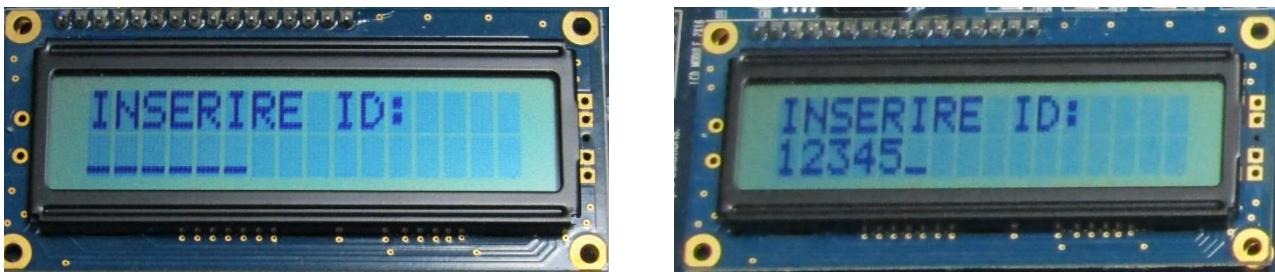


Fig. 3.1.2.1-4 Foto schermo LCD in state 1 con un esempio di inserimento ID

- **STATE 2**

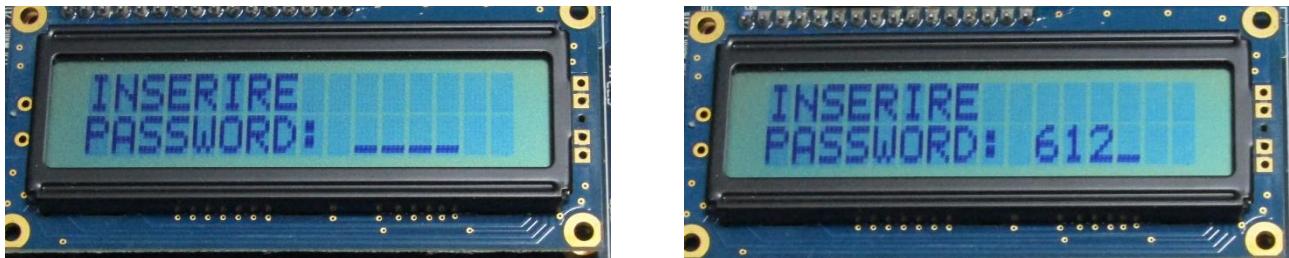
Viene mostrata la schermata di errore: "Utente non riconosciuto" (Fig. 3.1.2.1-5). Il segnale alto di "Press" farà tornare la macchina allo stato 1.



Fig. 3.1.2.1-5 Foto schermo LCD in state 2

- **STATE 3**

Sull'LCD viene selezionata la schermata “inserimento password”, che permette di inserire la password (Fig. 3.1.2.1-6) e viene posta a 1 l'uscita “en\_psw”. Il cambiamento di stato avviene quando viene settato alto l'ingresso “en\_ID”, se il segnale “ID\_ok” è alto la macchina passa allo stato 5, se è basso passa allo stato 4.



*Fig. 3.1.2.1-6 Foto schermo LCD in state 3 con esempio di inserimento password*

- **STATE 4**

Viene mostrata la schermata di errore: “password errata” (Fig. 3.1.2.1-7). Il segnale alto di “Press” farà tornare allo stato 3.

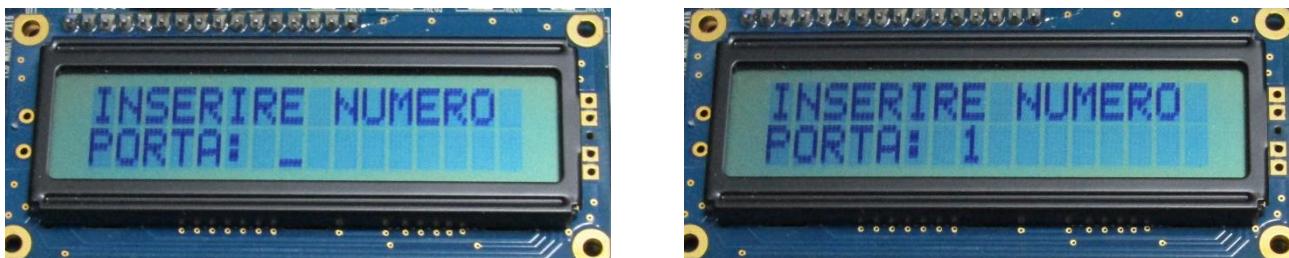


*Fig. 3.1.2.1-7 Foto schermo LCD in state 4*

- **STATE 5**

Viene visualizzata la schermata: “inserire numero porta”, dove si può inserire il numero della porta (Fig. 3.1.2.1-8) e viene attivata l'uscita “Ins\_pt”, che permette di inserire il numero della porta dove si richiede l'accesso.

Il cambiamento di stato avviene quando viene settato alto l'ingresso “en\_pt”: se il segnale “pt\_ok” è alto la macchina passa allo stato 7, se è basso passa allo stato 6.



*Fig. 3.1.2.3-8 Foto schemro LCD in state 5 con esempio di inserimento codice porta*

- **STATE 6**

Viene mostrata la schermata “numero errato” (Fig. 3.1.2.1-9). L’attivazione del segnale “press” consente di tornare allo stato 5.



Fig. 3.1.2.1-9 Foto schermo LCD in state 6

- **STATE 7**

L’inserimento del numero porta è andato a buon fine, viene visualizzata la schermata “porta i attiva” (Fig. 3.1.2.1-10) e un ulteriore segnale di “press” cambierà lo stato in zero.



Fig. 3.1.2.1-10 Foto schermo LCD in state 7

In Fig. 3.1.2.1-11 è rappresentato il diagramma di flusso del blocco State\_Machine\_pagine:

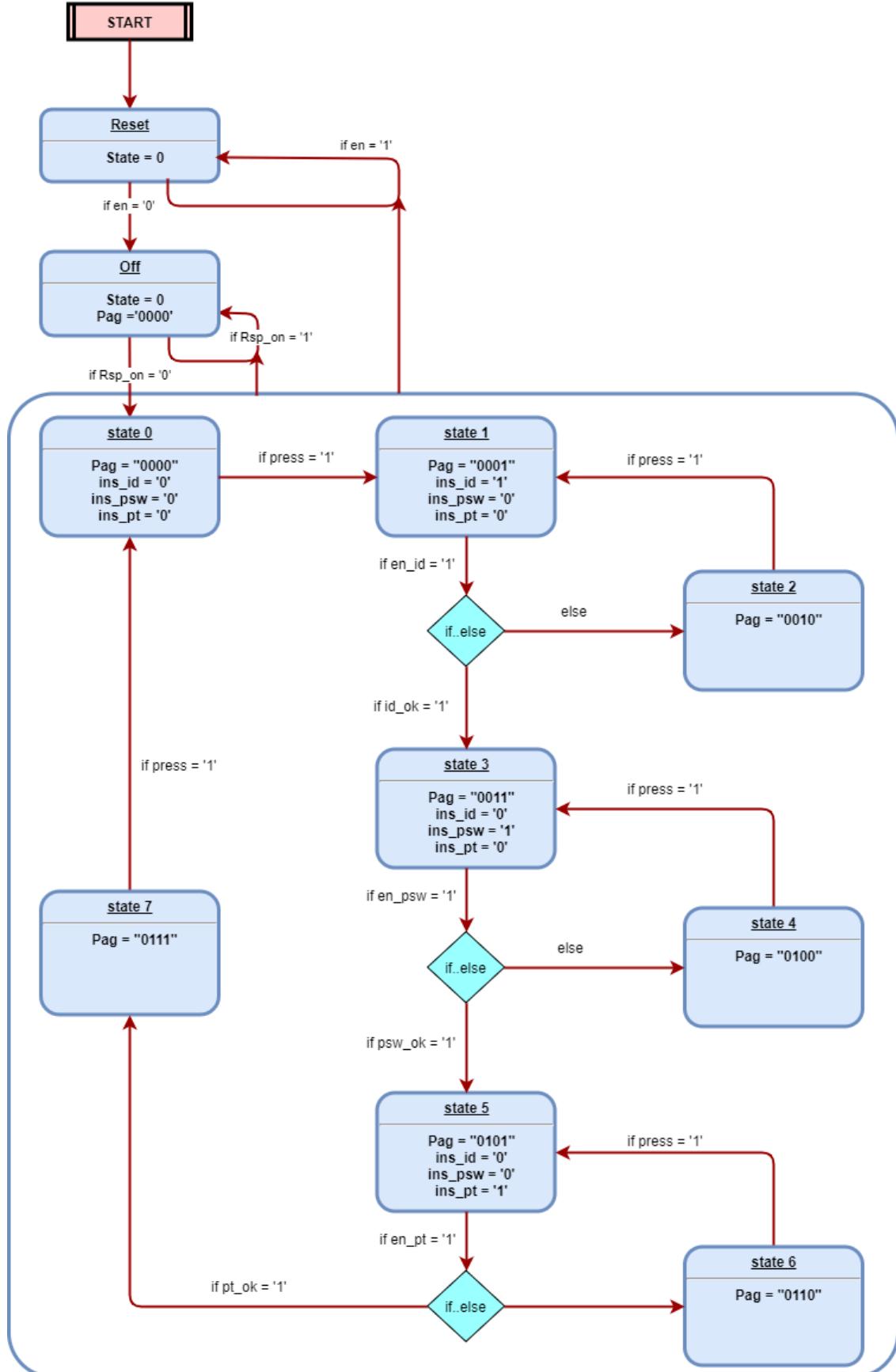


Fig. 3.1.2.1-11 Diagramma di flusso di State\_Machine\_pagine

## Codice VHDL:

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--questa è la macchina a stati che gestisce lo scorrimento delle schermate

entity State_Machine_Pagine is

Port (
    Clk : in std_logic;
    n_en : in std_logic;          -- posto a 1 blocca il funzionamento
    press : in std_logic;         -- segnale che arriva dal tastierino che determina la pressione di un
tasto
    Rsp_on :in std_logic;         -- segnale che indica se è presente il Raspberry, 1 se è presente, 0 se
è assente
    pt_ok : in std_logic;         -- sgn 1 se numero porta corretto
    en_pt : in std_logic;         -- sgn 1 se terminato il confronto del numero porta
    psw_ok :in std_logic;         -- sgn 1 se password inserita corretta
    en_psw : in std_logic;        -- sgn 1 se terminato il confronto password
    ID_ok : in std_logic;         -- sgn 1 se ID inserito presente nel database
    en_ID : in std_logic;         -- sgn 1 se terminata la ricerca dell'ID nel database
    Pag : out std_logic_vector(3 downto 0); --selettore delle pagine
    ins_ID : out std_logic:= '0';   -- alla schermata relativa all'inserimento ID attiva il blocco dedicato
    ins_psw: out std_logic:= '0';   -- alla schermata relativa all'inserimento password attiva il blocco
dedicato
    ins_pt: out std_logic:= '0');   -- alla schermata relativa all'inserimento porta attiva il blocco
dedicato
end State_Machine_Pagine;

architecture beh of State_Machine_Pagine is
signal state : integer := 0;

begin

process(Clk, press, ID_ok, psw_ok, pt_ok) is
begin

if (Clk'event and Clk = '1') then
if n_en = '1' then
    state <= 0;
else
    if Rsp_on = '1' then
        Pag <= "1000";
        state <= 0;
    else
        case state is
            when 0 =>
                ins_ID <= '0';
                ins_psw <= '0';
                ins_pt <= '0';
                Pag <= "0000"; --premere un tasto per iniziare
    end if;
end if;
end if;
end process;
end;
```

```

        if press = '1' then
            state <= 1;
        end if;
when 1 =>
    Pag <= "0001"; --inserire ID: __
    ins_ID <= '1';
    ins_psw <= '0';
    ins_pt <= '0';

    if en_ID = '1' then
        if ID_ok = '1' then
            state <= 3;
        else
            state <= 2;
        end if;
    end if;

when 2 =>
    Pag <= "0010"; --ID non riconosciuto
    if press = '1' then
        state <= 1;
    end if;

when 3 =>
    Pag <= "0011"; --inserire password: __
    ins_psw <= '1';
    ins_ID <= '0';
    ins_pt <= '0';
    if en_psw = '1' then
        if psw_ok = '1' then
            state <= 5;
        else
            state <= 4;
        end if;
    end if;

when 4 =>
    Pag <= "0100"; --password errata
    if press = '1' then
        state <= 3;
    end if;

when 5 =>
    Pag <= "0101"; --inserire porta: __
    ins_pt <= '1';
    ins_ID <= '0';
    ins_psw <= '0';

    if en_pt = '1' then
        if pt_ok = '1' then
            state <= 7;
        else

```

```

        state <= 6;
    end if;
    end if;
when 6 =>
    Pag <= "0110"; --porta errata
    if press = '1' then
        state <= 5;
    end if;
when 7 =>
    Pag <= "0111"; --porta attiva
    if press = '1' then
        state <= 0;
    end if;
when others =>
    Pag <= "1000";
end case;

end if;
end if;
end if;
end process;

end beh;
```

### 3.1.2.2 Ins\_ID

In Fig. 3.1.2.2-1 il simbolo di Ins\_ID.

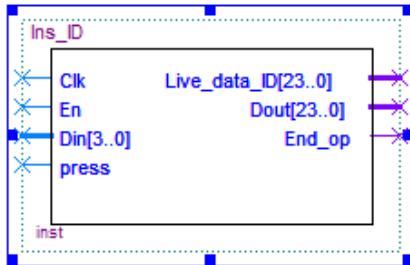


Fig. 3.1.2.2-1 Simbolo di Ins\_ID

**Porte:** in Tabella 3.1.2.2-1 gli input e output di Ins\_ID.

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
En	I	1	Attivazione del blocco
Din[3..0]	I	1	Cifra premuta sul tastierino
Press	I	1	Segnale di pressione sul tastierino
Live_data_ID[23..0]	O	24	ID aggiornato in tempo reale, destinato al display
Dout[23..0]	O	24	ID inserito
End_op	O	1	Segnale di fine operazione

Tabella 3.1.2.2-1 Input e output di Ins\_ID

Questo blocco salva l'ID utente.

L'acquisizione dell'ID è svolta dal blocco inserimento\_ID che sarà descritto nel prossimo paragrafo 3.1.2.2.1. Il dato in uscita da questo blocco viene inviato come segnale "Live\_data\_ID" di output perché rappresenta le cifre inserite in tempo reale ed è necessario per l'LCD. Al termine dell'inserimento il segnale "End\_op" del blocco Inserimento\_ID viene utilizzato dal transition finder per creare un impulso della durata di un ciclo di clock che serve a far scattare l'enable del blocco registro. Al ciclo di clock successivo il dato in uscita dal registro sarà aggiornato con il valore appena inserito dall'utente. In questo caso è stato utilizzato il blocco registro impostato con un vettore di 24 bit, infatti l'ID è composto da 6 cifre ciascuna rappresentata da 4 bit. Lo stesso segnale di enable viene portato al blocco Delay che semplicemente lo riproduce allo stesso modo nel ciclo di clock successivo per essere sincronizzato con il segnale di output "Dout" proveniente dal registro che sarà aggiornato quando il segnale "end\_op" sarà alto.

In Fig. 3.1.2.2-2 lo schema a blocchi di Ins\_ID.

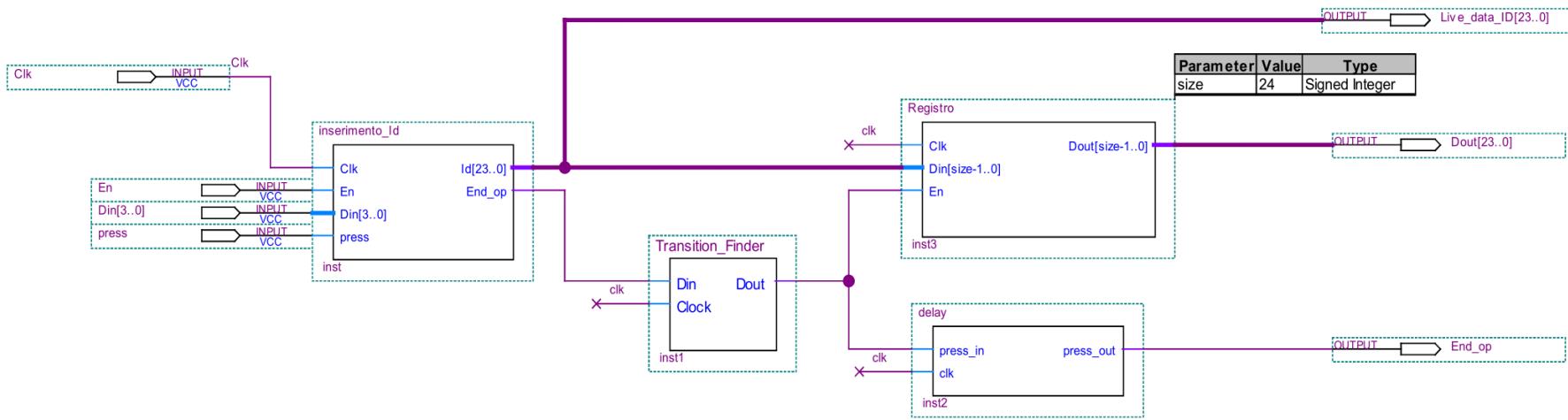


Fig. 3.1.2.2-2 Diagramma a blocchi di `Ins_ID`

### 3.1.2.2.1 Inserimento\_ID

In Fig. 3.1.2.2.1-1 il simbolo di inserimento\_ID.

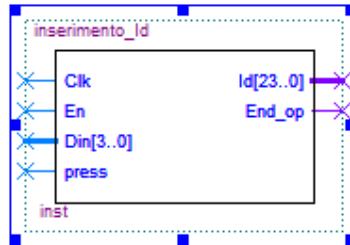


Fig. 3.1.2.2.1-1 Simbolo di inserimento\_ID

**Porte:** in Tabella 3.1.2.2.1-1 gli input e output di inserimento\_ID.

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
En	I	1	Attivazione del blocco
Din[3..0]	I	4	Cifra premuta sul tastierino
Press	I	1	Segnale di pressione sul tastierino
ID[23..0]	O	24	ID inserito, aggiornato in tempo reale
End_op	O	1	Segnale di fine operazione

Tabella 3.1.2.2.1-1 Input e output di inserimento\_ID

Questo blocco riceve l'ID inserito dall'utente tramite il tastierino.

Inizialmente ogni cifra è uguale a 10, in esadecimale, nel blocco LCD è stata fatta una piccola modifica rispetto al blocco fornитoci dal Prof. Santinelli, ogni cifra oltre al 9 viene mostrata come underscore, perciò l'ID è settato inizialmente come "AAAAAA" che sull'LCD viene mostrato come 6 trattini bassi.

Se viene premuto il tasto "#", corrispondente al numero 15, l'inserimento si resetta e si ritorna alla schermata iniziale senza nessuna cifra inserita. A mano a mano che l'utente preme i tasti, il tastierino invia il segnale di press e la cifra premuta, la cifra viene salvata sul segnale "ID" che viene quindi aggiornato in tempo reale.

Il blocco è in funzione solo se l'enable "En" è alto, altrimenti torna allo stato iniziale.

Questo blocco riceve l'enable dal blocco State\_Machine\_Pagine che lo attiva solo quando è necessario. Al termine dell'inserimento viene settato a uno il segnale "End\_op" che segnala che sono state inserite tutte e 6 le cifre di un ID.

#### Elenco Stati:

- **RESET 0:**

Il segnale "ID" viene resettato al valore "AAAAAA" che corrisponde sull'LCD a 6 trattini bassi, il segnale "End\_op" che indica la fine dell'inserimento ID viene resettato. Al segnale di "press" che dura per un ciclo di clock, si passa allo stato PRIMA.

Si entra in questo stato ogni volta che viene premuto il tasto cancelletto, corrispondente al numero 15 oppure quando l'ingresso "En" è basso. Si può procedere con gli altri stati solo tale ingresso è alto.

- **PRIMA:**

Il segnale di "press" rappresenta la pressione di un tasto, il numero premuto viene salvato in "Din" fino all'arrivo del dato successivo. Questo numero è la rappresentazione a 4 bit della cifra premuta, viene concatenato ad "AAAAA" in modo che sul display si veda la prima cifra e 5 trattini bassi. Al successivo segnale di "press" si passa allo stato successivo SECONDA. Se invece viene premuto il tasto cancelletto si torna allo stato RESET 0 quindi si annulla ogni cifra inserita, questo vale per tutti gli stati.

- **SECONDA:**

La cifra appena inserita è stata salvata nell'ingresso "Din", esso viene concatenato con i primi 4 bit dell'array di uscita "ID" e con "AAAAA" in questo modo sul display si vedrà la prima cifra inserita precedentemente, la seconda inserita ora e 4 trattini bassi. Il segnale di "press" effettua il passaggio allo stato TERZA.

- **TERZA:**

Sempre con lo stesso meccanismo la terza cifra viene concatenata ai primi 8 bit dell'uscita "ID" e altri 3 trattini bassi. Il segnale di "press" indica la transizione allo stato QUARTA.

- **QUARTA:**

La quarta cifra viene concatenata ai primi 12 bit dell'uscita "ID" e altri 2 trattini bassi.

- **QUINTA:**

La quinta cifra viene unita alle precedenti e rimane un solo trattino basso.

- **SESTA:**

La sesta cifra viene unita agli altri 20 bit che compongono l'uscita "ID" e viene settata alta l'uscita "End\_op". Essa tornerà a resettarsi quando si abbasserà il segnale "En", comandato dal segnale "ID\_En", che viene inviato dal blocco State\_Machine\_pagine. Il segnale "End\_op" di questo blocco, come si può vedere nello schematico precedente (Fig. 3.1.2.2-2) verrà elaborato da un transition finder per renderlo della durata di un ciclo di clock.

In Fig. 3.1.2.2.1-2 il diagramma di flusso di Inserimento\_ID.

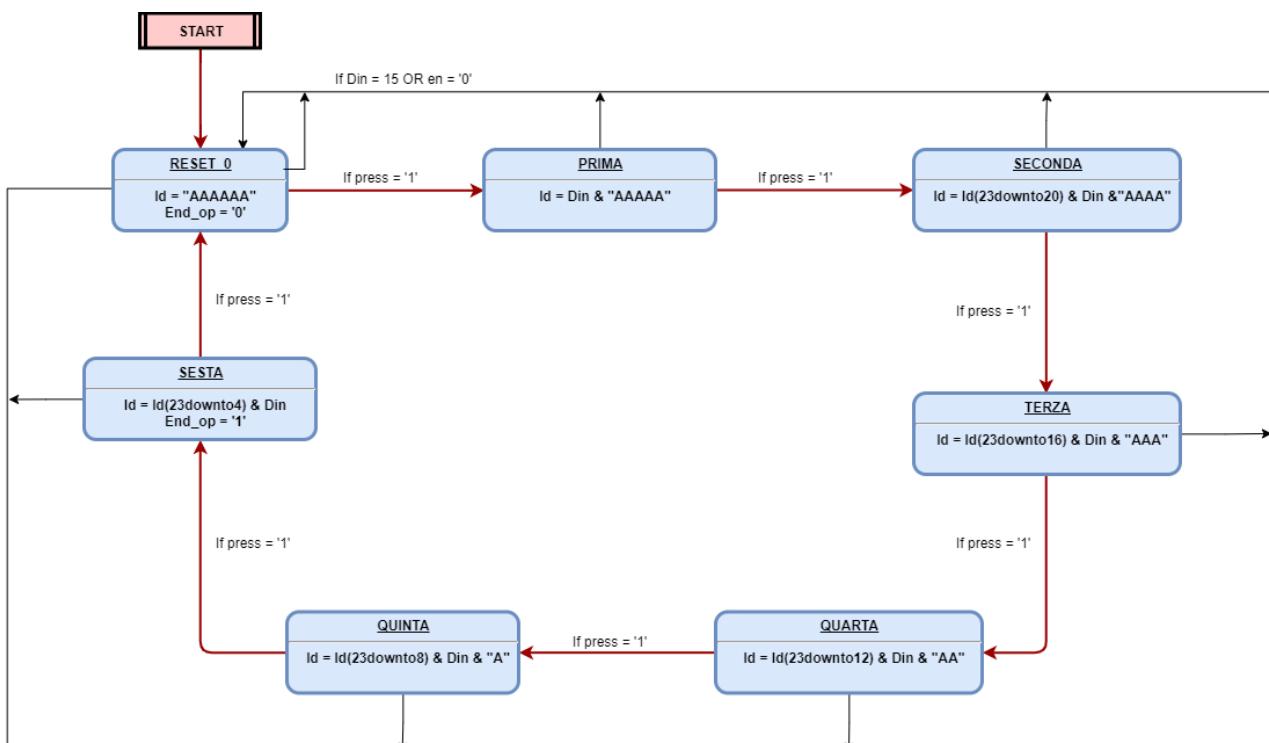


Fig. 3.1.2.2.1-2 Diagramma di flusso di Inserimento\_ID

## Codice VHDL

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity inserimento_ID is
Port (
    Clk :      in std_logic;
    En :       in std_logic;                                -- segnale di attivazione del blocco
    Din :      in std_logic_vector(3 downto 0);           -- cifra in ingresso, la cifra 15 è il reset
    press :    in std_logic;                               -- segnale che IDentifica la pressione di un tasto
    ID :       buffer std_logic_vector(23 downto 0):= X"AAAAAA";   -- l'ID composto da 6 cifre
    End_op :   out std_logic := '0';                      -- segnala la fine dell'inserimento
)
end inserimento_ID;

architecture beh of inserimento_ID is
type STATE_TYPE is (PRIMA, SECONDA, TERZA, QUARTA, QUINTA, SESTA, RESET_0);
signal state: STATE_TYPE:= RESET_0;

begin
process(clk, press, Din)
begin
    if (clk'event and clk = '1') then
        if En = '0' then
            state <= PRIMA;
            ID <= X"AAAAAA";
            End_op <= '0';
        else
            if press = '1' then
                if Din = 15 then
                    state <= PRIMA;
                    ID <= X"AAAAAA";
                    End_op <= '0';
                else
                    case state is
                        when RESET_0 =>
                            ID <= X"AAAAAA";
                            End_op <= '0';
                            state <= PRIMA;
                        when PRIMA =>
                            ID <= Din & X"AAAAA";
                            state <= SECONDA;
                        when SECONDA =>
                            ID <= ID(23 downto 20) & Din & X"AAA";
                            state <= TERZA;
                        when TERZA =>
                            ID <= ID(23 downto 16) & Din & X"AA";
                            state <= QUARTA;
                        when QUARTA =>
                            ID <= ID(23 downto 12) & Din & X"AA";
                            state <= QUINTA;
                    end case;
                end if;
            end if;
        end if;
    end if;
end process;

```

```
when QUINTA =>
    ID <= ID(23 downto 8) & Din & X"A";
    state <= SESTA;
when SESTA =>
    ID <= ID(23 downto 4) & Din;
    End_op <= '1';
    state <= RESET_0;
end case;
end if;
end if;
end if;
end if;
end process;
end beh;
```

### 3.1.2.3 Finder

In Fig. 3.1.2.3-1 il simbolo del blocco Finder:

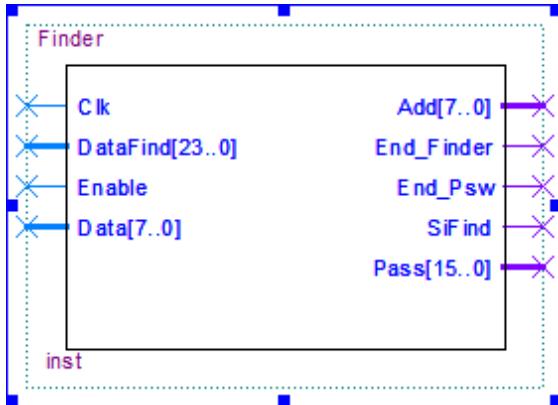


Fig. 3.1.2.3-1 Simbolo di Finder

**Porte:** in Tabella 3.1.2.3-1 l'elenco degli input e output di Finder.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
DataFind	I	24	Id utente da cercare in memoria
Enable	I	4	Segnale di inizio ricerca
Data	I	8	Dati in ingresso dalla memoria
Add	O	8	Indirizzo della memoria selezionato
End_Finder	O	1	Segnale di fine ricerca
End_Psw	O	1	Segnale di invio password
SiFind	O	1	Segnale di individuazione utente in memoria
Pass	O	16	Password

Tabella 3.1.2.3-1 Input e output di Finder

Questo blocco è una macchina a stati, la quale interagisce con il blocco RAM che controlla la lettura e la scrittura della memoria RAM interna all'FPGA. Il suo scopo è quello di individuare un utente nella memoria e, se trovato, inviare in uscita la password associata all'utente. La macchina a stati si può dividere in due sotto-parti:

- **Ricerca Utente**

Noto l'ID dell'utente da cercare, la macchina a stati inizia a cercarlo in memoria. Se questo viene trovato, viene memorizzato l'indirizzo per il passaggio successivo e i segnali "SiFind" e "End\_Finder" si portano a 1. Se, una volta cercato l'ID in tutta la memoria, questo non viene trovato si interrompe la ricerca e il segnale "End\_Finder" si porta a 1.

- **Invio Password**

Nota la posizione in memoria dell'utente, viene memorizzata la password associata all'utente su un segnale di registro e viene inviata infine all'uscita "Pass".

La macchina a stati segue la disposizione dei dati in memoria, che è stata già descritta nel paragrafo riguardante il blocco writer. Le 6 cifre degli ID degli utenti sono salvate nelle locazioni "X0", "X1" e "X2" della memoria, dove X è il generico indirizzo esadecimale dell'utente. Nel primo stato confronta le due cifre in memoria all'indirizzo "00" con le prime due cifre dell'ID: se queste corrispondono, confronta le due cifre in memoria all'indirizzo "01" con le seconde due cifre dell'ID. Questo continua anche per le ultime due cifre dell'ID. Se le due cifre non dovessero combaciare a ciascun livello, l'utente in memoria all'indirizzo "0" non è

quello cercato, per cui si passa all'indirizzo "1", iniziando a confrontare le prime due cifre all'indirizzo "10" con le prime 2 cifre dell'ID. Una volta trovato l'utente si attiva il segnale "SiFind" e si va allo stato "Fine\_Ricerca". Questo procedimento continua fino a trovare l'utente. Se anche l'utente all'indirizzo "F" non dovesse essere quello cercato, si va allo stato "Fine\_Ricerca" senza avere il segnale "SiFind" attivato, che è il discriminante per sapere se la password in arrivo è quella dell'utente cercato o no.

Una volta iniziata la fase di invio password, si prelevano le 4 cifre della password associata all'utente dalla memoria, le quali si trovano nelle locazioni "X3" e "X4", dove X è il generico indirizzo esadecimale dell'utente. Si porta infine il valore dell'indirizzo nella porta di uscita "Pass" e si porta a 1 il segnale "End\_Psw".

Ogni volta che è necessario aggiornare l'indirizzo, lo si fa in uno stato a parte: quando si aggiorna l'indirizzo, il nuovo dato in memoria non arriva allo stesso ciclo di clock, ma necessita di un ulteriore stato di attesa (HOLD) per dare il tempo al blocco RAM di potersi aggiornare.

#### **Elenco Stati:**

- **STATO\_INIZIALE**

È lo stato nel quale vengono resettati i segnali "Add", "End\_Finder", "End\_Psw" e "SiFind". A questo stato, segue uno stato di attesa (HOLD) di un ciclo di clock.

- **LEGGIPAG1**

Confronta le prime due cifre dell'ID (del segnale "DataFind" in ingresso al blocco) con le cifre memorizzate nell'indirizzo "X0" di memoria, dove "X" rappresenta le 4 cifre più significative dell'indirizzo "Add" e identifica la posizione in memoria dell'utente X-simo. Se combaciano, si va allo stato Add1\_2 per passare all'indirizzo "X1", continuando a confrontare le cifre successive dell'ID nello stato LEGGIPAG2. Se non combaciano, si va allo stato Add16 che porta l'indirizzo al valore "(X+1)0": questo perché, dato che non combaciano le due cifre, l'utente nella locazione "X" non è quello cercato, per cui si passa all'utente "(X+1)". Se "X" è pari a "F" in esadecimale, si è terminata la ricerca in tutta la memoria; l'utente quindi non è stato trovato e si va allo stato FINE\_RICERCA.

- **LEGGIPAG2**

Confronta la terza e la quarta cifra dell'ID utente con le due cifre memorizzate nell'indirizzo "X1" di memoria. Se combaciano, si va allo stato Add1\_3 per passare all'indirizzo "X2", continuando a confrontare le cifre successive dell'ID nello stato LEGGIPAG3, altrimenti si va allo stato Add15, che porta l'indirizzo al valore "(X+1)0", per poi tornare allo stato LEGGIPAG1 per iniziare a confrontare le cifre dell'ID con l'utente "(X+1)" .

- **LEGGIPAG3**

Confronta la terza e la quarta cifra dell'ID utente con le due cifre memorizzate nell'indirizzo "X1" di memoria. Se combaciano, l'utente è stato trovato. Il segnale "SiFind" viene settato ad "1", indicando il corretto ritrovamento dell'utente in memoria. Si passa così allo stato FINE\_RICERCA per iniziare la fase di invio password, se invece le cifre non combaciano si va allo stato Add15, che porta l'indirizzo al valore "(X+1)0", per poi tornare allo stato LEGGIPAG1 per iniziare a confrontare le cifre dell'ID con l'utente "(X+1)" .

- **Add1\_2**

Incrementa di 1 l'indirizzo "add" ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Successivamente si va allo stato LEGGIPAG2.

- **Add1\_3**

Incrementa di 1 l'indirizzo “add” ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Successivamente si va allo stato LEGGIPAG3.

- **Add16**

Incrementa di 16 l'indirizzo “add” ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Successivamente si va allo stato LEGGIPAG1.

- **Add15**

Incrementa di 15 l'indirizzo “add” ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Successivamente si va allo stato LEGGIPAG1.

- **Add14**

Incrementa di 14 l'indirizzo “add” ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Successivamente si va allo stato LEGGIPAG1.

- **FINE\_RICERCA**

Pone il segnale “End\_Finder” ad 1, il che identifica la fine del processo di ricerca utente. Questo segnale, tuttavia, non discrimina l'avvenuto ritrovamento dell'utente in memoria. Viene seguito dallo stato Add1Pass.

- **Add1Pass**

Incrementa di 1 l'indirizzo “add” ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Questo perché le prime due cifre della password sono salvate nell'indirizzo successivo a quello cui corrispondono le ultime due cifre dell'ID Utente. Successivamente si va allo stato Salva.

- **Salva**

Si assegna al segnale “pass1\_2” il segnale “Data”, in questo modo le prime due cifre della password vengono memorizzate. Si va successivamente allo stato Add2Pass.

- **Add2Pass**

Incrementa di 1 l'indirizzo “add” ed è seguito da uno stato di HOLD della durata di un ciclo di clock, che serve per far aggiornare correttamente l'indirizzo in tutto il programma. Questo perché le ultime due cifre della password sono salvate nell'indirizzo successivo. Successivamente si va allo stato Salva2.

- **Salva2**

Si assegna al segnale “pass3\_4” il segnale di ingresso “Data”, in questo modo le ultime due cifre della password vengono memorizzate. Si va successivamente allo stato INVIA\_PASS.

- **INVIA\_PASS**

La password viene combinata nel segnale di uscita “Pass” dai due segnali “pass1\_2” e “pass3\_4”. Si porta ad 1 il segnale “End\_Psw”, che identifica il corretto aggiornamento del segnale “Pass”. La password sarà quella

corretta solo se il segnale "Si\_Find" è pari a 1, altrimenti è solo il valore della password dell'ultimo utente scansionato.

Di seguito (Fig. 3.1.2.3-2) il diagramma di flusso del blocco Finder:

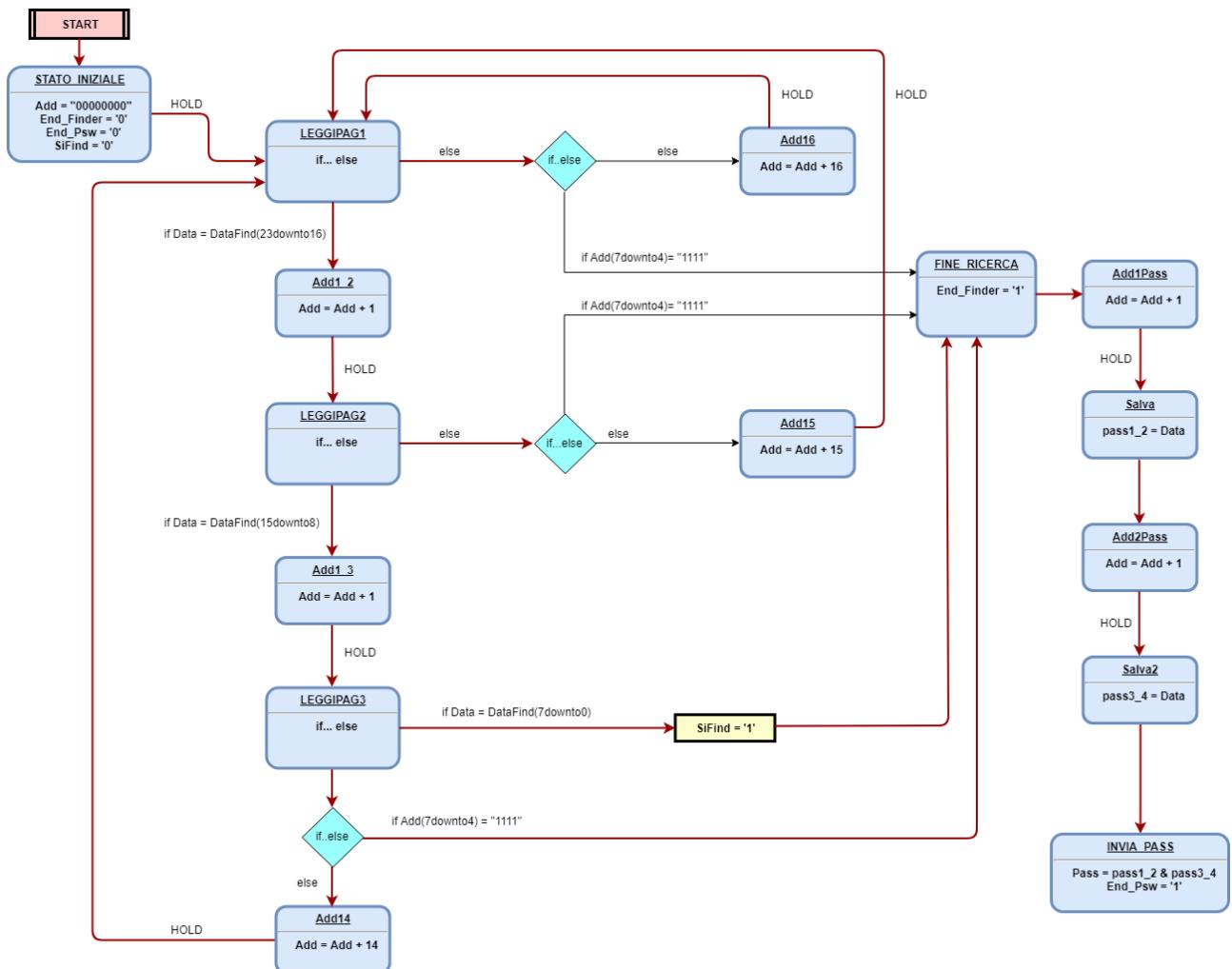


Fig. 3.1.2.3-2 Diagramma di flusso di Finder

### Codice VHDL:

```

library ieee;
use ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Questo blocco riceve in ingresso un ID utente e scorre la memoria per
-- verificare che l'utente sia presente.
-- Se lo trova salva la password corrispondente a tale ID e la trasmette in uscita.

```

*entity Finder is*

#### Port

```

Clk : in STD_LOGIC;
DataFind : STD_LOGIC_VECTOR (23 downto 0);          -- id da cercare
Enable : in STD_LOGIC;
Add : buffer STD_LOGIC_VECTOR (7 downto 0);          -- indirizzo di memoria

```

```

Data : in STD_LOGIC_VECTOR (7 downto 0); -- dati presenti in una cella di memoria
End_Finder: out STD_LOGIC; -- fine dell'operazione di ricerca
End_Psw : out STD_LOGIC; --fine dell'operazione di salvataggio password
SiFind : out STD_LOGIC; -- alto se la ricerca ha prodotto risultato positivo
Pass: out STD_LOGIC_VECTOR (15 downto 0); -- password rilevata
end Finder;

architecture beh of Finder is

type STATE_TYPE is (STATO_INIZIALE, LEGGIPAG1, LEGGIPAG2, LEGGIPAG3, FINE_RICERCA, Add1_2,
Add1_3, Add16, Add15, Add14, Hold, INVIA_PASS, Add1Pass, Add2Pass, Salva, Salva2);
signal state : STATE_TYPE := STATO_INIZIALE;
signal state_2 : STATE_TYPE;
signal pass1_2 : STD_LOGIC_VECTOR (7 downto 0);
signal pass3_4 : STD_LOGIC_VECTOR (7 downto 0);

begin
process (Clk,Data,Enable) is
begin
    if (Clk'event and Clk = '1') then
        if (Enable = '1') then
            case state is
                when STATO_INIZIALE =>
                    Add <= "00000000";
                    End_Finder <= '0';
                    End_Psw <= '0';
                    SiFind <= '0';
                    state <= Hold;
                    state_2 <= LEGGIPAG1;
                when LEGGIPAG1 =>
                    if DataFind = X"000000" then
                        state <= FINE_RICERCA;
                    else
                        if Data = DataFind (23 downto 16) then
                            state <= Add1_2;
                        else
                            state <= Add16;
                            if Add (7 downto 4) = "1111" then
                                state <= FINE_RICERCA;
                            end if;
                        end if;
                    end if;
                when LEGGIPAG2 =>
                    if Data = DataFind (15 downto 8) then
                        state <= Add1_3;
                    else
                        state <= Add15;
                        if Add (7 downto 4) = "1111" then
                            state <= FINE_RICERCA;
                        end if;
                    end if;
                when LEGGIPAG3 =>
                    if Data = DataFind (7 downto 0) then

```

```

        SiFind <= '1';
        state   <= FINE_RICERCA;
    else
        state <= Add14;
        if Add (7 downto 4) = "1111" then
            state   <= FINE_RICERCA;
        end if;
    end if;
when FINE_RICERCA =>
    End_Finder <= '1';
    state <= Add1Pass;
when Add1_2 =>
    Add <= Add + 1;
    state <= Hold;
    state_2 <= LEGGIPAG2;
when Add1_3 =>
    Add <= Add + 1;
    state <= Hold;
    state_2 <= LEGGIPAG3;
when Add16 =>
    Add <= Add + 16;
    state <= Hold;
    state_2 <= LEGGIPAG1;
when Add15 =>
    Add <= Add + 15;
    state <= Hold;
    state_2 <= LEGGIPAG1;
when Add14 =>
    Add <= Add + 14;
    state <= Hold;
    state_2 <= LEGGIPAG1;
when Add1Pass =>
    Add <= Add + 1;
    state <= Hold;
    state_2 <= Salva;
when Add2Pass =>
    Add <= Add + 1;
    state <= Hold;
    state_2 <= Salva2;
when Salva =>
    pass1_2 <= Data;
    state <= Add2Pass;
when Salva2 =>
    pass3_4 <= Data;
    state <= INVIA_PASS;
when INVIA_PASS =>
    Pass <= pass1_2 & pass3_4;
    End_Psw <= '1';
when Hold =>
    state <= state_2;
end case;
else

```

```
        state <= STATO_INIZIALE;  
    end if;  
end if;  
end process;  
end beh;
```

### 3.1.2.4 Ins\_Psw

In Fig. 3.1.2.4-1 il simbolo del blocco Ins\_Psw:

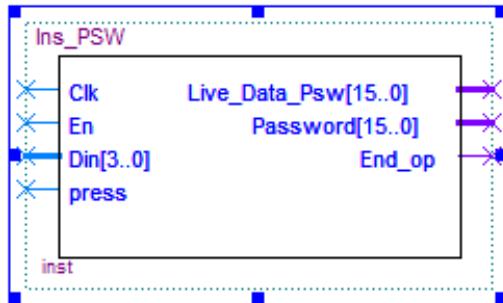


Fig. 3.1.2.4-1 Simbolo di Ins\_Psw

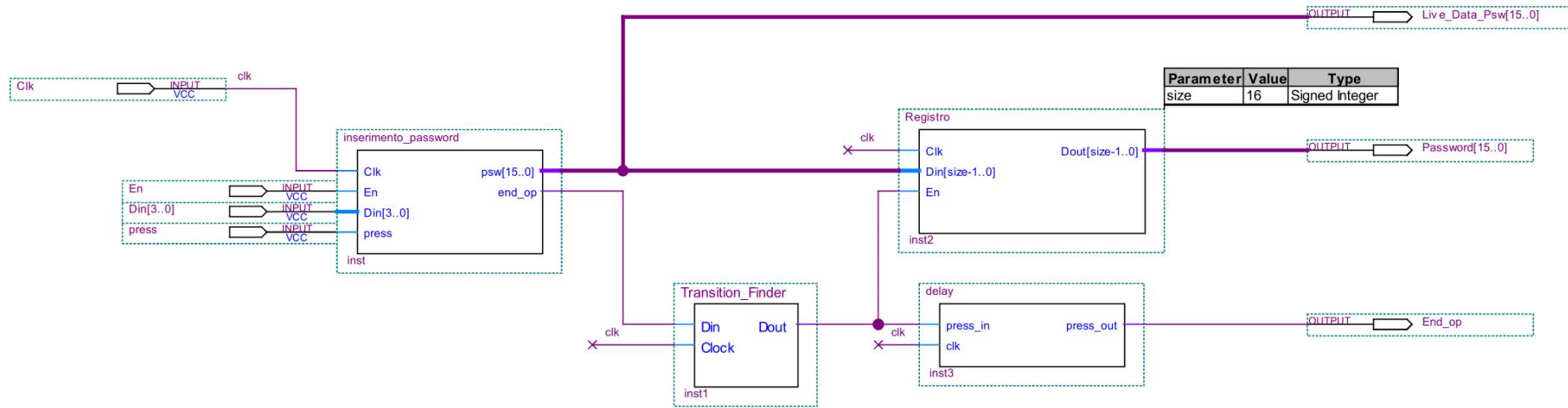
**Porte:** in Tabella 3.1.2.4-1 l'elenco degli input e output di Ins\_Psw

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
En	I	1	Attivazione del blocco
Din[3..0]	I	1	Cifra premuta sul tastierino
Press	I	1	Segnale di pressione sul tastierino
Live_data_psw[15..0]	O	16	Password aggiornata in tempo reale, destinata al display
Password[15..0]	O	16	Password inserita
End_op	O	1	Segnale di fine operazione

Tabella 3.1.2.4-1 Input e output di Ins\_Psw

Questo è il blocco che salva la password, del tutto simile al blocco di inserimento ID: l'unica differenza è la dimensione del registro, impostato per salvare un vettore di 16 bit. Ciò perché la password è composta da 4 cifre e il blocco chiave in questo caso è il blocco Inserimento password, che verrà descritto nel *paragrafo successivo (3.1.2.4.1)*.

Di seguito (Fig. 3.1.2.4-2) il diagramma a blocchi del blocco Ins\_Psw:



### 3.1.2.4.1 Inserimento\_password

In Fig. 3.1.2.4.1-1 il simbolo del blocco Inserimento\_password:

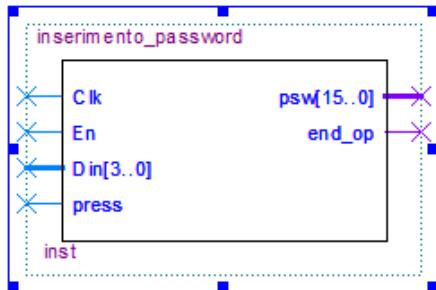


Fig. 3.1.2.4.1-1 Simbolo di Inserimento\_password

**Porte:** in Tabella 3.1.2.4.1-1 l'elenco degli input e output di Inserimento\_password

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
En	I	1	Attivazione del blocco
Din	I	4	Cifra premuta sul tastierino
Press	I	1	Segnale di pressione sul tastierino
Psw	O	16	Id inserito, aggiornato in tempo reale
End_op	O	1	Segnale di fine operazione

Tabella 3.1.2.4.1-1 Input e output di Inserimento\_password

Questo blocco riceve la password inserita dall'utente tramite il tastierino; come funzionamento rispecchia pienamente il blocco Inserimento\_Id.

**Elenco Stati:**

- **RESET 0:**

Il segnale di “psw” viene resettato al valore “AAAA”, che corrisponde sull’LCD a 4 trattini bassi; il segnale di “End\_op”, che indica la fine dell’inserimento password, viene resettato. Al segnale di “press”, che dura per un ciclo di clock, si passa allo stato PRIMA. Si entra in questo stato ogni volta che viene premuto il tasto cancelletto, corrispondente al numero 15, oppure quando l’ingresso “En” è basso. Si può procedere con gli altri stati solo tale ingresso è alto.

- **PRIMA:**

Il segnale di “press” rappresenta la pressione di un tasto, il numero premuto viene salvato in “Din” fino all’arrivo del dato successivo. Questo numero è la rappresentazione a 4 bit della cifra premuta, viene concatenato ad “AAA” in modo che sul display si veda la prima cifra e 3 trattini bassi. Al successivo segnale di press si passa allo stato successivo: SECONDA. Se invece viene premuto il tasto cancelletto si torna allo stato RESET 0, quindi si annulla ogni cifra inserita (questo vale per tutti gli stati).

- **SECONDA:**

La cifra appena inserita è stata salvata nell’ingresso “Din”, che viene concatenato con i primi 4 bit dell’array di uscita “psw” e con “AA”; in questo modo sul display si vedrà la prima cifra, inserita precedentemente, la seconda, inserita ora, e 2 trattini bassi. Il segnale di “press” effettua il passaggio allo stato TERZA.

- **TERZA:**

Sempre con lo stesso meccanismo, la terza cifra viene concatenata ai primi 8 bit dell'uscita "psw" e un altro trattino basso. Il segnale di "press" indica la transizione allo stato QUARTA.

- **QUARTA:**

La quarta cifra viene unita agli altri 12 bit che compongono l'uscita "psw" e viene settata alta l'uscita "End\_op". Quest'ultima tornerà a resettarsi quando si abbasserà il segnale "En", comandato dal segnale "psw\_En", che viene inviato dal blocco State\_Machine\_pagine. Il segnale "End\_op" di questo blocco, come si può vedere nello schema in Fig. 3.1.2.4.2, verrà elaborato da un transition finder per renderlo della durata di un ciclo di clock.

Di seguito (Fig. 3.1.2.4.1-2) il diagramma di flusso del blocco Inserimento\_password:

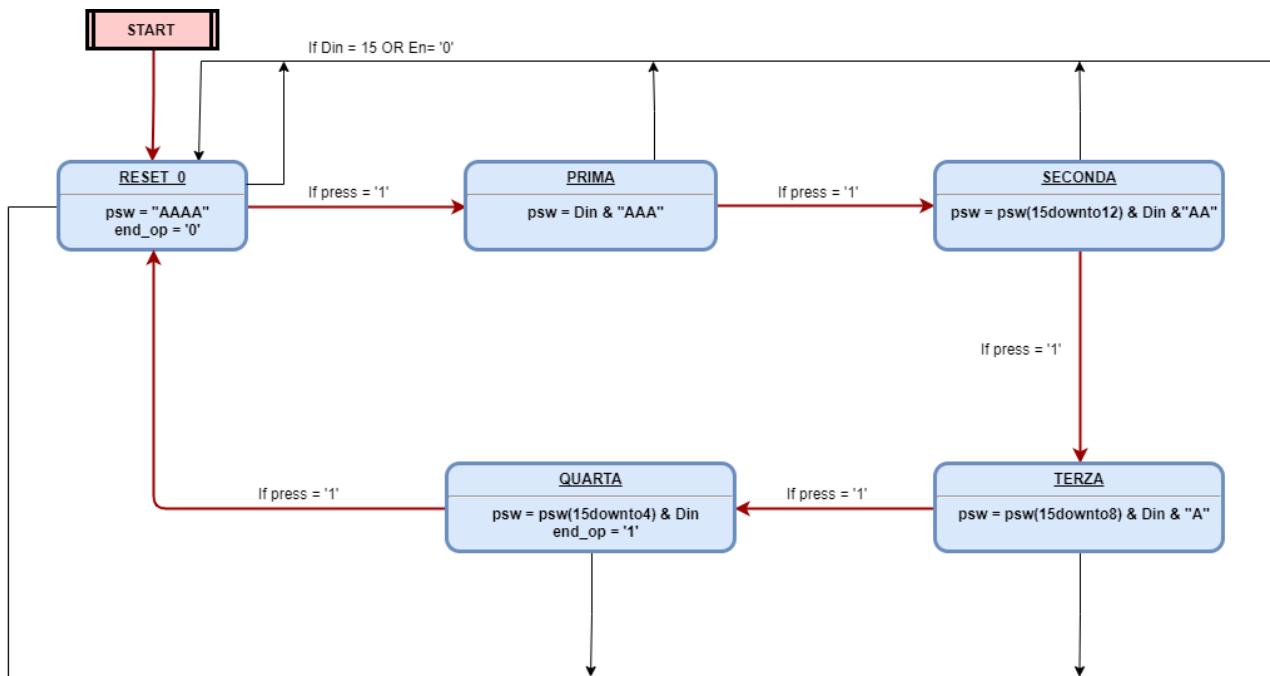


Fig. 3.1.2.4.1- 2 Diagramma di flusso di Inserimento\_password

## Codice VHDL

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity inserimento_password is

Port (
    Clk : in std_logic;
    En : in std_logic;                                -- segnale di attivazione del blocco
    Din : in std_logic_vector(3 downto 0);            -- cifra in ingresso, la cifra 15 è il reset
    press : in std_logic;                            -- segnale che identifica la pressione di un tasto
    psw : buffer std_logic_vector(15 downto 0):=X"AAAA"; -- la password composta da 6 cifre
    end_op : out std_logic);                         -- segnala la fine dell'inserimento
end inserimento_password;

architecture beh of inserimento_password is
type STATE_TYPE is (PRIMA, SECONDA, TERZA, QUARTA, RESET_0);
signal state : STATE_TYPE:= RESET_0;

begin
process(clk, press, Din)
begin
    if (clk'event and clk = '1') then
        if En = '0' then
            end_op <= '0';
            psw <= X"AAAA";
            state <= PRIMA;
        else
            if press = '1' then
                if Din = 15 then
                    state <= PRIMA;
                    psw <= X"AAAA";
                    end_op <= '0';
                else
                    case state is
                        when RESET_0 =>
                            psw <= X"AAAA";
                            end_op <= '0';
                            state <= PRIMA;
                        when PRIMA =>
                            psw <= Din & X"AA";
                            state <= SECONDA;
                        when SECONDA =>
                            psw <= psw(15 downto 12) & Din & X"AA";
                            state <= TERZA;
                        when TERZA =>
                            psw <= psw(15 downto 8) & Din & X"A";
                            state <= QUARTA;
                        when QUARTA =>
                            psw <= psw(15 downto 4) & Din;
                    end case;
                end if;
            end if;
        end if;
    end if;
end process;
end;
```

```
        end_op <= '1';
        state <= RESET_0;
    end case;
end if;
end if;
end if;
end if;
end process;
end beh;
```

### 3.1.2.5 Check\_psw

In Fig. 3.1.2.5-1 il simbolo del blocco Check\_psw:

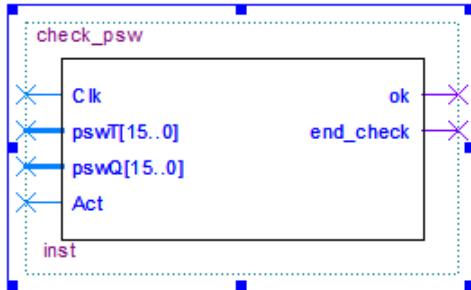


Fig. 3.1.2.5-1 Simbolo di Check\_psw

**Porte:** in Tabella 3.1.2.5-1. l'elenco degli input e output di Check\_psw

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
pswT	I	16	Password presente nel database
pswQ	I	16	Password inserita dall'utente
Act	I	1	Segnale di attivazione del blocco
ok	O	1	Segnale esito confronto password
End_check	O	1	Segnale di fine operazione

Tabella 3.1.2.5-1 Input e Output di Check\_psw

Questo blocco effettua il controllo della password: riceve in ingresso la password vera, estratta dal database dal blocco Finder, e la password inserita dall'utente dal blocco Inserimento\_Password e, appena il segnale "Act" è alto, effettua il confronto tra le due. In uscita produce due segnali: uno per il termine dell'operazione, "End\_Check", che diventa alto per un ciclo di clock e un secondo segnale "ok" per l'esito, alto se le due password corrispondono.

### Codice VHDL

```

library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity check_psw is
Port (
    Clk : in std_logic;
    pswT: in std_logic_vector(15 downto 0);          -- password corretta
    pswQ : in std_logic_vector(15 downto 0);          -- password da confrontare
    Act : in std_logic;                               -- segnale di attivazione
    ok : out std_logic:= '0';                         -- esito del confronto: 1 se positivo, 0 se negativo
    end_check : out std_logic:= '0'                  -- segnale di fine operazione
);
end check_psw;

```

```
architecture beh of check_psw is
begin
process(clk) is
begin
  if (clk'event and clk = '1') then
    if Act = '1' then
      if pswT = pswQ then
        ok <= '1';
        end_check <= '1';
      else
        ok <= '0';
        end_check <= '1';
      end if;
    else
      end_check <= '0';
      ok <= '0';
    end if;
  end if;
end process;
end beh;
```

### 3.1.2.6 Inserimento\_porta

In Fig. 3.1.2.6-1 il simbolo del blocco Inserimento\_Porta:

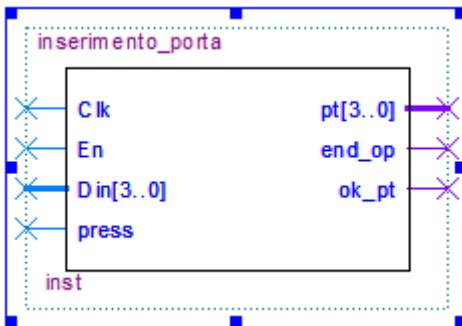


Fig. 3.1.2.6-1 Simbolo di inserimento\_porta

**Porte:** in Tabella 3.1.2.6-1. l'elenco degli input e output di Inserimento\_Porta.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
En	I	1	Segnale di attivazione del blocco
Din	I	4	Cifra digitata sul tastierino
Press	I	1	Segnale di pressione di un tasto
pt	O	4	Numero porta inserito
End_op	O	1	Segnale di fine operazione
Ok_pt	O	1	Segnale di numero porta corretto

Tabella 3.1.2.6-1 Input e output di inserimento\_porta

Questo blocco riceve il numero di porta inserito dall'utente tramite il tastierino e, pur essendo molto simile ai blocchi di inserimento ID e password, presenta delle differenze: dopo aver inserito il numero della porta è necessario che l'utente prema anche il tasto di "Ok" o un tasto qualsiasi; la pressione del tasto fa sì che venga verificato che il numero della porta sia corretto o no. In entrambi i casi viene settato a uno il segnale di fine inserimento "end\_op". Ricordiamo che questo segnale viene rilevato dal blocco State\_Machine\_Pagine, che gestisce le schermate e riceve in input anche il segnale "ok\_pt", determinando quale schermata mostrare. Nei casi di inserimento ID e password, il dato veniva inviato ai successivi blocchi appena si raggiungeva il numero di cifre necessario; la scelta di aggiungere il tasto di conferma è dovuta al fatto che, essendo la porta rappresentata da un'unica cifra, questa sarebbe subito scomparsa dall'LCD e sostituita, in un lasso di tempo impercettibile, dalla schermata successiva. Anche negli altri casi l'ultima cifra non è visibile all'occhio umano, ma ciò non comporta alcun problema, anzi accelera le operazioni per l'utente; in caso di ID o password sbagliata l'utente sarebbe comunque avvertito e invitato a reinserire il dato. Il blocco funziona solo se l'enable "En", controllato dal blocco State\_Machine\_Pagine, è alto.

Il tasto "#" permette di cancellare il numero inserito e tornare alla schermata di inserimento numero porta senza alcuna cifra inserita. Il numero massimo di porte è una variabile che si può modificare in base alle esigenze.

Come si può vedere nello schematico generale del blocco Safe\_Mode (Fig. 3.1.2-2), il segnale "End\_op" e "ok\_pt" vengono portati ad una porta AND e a un transition finder, che segnala il termine di tutte le operazioni di inserimento dati attraverso l'output "fine\_ok". Tale segnale serve ad attivare il blocco Com\_PIC.

## Elenco Stati:

- **RESET 0:**

il segnale di uscita “pt” è resettato al valore “A”, che corrisponde ad un trattino basso sul display. Vengono resettate anche le uscite “end\_op” e “ok\_pt”. Si esce da questo stato solo se l’ingresso “En” è alto e viene premuto un tasto.

- **PRIMA:**

La cifra inserita, che si trova nel dato in ingresso “Din”, viene salvata nella variabile di output “pt” e quindi mostrata sull’LCD. Il segnale di “press” fa scattare lo stato successivo: se il tasto premuto è il cancelletto si ritorna allo stato RESET 0 altrimenti si procede con lo stato CONFRONTO.

- **CONFRONTO:**

Il numero di porta inserito viene controllato: se è diverso da zero e minore del numero massimo di porte più uno allora è corretto, e verrà settato alto il segnale “ok\_pt”, in caso contrario tale segnale sarà basso. Si torna allo stato RESET 0 quando il segnale “En” torna a zero. Questo segnale è controllato dall’uscita “pt\_en” del blocco State\_machine\_pagine.

Di seguito (Fig. 3.1.2.6-2) il diagramma di flusso del blocco Inserimento\_porta:

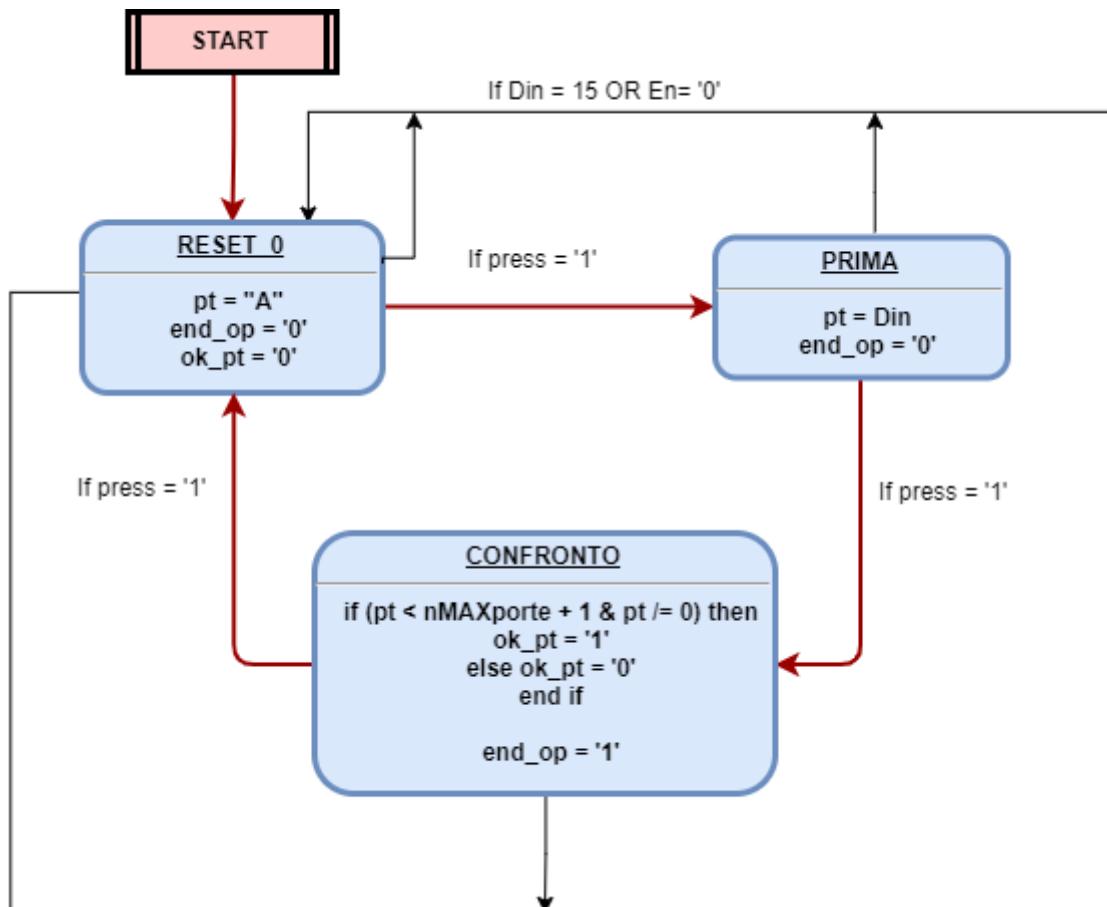


Fig. 3.1.2.6-2 Diagramma di flusso di inserimento\_porta

## Codice VHDL

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity inserimento_porta is
generic (nMAXporte: INTEGER:= 4);

Port (
    Clk : in std_logic;
    En : in std_logic;                                -- segnale di attivazione del blocco
    Din : in std_logic_vector(3 downto 0);            -- cifra in ingresso, la cifra 15 è il reset
    press : in std_logic;                            -- segnale che identifica la pressione di un tasto
    pt : buffer std_logic_vector(3 downto 0):= X"A"; -- il numero porta, una cifra
    end_op : out std_logic:= '0';                    -- segnala la fine dell'inserimento
    ok_pt : out std_logic := '0';                   -- alto se il numero di porta è corretto. basso se il
                                                numero di porta eccede quello del massimo numero di porte o è zero
end inserimento_porta;

architecture beh of inserimento_porta is
type STATE_TYPE is (PRIMA, RESET_0, CONFRONTO);
signal state : STATE_TYPE:= RESET_0;
signal next_command : STATE_TYPE;

begin
process(clk, press, Din)
begin
    if (clk'event and clk = '1') then
        if En = '0' then
            end_op <= '0';
            ok_pt <= '0';
            pt <= X"A";
            state <= PRIMA;
        else
            if press = '1' then
                if Din = 15 then
                    state <= PRIMA;
                    pt <= X"A";
                    end_op <= '0';
                else
                    case state is
                        when RESET_0 =>
                            pt <= X"A";
                            state <= PRIMA;
                            end_op <= '0';
                        when PRIMA =>
                            pt <= Din;
                            state <= CONFRONTO;
                            end_op <= '0';
                        when CONFRONTO =>
                            if (pt < nMAXporte + 1 and pt /= 0) then

```

```
        ok_pt <= '1';
    end if;
    end_op <= '1';
    state <= RESET_0;
end case;
end if;
end if;
end if;
end if;
end process;
end beh;
```

### 3.1.3 COM\_PIC

In Fig. 3.1.3-1 il simbolo del blocco COM\_PIC:

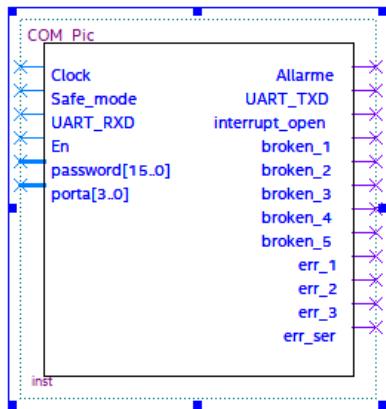


Fig. 3.1.3-1 Simbolo di COM\_PIC

**Porte:** in Tabella 3.1.3-1 l'elenco degli input e output di COM\_PIC.

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
Safe_mode	I	1	Segnale di attivazione di Safe Mode
UART_RXD	I	1	Pin Rx della trasmissione seriale
En	I	1	Segnale di attivazione del blocco
password	I	16	Password che deve essere inviata al PIC
porta	I	4	Numero porta che deve essere inviato al PIC
Allarme	O	1	Segnale di PIC Master non collegato (LED rosso)
UART_TXD	O	1	Pin Tx della trasmissione seriale
interrupt_open	O	1	Segnale di porta aperta (LED rosso)
Broken_1	O	1	Segnale di malfunzionamento della porta 1 (LED rosso)
Broken_2	O	1	Segnale di malfunzionamento della porta 2 (LED rosso)
Broken_3	O	1	Segnale di malfunzionamento della porta 3 (LED rosso)
Broken_4	O	1	Segnale di malfunzionamento della porta 4 (LED rosso)
Broken_5	O	1	Segnale di malfunzionamento della porta 5 (LED rosso)
Err_1	O	1	Segnale di errore comunicazione CAN tipo 1 (LED rosso)
Err_2	O	1	Segnale di errore comunicazione CAN tipo 2 (LED rosso)
Err_3	O	1	Segnale di errore comunicazione CAN tipo 3 (LED rosso)
Err_ser	O	1	Segnale di errore comunicazione seriale (LED rosso)

Tabella 3.1.3-1 Input e output di COM\_PIC

Le funzioni di questo blocco sono relative alla comunicazione seriale con il PIC Master:

- Inviare il carattere di alive “FF” ogni 2 secondi;
- Inviare i dati per permettere all’utente di entrare attraverso un protocollo dedicato;
- Identificazione dei messaggi ricevuti;
- Controllare l’arrivo del segnale di alive;
- Segnalare la presenza di una porta aperta che rende impossibile accedere;
- Segnalazione all’utente di eventuali guasti o allarmi nell’impianto attraverso l’accensione di LED

Di seguito (Fig. 3.1.3-2) il diagramma a blocchi del blocco COM\_PIC:

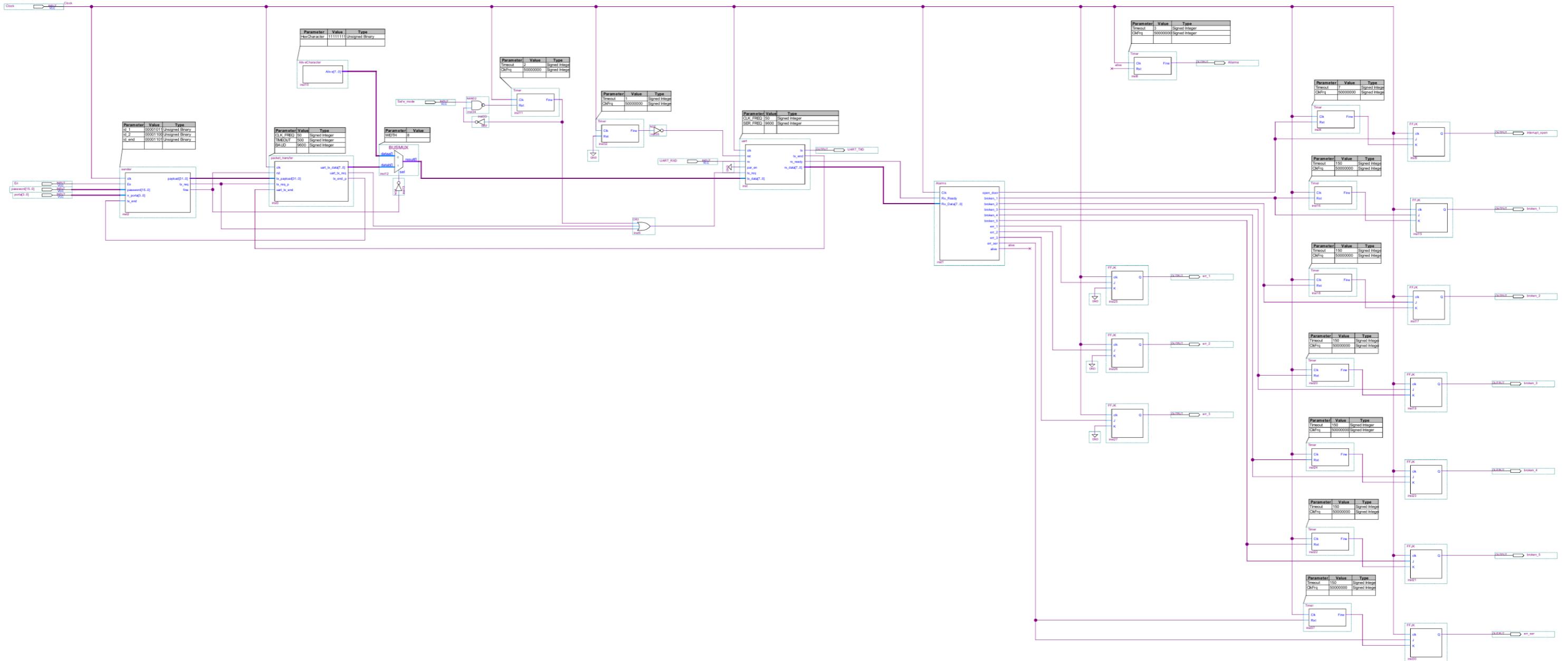


Fig. 3.1.3-2 Diagramma a blocchi di COM\_PIC

La prima funzione è l'invio del carattere di alive “FF” ogni 2 secondi: il blocco UART viene usato sia per l'invio del segnale di alive che per l'invio dei dati, perciò è stato usato un blocco BUSMUX che determina qual è la fonte dei dati di “tx\_data”. Il BUSMUX è normalmente settato sulla posizione zero, cioè il carattere di alive. L'ingresso “tx\_req” del blocco UART viene controllato da un timer settato a 2 secondi, che viene resettato dalla sua stessa uscita garantendo un funzionamento perpetuo. L'accensione del timer è vincolata anche all'assenza del Raspberry, quindi l'invio dell'alive avviene solo se il Raspberry è scollegato; questo serve a comunicare al PIC che il Raspberry è tornato in funzione, permettendogli di switchare dalla connessione con l'FPGA a quella con il Raspberry.

Come anticipato, un'altra funzione è l'invio dei dati per l'accesso: dopo che l'utente ha completato le operazioni di inserimento dati, se tutto è risultato corretto, il blocco Safe\_mode invierà il segnale di “Fine\_ok”, qui ricevuto in ingresso come “En”. Il blocco Sender compone i pacchetti dati e li trasmette al packet transfer che a sua volta si occupa del loro invio attraverso il pin “TX”. Il BUSMUX selettore dei dati per “tx\_data” viene spostato nella posizione 1 dal complementare del segnale “fine” del blocco Sender; tale segnale è sempre alto tranne quando il Sender sta effettuando delle operazioni atte all'invio dei pacchetti; essendo poi negato, il selettore è sul canale 1 solo in questo caso. Non è stato previsto un controllo dell'effettiva ricezione dei dati per semplicità progettuale e perché, nel caso ci fossero errori gravi, il controllo dell'alive lo mostrerebbe.

L'altra importante funzione svolta dal blocco Com\_PIC è la ricezione e identificazione dei segnali in arrivo: i messaggi in arrivo dal PIC Master sono solo byte singoli, che possono avere diversi significati interpretati dal blocco Alarms. Gli errori vengono gestiti in due modi a seconda della loro natura:

1. Per gli errori temporanei, si applica la tecnica già vista per il Raspberry nel *paragrafo 2.1*. Il segnale in uscita dal blocco Alarms attiva un flip flop JK (che è direttamente connesso all'uscita del blocco, quindi questo segnale è ciò che vede l'utente tramite LED) e allo stesso tempo un timer (di tempo variabile a seconda dell'errore), lo scatto del timer è collegato all'ingresso K del flip flop, quindi abbassa l'uscita, causando lo spegnimento del LED.
2. Per gli errori più gravi, il LED viene lasciato acceso fino a un reset del sistema. In questo caso il flip flop JK non è resettato da un timer.

Gli errori che si possono ricevere sono:

- Segnalazione di porta aperta: in questa condizione è necessario che il sistema di accesso venga bloccato, il segnale corrispondente in output infatti va a disattivare la modalità ridotta, presentando all'utente lo schermo LCD spento e impendendo qualsiasi azione; viene anche acceso il LED corrispondente a tale condizione, il suo timer è impostato a 7 secondi.
- Porte fuori uso: gli errori dovuti alle porte fuori uso sono semplicemente indicati tramite LED e l'utente deve evitare, in autonomia, di selezionare una delle porte fuori uso. (Un possibile sviluppo futuro sarà quello di segnalarlo tramite LCD e mostrare un errore se viene selezionata tale porta.)
- Errori CAN: anche gli errori più gravi, che riguardano il CAN, sono mostrati tramite LED.
- Errore seriale: indica che il messaggio non è stato inviato correttamente, ciò rende impossibile accedere all'impianto, il timer corrispondente è impostato a 150 secondi. Se questo errore capita spesso l'utente dovrà richiedere il supporto tecnico. (Una possibile risoluzione futura di questo errore è quella di ritentare l'invio del messaggio al PIC.)

Per il caso in cui non sia stato ricevuto il segnale di alive dal PIC per oltre 3 secondi, si accende un LED corrispondente. Non appena si riceve di nuovo un carattere di alive il segnale “Allarme” si resetta e il LED si spegne.

In Tabella 3.1.3-2 il sistema di codifica dei messaggi ricevuti dal PIC:

Messaggio di errore o segnale ricevuto	Codice errore (hex)	Conseguenza
Mancanza di alive per oltre 3 secondi	FF	Accensione LED rosso 1 fino alla ricezione di un carattere di alive corretto
Segnale di porta aperta	DD	Disattivazione dello schermo LCD, accensione LED rosso 2 per 7 secondi*
Segnale di malfunzionamento della porta 1	D1	Accensione LED rosso 7 per 150 secondi**
Segnale di malfunzionamento della porta 2	D2	Accensione LED rosso 8 per 150 secondi**
Segnale di malfunzionamento della porta 3	D3	Accensione LED rosso 9 per 150 secondi**
Segnale di malfunzionamento della porta 4	D4	Accensione LED rosso 10 per 150 secondi**
Segnale di malfunzionamento della porta principale	D5	Accensione LED rosso 11 per 150 secondi**
Segnale di errore comunicazione CAN tipo 1	E1	Accensione LED rosso 16 fino al reset manuale del sistema
Segnale di errore comunicazione CAN tipo 2	E2	Accensione LED rosso 17 fino al reset manuale del sistema
Segnale di errore comunicazione CAN tipo 3	E3	Accensione LED rosso 18 fino al reset manuale del sistema
Segnale di errore comunicazione seriale	EE	Accensione LED rosso 14 per 150 secondi

Tabella 3.1.3-2 Messaggi di segnalazione dal PIC

\*Il messaggio relativo all'apertura di una porta viene trasmesso ogni 6 secondi, perciò il timer è settato a 7 secondi.

\*\*Il messaggio relativo alle porte fuori uso viene inviato ogni 2 minuti circa perciò il timer per questi LED è settato a 150 secondi.

### 3.1.3.1 Sender

In Fig. 3.1.3.1-1 il simbolo di Sender:

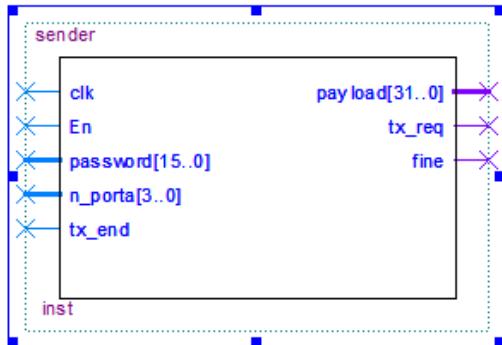


Fig. 3.1.3.1-1 Simbolo di Sender

**Porte:** in Tabella 3.1.3.1-1 l'elenco degli input e output di Sender.

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
En	I	1	Segnale di attivazione del blocco
password	O	16	Password che deve essere inviata al PIC
N_porta	O	4	Numero porta che deve essere inviato al PIC
Tx_end	O	1	Segnale di fine trasmissione
Payload	O	32	Payload di 4 byte per il blocco packet transfer
Tx_req	O	1	Segnale di richiesta inizio trasmissione
Fine	O	1	Segnale di fine operazione

Tabella 3.1.3.1-1 Input e output di Sender

Questo blocco forma i pacchetti di dati da inviare al PIC attraverso il blocco packet transfer.

Riprendendo ciò che è già stato detto nel paragrafo 2.3, questo (Tabella 3.1.3.1-2) è il protocollo di trasmissione verso il PIC Master, forma lo stesso messaggio che formava il Raspberry perché il PIC deve lavorare allo stesso modo.

Primo pacchetto:	Secondo pacchetto:	
AA	AA	Header
55	55	
0B	0C	Payload
0000 + numero porta (4 bit)	0000 + terza cifra id	
0000 + prima cifra id	0000 + quarta cifra id	
0000 + seconda cifra id	0D	

Tabella 3.1.3.1-2 codifica messaggio di apertura porta

**Macchina a stati:**

- **IDLE:**

I segnali interi "data1" e "data2" sono azzerati.

Se viene ricevuto il segnale di enable, vengono formati i due payload con i dati in ingresso, essi sono gli ultimi 4 byte di ogni pacchetto. Viene azzerato il segnale di fine e viene aggiornato lo stato a SEND.

- **SEND:**

Viene alzato il segnale “tx\_req” che è connesso al “tx\_req” del packet transfer e indica che è possibile inviare il dato. Il payload in uscita assume il valore di uno dei pacchetti, a seconda che il contatore a 1 bit count sia a zero o a uno, quindi se la macchina a stati si trova all’invio del primo pacchetto o del secondo. Lo stato cambia in HOLD.

- **HOLD:**

Viene resettato “tx\_req”. Appena si riceve il segnale “tx\_end” dal packet transfer, che indica che la trasmissione del pacchetto è terminata, se è terminato il primo invio si torna allo stato SEND, se invece è stato completato il secondo invio si torna in IDLE e viene aggiornato il contatore.

In Fig. 3.1.3.1-2 il diagramma di flusso della macchina a stati di Sender.

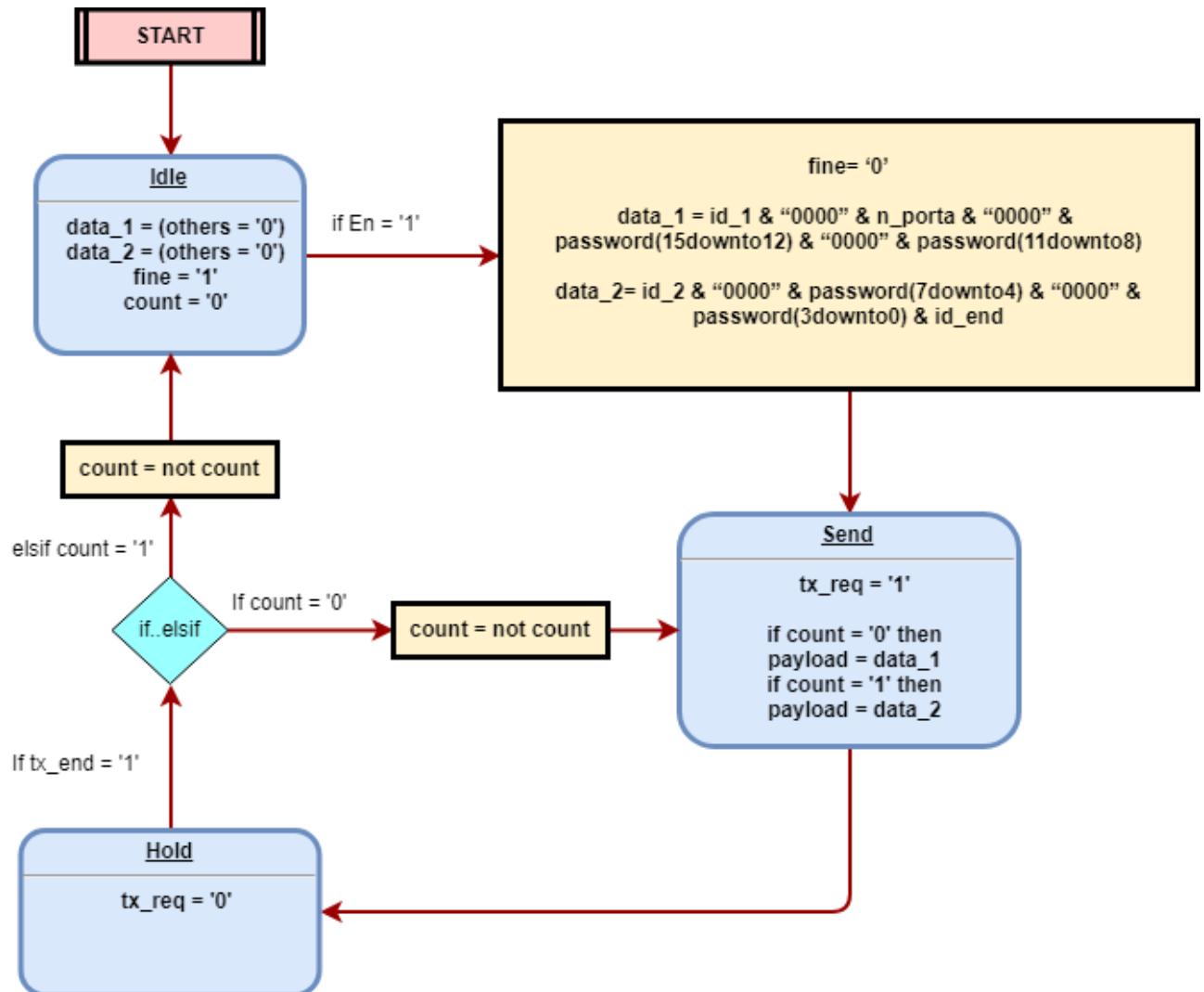


Fig. 3.1.3.1-2 Diagramma di flusso di Sender

## Codice VHDL:

```
library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sender is
generic (
    id_1      : STD_LOGIC_VECTOR(7 downto 0):= X"11";          -- 0B primo identificativo
    id_2      : STD_LOGIC_VECTOR(7 downto 0):= X"12";          -- 0C secondo identificativo
    id_end   : STD_LOGIC_VECTOR(7 downto 0) := X"13";          -- 0D terzo identificativo

Port(
    clk        : in STD_LOGIC;
    En         : in STD_LOGIC;                                -- alto se è richiesto l'inizio dell'invio
    Password   : in STD_LOGIC_VECTOR(15 downto 0);-- password di 4 cifre inserita dall'utente
    n_porta    : in STD_LOGIC_VECTOR(3 downto 0); -- numero porta inserito dall'utente
    tx_end     : in STD_LOGIC;                                -- sgn di fine trasmissione dal blocco packet transfer
    payload    : out STD_LOGIC_VECTOR(31 downto 0);
    tx_req     : out STD_LOGIC:= '0';                      -- sgn che attiva la trasmissione del packet transfer
    fine       : out STD_LOGIC:= '0');                      -- alto quando il sender ha completato le operazioni
end sender;

architecture behave of sender is
type state_type is (idle, send, hold);
signal data_1, data_2 : STD_LOGIC_VECTOR(31 downto 0);
signal state         : state_type := idle;
signal count         : STD_LOGIC:= '0';

begin
process(clk, en, tx_end)
begin
    if (clk'event and clk = '1') then
        case state is
            when idle =>
                data_1 <= (others=>'0');
                data_2 <= (others=>'0');
                fine <= '1';
                count <= '0';
                if en = '1' then
                    state <= send;
                    fine <= '0';
                    data_1 <= id_1 & "0000" & n_porta & "0000" & password(15
downto 12)& "0000" & password(11 downto 8);
                    data_2 <= id_2 & "0000" & password(7 downto 4) & "0000" &
password(3 downto 0) & id_end;
                end if;
            when send =>
                tx_req <= '1';
                if count = '0' then
                    payload <= data_1;
                elsif count = '1' then

```

```

        payload <= data_2;
    end if;
    state <= hold;
when hold =>
    tx_req <= '0';
    if tx_end = '1' then
        if count = '0' then
            state <= send;
        elsif count = '1' then
            state <= idle;
        end if;
        count <= not count;
    end if;
end case;
end if;
end process;
end behave;

```

### 3.1.3.2 Alarms

In Fig.3.1.3.2-1 il simbolo di Alarms.

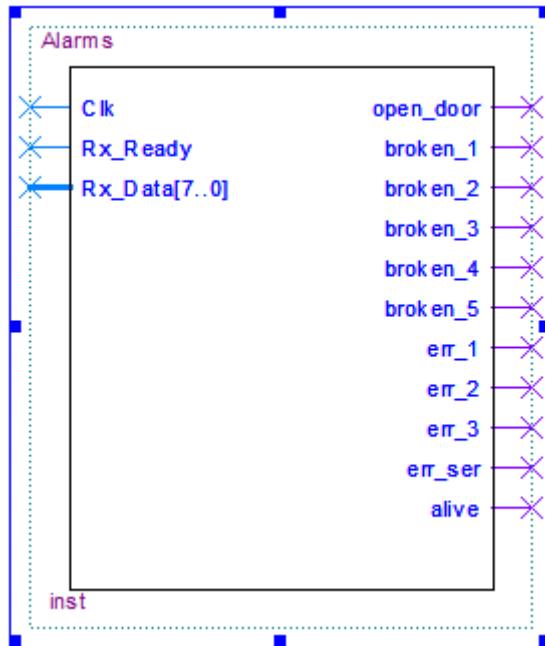


Fig. 3.1.3.2-1 Simbolo di Alarms

**Porte:** in Tabella 3.1.3.2-1 gli input e output di Alarms.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
Rx_Ready	I	1	Segnale di ricezione di un messaggio seriale
RX_Data	I	1	Dati in arrivo dal blocco UART
Open_door	O	1	Segnale di porta aperta
Broken_1	O	1	Segnale di malfunzionamento della porta 1
Broken_2	O	1	Segnale di malfunzionamento della porta 2
Broken_3	O	1	Segnale di malfunzionamento della porta 3
Broken_4	O	1	Segnale di malfunzionamento della porta 4
Broken_5	O	1	Segnale di malfunzionamento della porta principale
Err_1	O	1	Segnale di errore comunicazione CAN tipo 1
Err_2	O	1	Segnale di errore comunicazione CAN tipo 2
Err_3	O	1	Segnale di errore comunicazione CAN tipo 3
Err_ser	O	1	Segnale di errore comunicazione seriale
alive	O	1	Segnale di alive

Tabella 3.1.3.2-1 Input e output di Alarms

Questo blocco controlla i messaggi ricevuti dal PIC, che è programmato per inviare messaggi di avviso per vari eventi.

Per ogni errore il corrispondente segnale di uscita viene settato alto per un ciclo di clock, questi segnali saranno poi elaborati dagli altri componenti del blocco COM\_PIC.

**Codice VHDL:**

```

library ieee;
use ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Alarms is
    Port(
        Clk : in STD_LOGIC;
        Rx_Ready : in STD_LOGIC;
        Rx_Data : in STD_LOGIC_VECTOR (7 downto 0);
        open_door : out STD_LOGIC;
        broken_1 : out STD_LOGIC;
        broken_2 : out STD_LOGIC;
        broken_3 : out STD_LOGIC;
        broken_4 : out STD_LOGIC;
        broken_5 : out STD_LOGIC;
        err_1 : out STD_LOGIC;
        err_2 : out STD_LOGIC;
        err_3 : out STD_LOGIC;
        err_ser : out STD_LOGIC;
        alive : out STD_LOGIC
    );
-- Clock
-- Rx_Ready
-- Rx_Data
-- sgn di porta aperta
-- sgn di porta 1 fuori servizio
-- sgn di porta 2 fuori servizio
-- sgn di porta 3 fuori servizio
-- sgn di porta 4 fuori servizio
-- sgn di porta princ. fuori servizio
-- sgn di errore CAN di tipo 1
-- sgn di errore CAN di tipo 2
-- sgn di errore CAN di tipo 3
-- sgn di errore seriale
-- sgn di alive

end Alarms;

architecture behave of Alarms is
begin
process(Clk, Rx_Ready) is
begin
    if (Clk'event and Clk = '1') then
        open_door <= '0';
        broken_1 <= '0';
        broken_2 <= '0';
        broken_3 <= '0';
        broken_4 <= '0';
        broken_5 <= '0';
        err_1 <= '0';
        err_2 <= '0';
        err_3 <= '0';
        err_ser <= '0';
        alive <= '0';

        if Rx_Ready = '1' then
            if rx_Data = "11011101" then
                open_door <= '1'; -- porta aperta X"DD"
            elsif rx_Data = "11010001" then
                broken_1 <= '1'; -- porta 1 fuori uso X"D1"
            elsif rx_Data = "11010010" then
                broken_2 <= '1'; -- porta 2 fuori uso X"D2"
            elsif rx_Data = "11010011" then
                broken_3 <= '1'; -- porta 3 fuori uso X"D3"
            end if;
        end if;
    end if;
end process;

```

```

    elsif rx_Data = "11010100" then          -- porta 4 fuor uso X"D4"
        broken_4 <= '1';
    elsif rx_Data = "11010101" then          -- porta 5 rotta X"D5"
        broken_5 <= '1';
    elsif rx_Data = "11100001" then          -- errore CAN tipo 1 X"E1"
        err_1 <= '1';
    elsif rx_Data = "11100010" then          -- errore CAN tipo 2 X"E2"
        err_2 <= '1';
    elsif rx_Data = "11100011" then          -- errore CAN tipo 3 X"E3"
        err_3 <= '1';
    elsif rx_Data = "11101110" then          -- errore seriale X"EE"
        err_ser <= '1';
    elsif rx_Data = "11111111" then          -- alive X"FF"
        alive <= '1';
    end if;
end if;
end process;
end behave;

```

### 3.1.4 Tastierino

In Fig. 3.1.4-1 il simbolo di Tastierino.

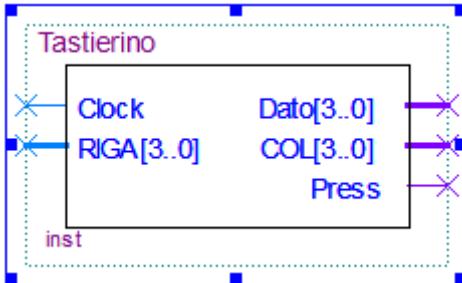


Fig. 3.1.4-1 Simbolo di Tastierino

**Porte:** in Tabella 3.1.4-1 gli input e output di Tastierino.

Pin	I/O	bit	Descrizione
Clock	I	1	Clock
RIGA	I	4	Attivazione del blocco
Dato	O	4	Cifra premuta sul tastierino
COL	O	4	Segnale di pressione sul tastierino
Press	O	1	Password aggiornata in tempo reale, destinata al display

Tabella 3.1.4-1 Input e output di Tastierino

Questo blocco ha le porte di output e input collegate con i pin general purpose ai terminali del tastierino, si occupa di determinare quando un utente preme un tasto, quando si verifica questo evento il blocco tastierino trasmette in uscita il valore del tasto premuto e il segnale “Press” della durata di un ciclo di clock per segnalare la pressione del tasto.

Il blocco counter è un contatore a 4 bit, che indicheremo con (YYXX), che permette di scorrere tutti i 16 tasti del tastierino (il cui schema si può osservare in Fig.3.1.4-2 e 3.1.4-3). Non utilizza la frequenza del clock ma usa un suo sottomultiplo; il blocco ck2ck è il prescaler che gli offre il clock rallentato.

I due bit meno significativi del contatore (XX) effettuano la scansione delle righe; nel blocco dec2\_4 queste due cifre sono il selettore dell'uscita: l'ingresso "00" corrisponde alla riga "0001" per cui il segnale di Vcc viene direzionato sulla riga 1 e così via. I due bit più significativi (YY) effettuano lo scorrimento delle colonne; il blocco mux4\_1 presenta in ingresso i 4 terminali delle colonne e una sola uscita, i due bit del contatore selezionano quale ingresso collegare all'uscita: se il selettore è "00" viene portato in output il valore di tensione della colonna 1.

Perciò i quattro bit del contatore possono essere identificati come (Col1, Col0, Rig1, Rig0).

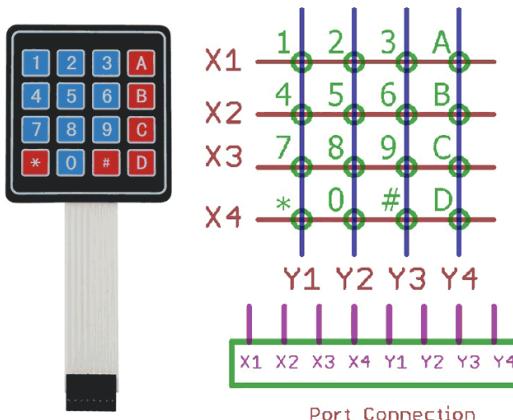


Fig. 3.1.4-2 Schema elettronico del keypad

Posizione tasti					Riga (Rig1, Rig0)
	1	2	3	A	11
	4	5	6	B	10
	7	8	9	C	01
	*	0	#	D	00
Colonna (Col1, Col0)	11	10	01	00	

Fig. 3.1.4-3 Schema tasti del tastierino e numeri di righe e colonne

Quando l'uscita del blocco mux4\_1 è alta significa che il contatore ha individuato il tasto che è stato premuto perché il segnale di alta tensione inviato nella riga XX è stato ricevuto sulla colonna YY e questo valore del contatore deve essere salvato. L'uscita viene, quindi, utilizzata per fermare il contatore e diventa il segnale di enable per il registro che salva il dato. Subito dopo il valore salvato viene convertito in BCD.

La tabella per la conversione counter – binario è la seguente (Tabella 3.1.4-2):

Tastierino	Counter	Binario
1	1111	0001
2	1011	0010
3	0111	0011
A	0011	1010
4	1110	0100
5	1010	0101
6	0110	0110
B	0010	1011
7	1101	0111
8	1001	1000
9	0101	1001
C	0001	1100
*	1100	1110
0	1000	0000
#	0100	1111
D	0000	1101

Tabella 3.1.4-2 Conversione tra indicatore counter e rappresentazione binaria del numero premuto

In pratica, si converte il simbolo sul tastierino nella sua rappresentazione esadecimale. I simboli \* e # prendono il posto di E e F in esadecimale, i quali mancano sul tastierino.

Per quanto riguarda il segnale di “press”, il debouncer e il transition finder sono necessari, il primo per escludere le oscillazioni dovute alla pressione non istantanea del tasto (perché anche se la velocità di scansione è inferiore a quella del clock, i tempi umani di pressione del tasto sono comunque molto più lunghi e il tasto risulta essere premuto per moltissime volte) e il secondo per avere un impulso che duri un ciclo di clock.

Infine, il blocco di delay consente di sincronizzare il segnale di “press” con il registro in modo che il registro sia aggiornato quando il segnale di “press” è alto, ciò è molto importante per i blocchi che utilizzeranno il tastierino.

In Fig. 3.1.4-4 lo schema a blocchi di Tastierino.

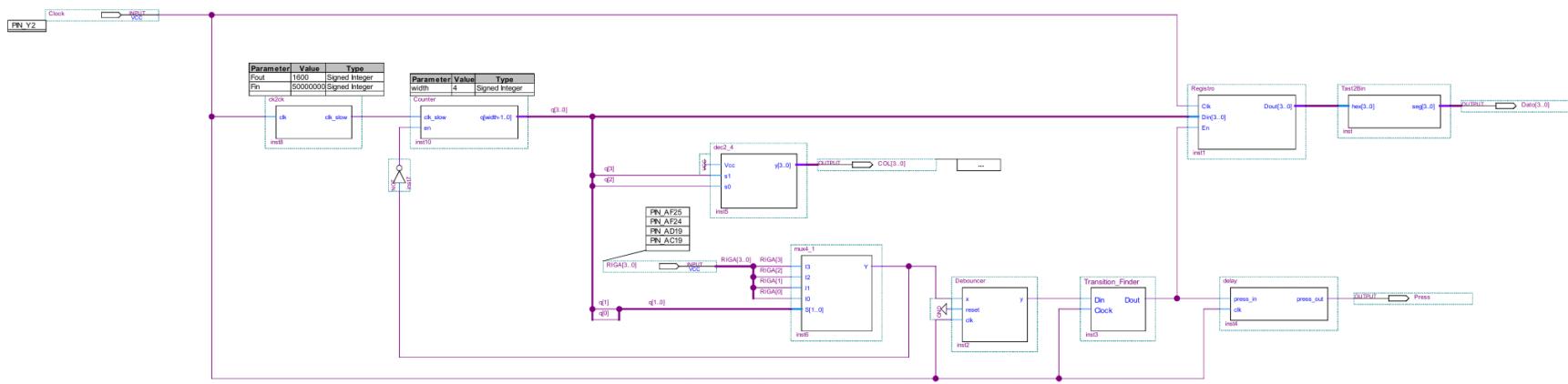


Fig. 3.1.4-4 Diagramma a blocchi di Tastierino

## 3.2 Altri blocchi

Di seguito verranno descritti nel dettaglio tutti i blocchi che in precedenza sono stati solamente citati e descritti velocemente, sono riportati in ordine alfabetico.

### 3.2.1 AliveCharacter

In Fig. 3.2.1-1 il simbolo di Alive Character:

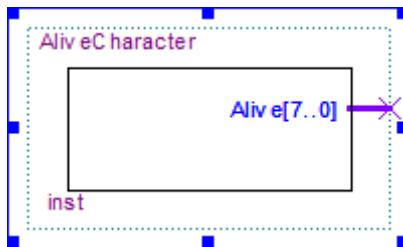


Fig. 3.2.1-1 Simbolo di AliveCharacter

**Porte:** in Tabella 3.2.1-1 gli input e output di AliveCharacter.

Pin	I/O	bit	Descrizione
Alive	O	8	Carattere di alive

Tabella 3.2.1-1 Input e output di AliveCharacter

Questo blocco serve a definire una costante di 1 byte, che sarà utilizzato come carattere di alive.

**Carattere di alive:** in Tabella 3.2.1-2.

Hex	Bin
FF	11111111

Tabella 3.2.1-2 carattere di alive

**Codice VHDL:**

```
library ieee;
use ieee.std_logic_1164.ALL;
-- Questo blocco definisce una costante di 8 bit

entity AliveCharacter is
generic (
    HexCharacter: std_logic_vector(7 downto 0) := X"ff"; -- Carattere X"55" = "01010101"
Port (
    Alive : out std_logic_vector(7 downto 0));
end AliveCharacter;

architecture dflow of AliveCharacter is
begin
    Alive <= HexCharacter;
end dflow;
```

### 3.2.2 Check alive

In Fig. 3.2.2-1 il simbolo di Check alive:

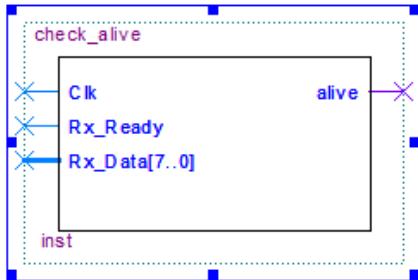


Fig. 3.2.2-1 Simbolo di Check\_Alive

**Porte:** in Tabella 3.2.2-1 gli input e output di Check alive.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
Rx_Ready	I	1	Segnale di ricezione di un messaggio seriale
Rx_Data	I	1	Dati in arrivo dal blocco UART
Alive	O	1	Segnale di alive

Tabella 3.2.2-1 Input e output di Check\_alive

Questo blocco controlla i messaggi ricevuti dal Raspberry e riconosce se si tratta di un messaggio di alive, Quando viene ricevuto un segnale di alive CC (1100 1100) l'uscita viene alzata per un ciclo di clock. Qualsiasi altro segnale viene ignorato.

**Codice VHDL:**

```

library ieee;
use ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity check_alive is
    Port(
        Clk : in STD_LOGIC;                                -- Clock
        Rx_Ready : in STD_LOGIC;                            -- Rx_Ready
        Rx_Data : in STD_LOGIC_VECTOR (7 downto 0);        -- Rx_Data
        alive : out STD_LOGIC);                            -- sgn di alive
end check_alive;

architecture behave of check_alive is
begin
process(Clk, Rx_Ready) is
begin
    if (Clk'event and Clk = '1') then
        alive <= '0';
        if Rx_Ready = '1' then
            if rx_Data = "11001100" then                -- alive X"CC"
                alive <= '1';
            end if;
        end if;
    end if;
end process;
end behave;

```

### 3.2.3 Ck2ck

In Fig. 3.2.3-1 il simbolo di ck2ck:

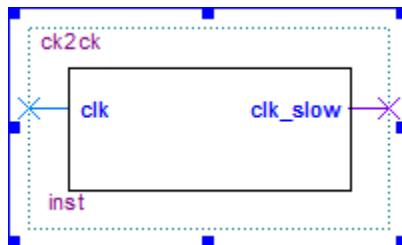


Fig. 3.2.3.1 Simbolo di ck2ck

**Porte:** in Tabella 3.2.3-1 gli input e output di Ck2ck.

Pin	I/O	bit	Descrizione
clk	I	1	Clock
clk_slow	O	1	Uscita di clock rallentata

Tabella 3.2.3-1 Input e output di ck2ck

Lo scopo di questo blocco è di generare un clock, partendo da quello dell'oscillatore, che oscilla a frequenza inferiore. In questo caso, è stato utilizzato per generare una frequenza di 1,6 kHz, ma è possibile modificarlo per qualsiasi frequenza con il vincolo che il rapporto tra la frequenza dell'oscillatore e quella in uscita sia in numero pari, in caso contrario, il funzionamento del blocco non è corretto.

Il blocco ha il comportamento di un contatore: partendo da un valore logico di uscita pari a 0 inizia a contare i cicli di clock fino a raggiungere il numero cui corrisponde la metà del periodo desiderato in uscita, quindi inverte l'uscita (toggle del segnale) e azzera il contatore.

**Codice VHDL:**

- Crea un clock a frequenza inferiore
- Il blocco è simile a un prescaler ma in uscita è un segnale di clock, non di impulsi.
- Il clock inizia sempre con il semiperiodo in cui il valore è 0.

*library IEEE;*

*use IEEE.std\_logic\_1164.all;*

*use IEEE.numeric\_std.all;*

*use IEEE.std\_logic\_unsigned.all;*

-- Si sceglie di utilizzare per il tastierino una velocità di sondaggio del tastierino molto inferiore  
-- a quella del clock, pari a 1,6 kHz.

```
entity ck2ck is
    generic (
        -- Il rapporto tra Fin e Fout DEVE essere un numero pari!
        Fout : integer := 1600; -- Frequenza desiderata in uscita
        Fin : integer := 50000000); -- Frequenza dell'oscillatore
    Port (
        clk : in std_logic; -- Clock
        clk_slow : out std_logic); -- Clock rallentato
end ck2ck;
```

*architecture beh of ck2ck is*

*signal count : integer := 0;*  
*signal state : std\_logic := '0';*

```
begin
process(clk) is
begin
    if (clk = '1' and clk'event) then
        count <= count + 1;
        if (count = (Fin /(2*Fout)) )then
            state <= not state;
            count <= 1;
        end if;
    end if;
end process;
clk_slow <= state;
end architecture beh;
```

### 3.2.4 Counter

In Fig. 3.2.4-1 il simbolo di Counter:

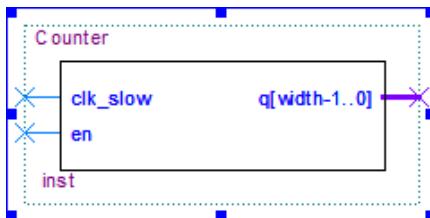


Fig. 3.2.4-1 Simbolo di Counter

**Porte:** in Tabella 3.2.4-1 gli input e output di Counter.

Pin	I/O	bit	Descrizione
clk_slow	I	1	Clock rallentato da un prescaler
en	I	1	Enable, quando è (1), il blocco conta ogni ciclo del clk_slow
q	O	4	Valore attuale del contatore

Tabella 3.2.4-1 Input e output di Counter

Questo blocco ha lo scopo di contare il numero dei cicli di clock in ingresso. È previsto che lavori con un ingresso di clock non alla frequenza dell'oscillatore (50 MHz), ma può lavorare in generale con qualsiasi frequenza di clock. Si può selezionare il numero di bit del contatore; in questo caso si utilizzano 4 bit e dopo aver contato fino a 15 l'uscita va in overflow e il contatore ricomincia da 0. Il segnale di enable "en" abilita il funzionamento del blocco quando è a 1. Se l'enable si porta a 0, il processo si ferma per poi ripartire dallo stesso punto quando l'enable torna a 1.

**Codice Vhdl:**

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Counter is
    generic(
        width : integer := 4); -- Numero di bit del contatore
    Port (
        clk_slow : in std_logic; -- Clock Rallentato
        en : in std_logic; -- Enable
        q : out std_logic_vector (width-1 downto 0)); -- Uscita
end Counter;

architecture beh of Counter is
signal temp : std_logic_vector (width-1 downto 0);
begin
    q <= temp;
    process(clk_slow) -- Counter process
    begin
        if (clk_slow'event and clk_slow = '1') then
            if (en = '1') then
                temp <= temp + 1;
            end if;
        end if;
    end process;
end beh;
```

### 3.2.5 Debouncer

In Fig. 3.2.5-1 il simbolo di Debouncer:

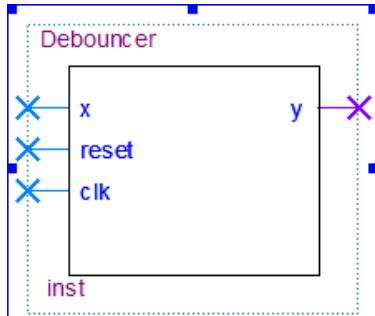


Fig. 3.2.5-1 Simbolo di Debouncer

**Porte:** in Tabella 3.2.5-1 gli input e output di Debouncer.

Pin	I/O	bit	Descrizione
x	I	1	Ingresso
reset	I	1	Reset
clk	I	1	Clock
y	O	1	Uscita

Tabella 3.2.5-1 Input e output di Debouncer

Il debouncer è un blocco che serve a risolvere il problema che si può presentare in presenza di transizioni spurie (rimbalzi) durante la pressione o il rilascio di un pulsante del tastierino. Il pulsanti non sono ideali ma, essendo elettromeccanici, sono soggetti a disturbi. Quando si preme il pulsante, il segnale di chiusura (Fig. 3.2.5-2), prima di stabilizzarsi, rimbalza un po' a causa del contatto (bounce, da cui il nome del blocco). Stessa cosa per il segnale di apertura quando smetto di premere il pulsante. Il debouncer è una rete che permette di eliminare questo disturbo. Il blocco è già fornito dal prof. Santinelli e, non essendo stato modificato, non verrà trattato ulteriormente.

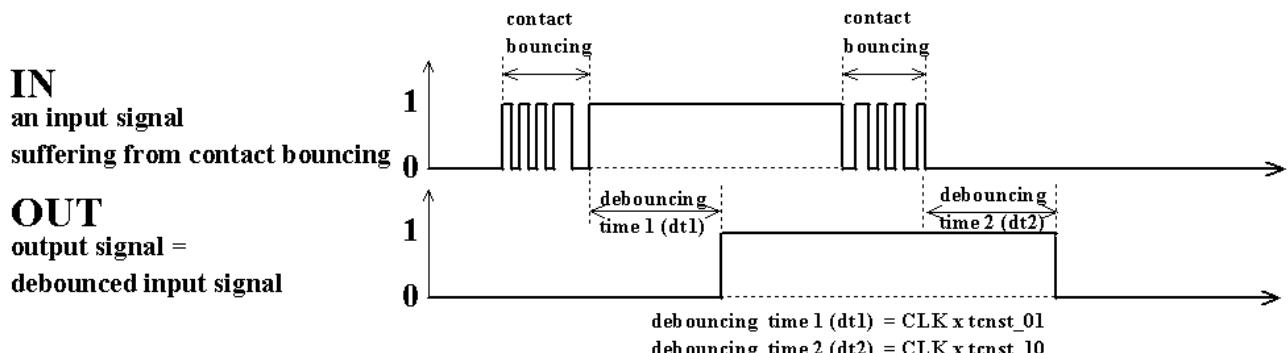


Fig. 3.2.5-2 Risposta nel tempo di Debouncer

**Codice VHDL:**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```

entity Debouncer is
  Port (
    X      : in   STD_LOGIC;
    y      : out  STD_LOGIC;
    reset  : in   STD_LOGIC;
    clk    : in   STD_LOGIC);
end Debouncer;

architecture Behavioral of Debouncer is
  -- output del registro a scorrimento
  signal finestra : STD_LOGIC_VECTOR (7 downto 0);
  signal set : STD_LOGIC;
  signal rst : STD_LOGIC;
  signal n_en_scorr: STD_LOGIC;
  -- definizione dei componenti usati internamente
  component RegistroScorrimento is
    Port (          x      : in   STD_LOGIC;
                    n_en   : in   STD_LOGIC;
                    clk    : in   STD_LOGIC;
                    y      : out  STD_LOGIC_VECTOR (7 downto 0));
  end component;
  component FF_SR is
    Port (          S      : in   STD_LOGIC;
                    R      : in   STD_LOGIC;
                    clk   : in   STD_LOGIC;
                    Q      : out  STD_LOGIC;
                    QN    : out  STD_LOGIC);
  end component;
  component prescaler is
    generic(      CLK_FREQ: natural := 100;           -- Main frequency      MHz (system clock)
                  OUT_FREQ: natural := 200);           -- Ouput frequency     Hz
    Port (          reset  : in   STD_LOGIC;
                    clk    : in   STD_LOGIC;
                    n_rc   : out  STD_LOGIC);
  end component;
begin
  Pres1:prescaler
    generic map(CLK_FREQ => 100, OUT_FREQ => 500)
    Port map(clk => clk, n_rc => n_en_scorr, reset => reset );

  -- collegamento del registro a scorrimento
  Reg1: RegistroScorrimento Port map (x => x, n_en => n_en_scorr, clk => clk, y => finestra);

  -- generazione di set e reset a partire dalle uscite del registro
  set <= finestra(0) and finestra(1) and finestra(2) and finestra(3) and finestra(4) and finestra(5) and
  finestra(6) and finestra(7);
  rst <= not(finestra(0) or finestra(1)or finestra(2) or finestra(3) or finestra(4) or finestra(5)or
  finestra(6) or finestra(7));

  -- flip flip SR per generare l'uscita coerente con un ingresso stabile
  FFSR1 : FF_SR Port map (S => set, R => rst, clk => clk, Q => y);
end Behavioral;

```

### 3.2.6 Dec2\_4

In Fig. 3.2.6-1 il simbolo di Dec2\_4:

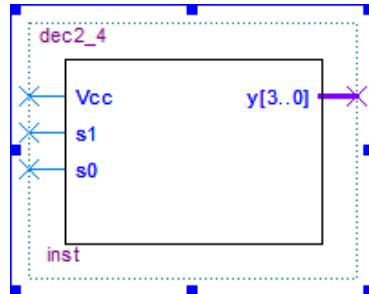


Fig. 3.2.6.1 Simbolo di Dec2\_4

**Porte:** in Tabella 3.2.1-1 gli input e output di Dec2\_4.

Pin	I/O	bit	Descrizione
Vcc	I	1	Da collegare a VCC, 1 logico
s1	I	1	Ingresso di selezione, bit più significativo
s0	I	1	Ingresso di selezione, bit meno significativo
y	O	4	Uscite

Tabella 3.2.6-1 Input e output di Dec2\_4

Questo decoder serve per smistare l'input "Vcc" (che ha valore logico 1) in una delle 4 possibili uscite del vettore "y" in funzione dei 2 bit in ingresso di selezione. La tabella di verità è la seguente (Tabella 3.2.6-2):

s1	s0	Vcc -> y
0	0	y0
0	1	y1
1	0	y2
1	1	y3

Tabella 3.2.6-2 Tabella di verità di Dec2\_4

**Codice VHDL:**

```

library ieee;
use ieee.std_logic_1164.all;

entity dec2_4 is
  Port (
    Vcc      : in std_logic;          -- Input da connettere a Vcc
    s1       : in std_logic;          -- Selettore 1
    s0       : in std_logic;          -- Selettore 0
    y        : out std_logic_vector(3 downto 0)); -- Uscite
end dec2_4;
architecture Behavioral of dec2_4 is
  signal ens: std_logic_vector(2 downto 0);
begin
  ens <= Vcc & s1 & s0;
  with ens select
    y <= "0001" when "100",
                "0010" when "101",
                "0100" when "110",
                "1000" when "111",
                "0000" when others;
end Behavioral;

```

### 3.2.7 Delay

In Fig. 3.2.7-1 il simbolo di Delay:

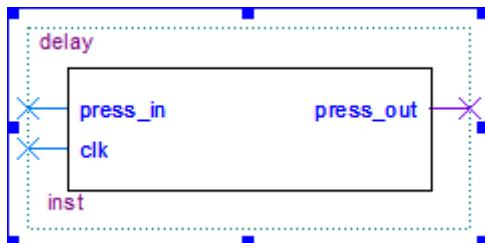


Fig. 3.2.7-1 Simbolo di Delay

**Porte:** in Tabella 3.2.7-1 gli input e output di Delay.

Pin	I/O	bit	Descrizione
press_in	I	1	Segnale in ingresso
clk	I	1	Clock
press_out	O	1	Segnale di uscita, segnale di ingresso ritardato

Tabella 3.2.7-1 Input e output di Delay

Lo scopo di questo blocco è quello di fornire un segnale di uscita in ritardo di un ciclo di clock rispetto al segnale in ingresso. Il ritardo è generato da un *if* che controlla ogni ciclo di clock il valore dell'ingresso e lo assegna all'uscita nel ciclo di clock successivo.

**Codice VHDL:**

```

library IEEE;
use IEEE.std_logic_1164.all;

-- Questo blocco serve per portare l'enable al ciclo successivo per essere sincronizzato con il dato sul Registro

entity delay is
Port(
    press_in      : in std_logic;                      -- Input
    clk           : in std_logic;                      -- Clock
    press_out     : out std_logic := '0');              -- Output
end delay;

architecture beh of delay is
begin
process(press_in)
begin
    if (clk'event and clk = '1') then
        if press_in = '1' then
            press_out <= '1';
        elsif press_in = '0' then
            press_out <= '0';
        end if;
    end if;
end process;
end beh;
```

### 3.2.8 FFJK

In Fig. 3.2.8-1 il simbolo di FFJK:

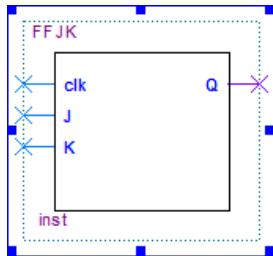


Fig. 3.2.8-1 Simbolo di FFJK

**Porte:** in Tabella 3.2.8-1 gli input e output di FFJK.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
J	I	1	Set
K	I	1	Reset
Q	O	1	Uscita

Tabella 3.2.8-1 Input e output di FFJK

FFJK è un blocco che riprende il comportamento del Flip Flop JK senza l'uscita negata e senza toggle quando entrambi gli ingressi sono alti. L'uscita è posta a 1 quando il Set "J" è attivo, è invece posta a 0 quando il Reset "K" è attivo. In tutte le altre combinazioni il valore dell'uscita non cambia.

**Tabella di verità** (Tabella 3.2.8-2):

J	K	Q	Descrizione
0	0	Q	Memoria (nessun cambiamento)
0	1	0	Reset
1	0	1	Set
1	1	Q	Memoria (nessun cambiamento)

Tabella 3.2.8-2 Tabella di verità di FFJK

**Codice VHDL:**

```

library IEEE;
use IEEE.std_logic_1164.all;
entity FFJK is
Port(
    clk : in std_logic;
    J: in std_logic;
    K: in std_logic;
    Q: out std_logic:= '0');
end FFJK;
architecture behave of FFJK is
begin
process(clk, J, K)
begin
    if (clk'event and clk = '1') then
        if J = '1' then
            Q <= '1';
        elsif K = '1' then
            Q <= '0';
        end if;
    end if;
end process;
end behave;
```

### 3.2.9 LCD

In Fig. 3.2.9-1 il simbolo di LCD:

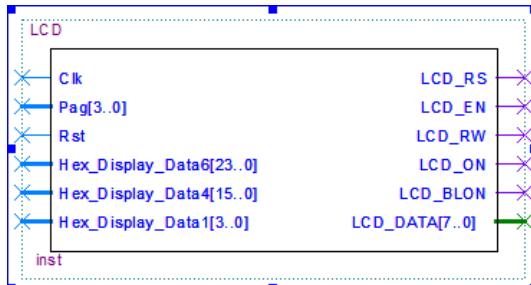


Fig. 3.2.9-1 Simbolo di LCD

**Porte:** in Tabella 3.2.9-1 gli input e output di LCD.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
Pag	I	4	Selettore pagine
Rst	I	1	Reset
Hex_Display_Data6	I	24	Campo live ID
Hex_Display_Data4	I	16	Campo live Password
Hex_Display_Data1	I	4	Campo live porta
LCD_RS	O	1	Segnale di gestione LCD – Register Select
LCD_EN	O	1	Segnale di gestione LCD – Enable
LCD_RW	O	1	Segnale di gestione LCD – Read/Write
LCD_ON	O	1	Segnale di gestione LCD – On/Off
LCD_BLON	O	1	Segnale di gestione LCD – Backlight On
LCD_DATA	O	8	Segnale di gestione LCD – Carattere

Tabella 3.2.9-1 Input e output di LCD

Questo blocco serve a gestire l'utilizzo dello schermo a cristalli liquidi presente sulla board. Il blocco è già fornito dal prof. Santinelli e, nel suo funzionamento generale, non verrà trattato ulteriormente. La modifica essenziale del blocco è l'introduzione delle pagine e dei segnali di campo live corrispondenti. La scelta della pagina da visualizzare è determinata dal segnale "Pag". La tabella 3.2.9-2 mostra la corrispondenza tra il selettore pagina e la pagina da visualizzare sullo schermo (% rappresenta un carattere dedicato al campo live):

Pag	Nome Pagina	Messaggio da visualizzare	Campo live
0000	Pagina0	PREMI UN TASTO QUALSIASI	
0001	Pagina1	INSERIRE ID: %%%%%	6 caratteri: ID utente
0010	Pagina2	UTENTE NON RICONOSCIUTO	
0011	Pagina3	INSERIRE PASSWORD: %%%%	4 caratteri: Password
0100	Pagina4	PASSWORD ERRATA	
0101	Pagina5	INSERIRE NUMERO PORTA: %	1 carattere: Numero porta
0110	Pagina6	NUMERO ERRATO	
0111	Pagina7	PORTA % ATTIVA	1 carattere: Numero porta
else	PaginaVuota		

Tabella 3.2.9-2 Schermate selezionabili

Nella visualizzazione del campo live è stata modificata la gestione dei caratteri oltre il 9, che in questa versione vengono visualizzati come trattini bassi. La riga in cui è stata apportata questa modifica è stata scritta in verde per essere visualizzata facilmente.

#### Codice VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Questo blocco serve per utilizzare lo schermo a cristalli liquidi sulla board.
-- Le frasi da dover mostrare sono 9, di cui 3 hanno un campo live.
-- Questo blocco nasce da un blocco già esistente LCD fornito dal prof. Santinelli.
-- Una modifica importante è stata realizzata alla riga 303, dove al posto di ogni carattere non numerico
-- viene inserito un underscore. Questo per poter illustrare la presenza di un campo live che deve essere
-- riempito.
-- 

entity LCD is
Port (
    Clk          : in std_logic; -- Clock
    Pag         : in std_logic_vector(3 downto 0); -- Selettore della pagina
    Rst          : in std_logic := '1'; -- Reset iniziale, sempre attivo
    Hex_Display_Data6   : in std_logic_vector((6*4)-1 downto 0); -- Campo Live ID
    Hex_Display_Data4   : in std_logic_vector((4*4)-1 downto 0); -- Campo Live Password
    Hex_Display_Data1   : in std_logic_vector((1*4)-1 downto 0); -- Campo Live Porta
    LCD_RS: out std_logic; -- Segnali di gestione dell'LCD
    LCD_EN: out std_logic; --
    LCD_RW       : out std_logic; --
    LCD_ON        : out std_logic; --
    LCD_BLON      : out std_logic; --
    LCD_DATA      : inout std_logic_vector(7 downto 0)); --
end LCD;

architecture beh of LCD is
-- Definisco un type per la stringa di caratteri come array di std logic vector.
type character_string is array (0 to 31) of std_logic_vector(7 downto 0);
-- Macchina a stati
type STATE_TYPE is (    HOLD, FUNC_SET, DISPLAY_ON, MODE_SET, Print_String,
                        LINE2, RETURN_HOME, DROP_LCD_E, RESET1,
RESET2, RESET3,
                        DISPLAY_OFF, DISPLAY_CLEAR);

signal state, next_command           : STATE_TYPE;
signal LCD_display_string          : character_string;
signal DATA_BUS_VALUE, Next_Char   : std_logic_vector(7 downto 0);
signal CLK_COUNT_400HZ              : std_logic_vector(19 downto 0);
signal CHAR_COUNT                  : std_logic_vector(4 downto 0);
signal CLK_400HZ_Enable            : std_logic;

```

```

signal LCD_RW_INT : std_logic;
-- Pagine
signal Pagina0      : character_string;
signal Pagina1      : character_string;
signal Pagina2      : character_string;
signal Pagina3      : character_string;
signal Pagina4      : character_string;
signal Pagina5      : character_string;
signal Pagina6      : character_string;
signal Pagina7      : character_string;
signal PaginaVuota   : character_string;

begin
    --Elenco pagine:
    --%% -> Campo Live

    Pagina0 <= (
        -- PREMI UN
        -- TASTO QUALSIASI
        -- Line 1 Page 0
        X"50", X"52", X"45", X"4D", X"49", X"20", X"55", X"4E",
        X"20", X"20", X"20", X"20", X"20", X"20", X"20", X"20",
        -- Line 2 Page 0
        X"54", X"41", X"53", X"54", X"4F", X"20", X"51", X"55",
        X"41", X"4C", X"53", X"49", X"41", X"53", X"49", X"20");
    Pagina1 <= (
        -- INSERIRE ID:
        -- %%%%%%
        -- Line 1 Page 1
        X"49", X"4E", X"53", X"45", X"52", X"49", X"52", X"45",
        X"20", X"49", X"44", X"3A", X"20", X"20", X"20", X"20",
        -- Line 2 Page 1
        X"0" & Hex_Display_Data6(23 downto 20), X"0" & Hex_Display_Data6(19 downto 16),
        X"0" & Hex_Display_Data6(15 downto 12), X"0" & Hex_Display_Data6(11 downto 8),
        X"0" & Hex_Display_Data6(7 downto 4), X"0" & Hex_Display_Data6(3 downto 0),
        X"20", X"20",
        X"20", X"20", X"20", X"20", X"20", X"20", X"20");
    Pagina2 <= (
        -- UTENTE NON
        -- RICONOSCIUTO
        -- Line 1 Page 2
        X"55", X"54", X"45", X"4E", X"54", X"45", X"20", X"4E",
        X"4F", X"4E", X"20", X"20", X"20", X"20", X"20", X"20",
        -- Line 2 Page 2
        X"52", X"49", X"43", X"4F", X"4E", X"4F", X"53", X"43",
        X"49", X"55", X"54", X"4F", X"20", X"20", X"20", X"20");
    Pagina3 <= (
        -- INSERIRE
        -- PASSWORD: %%%%

```

```

-- Line 1 Page 3
X"49", X"4E", X"53", X"45", X"52", X"49", X"52", X"45",
    X"20", X"20", X"20", X"20", X"20", X"20", X"20",
-- Line 2 Page 3
X"50", X"41", X"53", X"53", X"57", X"4F", X"52", X"44",
    X"3A", X"20",
    X"0" & Hex_Display_Data4(15 downto 12), X"0" & Hex_Display_Data4(11 downto 8),
    X"0" & Hex_Display_Data4(7 downto 4), X"0" & Hex_Display_Data4(3 downto 0),
    X"20", X"20");
Pagina4 <= (
-- PASSWORD ERRATA
--

-- Line 1 Page 4
X"50", X"41", X"53", X"53", X"57", X"4F", X"52", X"44",
    X"20", X"45", X"52", X"52", X"41", X"54", X"41", X"20",
-- Line 2 Page 4
X"20", X"20", X"20", X"20", X"20", X"20", X"20", X"20",
    X"20", X"20", X"20", X"20", X"20", X"20", X"20");
Pagina5 <= (
-- INSERIRE NUMERO
-- PORTA: %
-- Line 1 Page 5
X"49", X"4E", X"53", X"45", X"52", X"49", X"52", X"45",
    X"20", X"4E", X"55", X"4D", X"45", X"52", X"4F", X"20",
-- Line 2 Page 5
X"50", X"4F", X"52", X"54", X"41", X"3A", X"20", X"0" & Hex_Display_Data1(3 downto 0),
    X"20", X"20", X"20", X"20", X"20", X"20", X"20");
Pagina6 <= (
-- NUMERO ERRATO
--

-- Line 1 Page 6
X"4E", X"55", X"4D", X"45", X"52", X"4F", X"20", X"45",
    X"52", X"52", X"41", X"54", X"4F", X"20", X"20", X"20",
-- Line 2 Page 6
X"20", X"20", X"20", X"20", X"20", X"20", X"20", X"20",
    X"20", X"20", X"20", X"20", X"20", X"20", X"20");
Pagina7 <= (
-- PORTA % ATTIVA
--

-- Line 1 Page 7
X"50", X"4F", X"52", X"54", X"41", X"20", X"0" & Hex_Display_Data1(3 downto 0), X"20",
    X"41", X"54", X"54", X"49", X"56", X"41", X"20", X"20",
-- Line 2 Page 7
X"20", X"20", X"20", X"20", X"20", X"20", X"20", X"20",
    X"20", X"20", X"20", X"20", X"20", X"20", X"20");
PaginaVuota <= (
-- 
--

-- Line 1 Page Empty
X"20", X"20", X"20", X"20", X"20", X"20", X"20", X"20",
    X"20", X"20", X"20", X"20", X"20", X"20", X"20", X"20",
-- Line 2 Page Empty

```

```
X"20", X"20", X"20", X"20", X"20", X"20", X"20",
X"20", X"20", X"20", X"20", X"20", X"20");
```

-- Selezione Pagina:

```
process(Pag)
begin
  if (Pag = "0000") then LCD_display_string <= Pagina0;
  elsif (Pag = "0001") then LCD_display_string <= Pagina1;
  elsif (Pag = "0010") then LCD_display_string <= Pagina2;
  elsif (Pag = "0011") then LCD_display_string <= Pagina3;
  elsif (Pag = "0100") then LCD_display_string <= Pagina4;
  elsif (Pag = "0101") then LCD_display_string <= Pagina5;
  elsif (Pag = "0110") then LCD_display_string <= Pagina6;
  elsif (Pag = "0111") then LCD_display_string <= Pagina7;
  else      LCD_display_string <= PaginaVuota;
  end if;
end process;
```

-- Bidirectional tri-state LCD data bus

```
LCD_DATA <= DATA_BUS_VALUE when LCD_RW_INT = '0' else "ZZZZZZZZ";
```

-- get next character in display string

```
Next_Char <= LCD_display_string(conv_integer(CHAR_COUNT));
LCD_RW <= LCD_RW_INT;
```

-- LCD power on

```
LCD_ON      <= '1';
```

-- LCD back light on

```
LCD_BLON <='1';
```

```
process
begin
```

```
  wait until Clk'event and Clk = '1';
  if Rst = '0'      then
    CLK_COUNT_400HZ <= X"00000";
    CLK_400HZ_Enable <= '0';
  else
    if CLK_COUNT_400HZ < X"0F424"      then
      CLK_COUNT_400HZ <= CLK_COUNT_400HZ+1;
      CLK_400HZ_Enable <= '0';
    else
      CLK_COUNT_400HZ <= X"00000";
      CLK_400HZ_Enable <= '1';
    end if;
  end if;
end process;
```

```
process(Clk, Rst)
```

```

begin
if Rst = '0'      then
    state <= RESET1;
    DATA_BUS_VALUE <= X"38";
    next_command <= RESET2;
    LCD_EN <= '1';
    LCD_RS <= '0';
    LCD_RW_INT <= '1';
elsif(Clk'event and Clk = '1')   then
-- State Machine to send commands and data to LCD DISPLAY
    if(CLK_400HZ_Enable = '1') then
        case state is
-- Set Function to 8-bit transfer and 2 line display with 5x8 Font size
-- see Hitachi HD44780 family data sheet for LCD command and timing details
            when RESET1 =>
                LCD_EN                  <= '1';
                LCD_RS                 <= '0';
                LCD_RW_INT              <= '0';
                DATA_BUS_VALUE <= X"38";
                state                  <= DROP_LCD_E;
                next_command           <= RESET2;
                CHAR_COUNT             <= "00000";
            when RESET2 =>
                LCD_EN                  <= '1';
                LCD_RS                 <= '0';
                LCD_RW_INT              <= '0';
                DATA_BUS_VALUE <= X"38";
                state                  <= DROP_LCD_E;
                next_command           <= RESET3;
            when RESET3 =>
                LCD_EN                  <= '1';
                LCD_RS                 <= '0';
                LCD_RW_INT              <= '0';
                DATA_BUS_VALUE <= X"38";
                state                  <= DROP_LCD_E;
                next_command           <= FUNC_SET;
            when FUNC_SET =>
                LCD_EN                  <= '1';
                LCD_RS                 <= '0';
                LCD_RW_INT              <= '0';
                DATA_BUS_VALUE <= X"38";
                state                  <= DROP_LCD_E;
                next_command           <= DISPLAY_OFF;
-- Turn off Display and turn off cursor

```

```

when DISPLAY_OFF =>

    LCD_EN          <= '1';
    LCD_RS          <= '0';
    LCD_RW_INT      <= '0';
    DATA_BUS_VALUE <= X"08";
    state           <= DROP_LCD_E;
    next_command    <= DISPLAY_CLEAR;

-- Clear Display and Turn off cursor
when DISPLAY_CLEAR =>

    LCD_EN          <= '1';
    LCD_RS          <= '0';
    LCD_RW_INT      <= '0';
    DATA_BUS_VALUE <= X"01";
    state           <= DROP_LCD_E;
    next_command    <= DISPLAY_ON;

-- Turn on Display and Turn off cursor
when DISPLAY_ON =>

    LCD_EN          <= '1';
    LCD_RS          <= '0';
    LCD_RW_INT      <= '0';
    DATA_BUS_VALUE <= X"0C";
    state           <= DROP_LCD_E;
    next_command    <= MODE_SET;

-- Set write mode to auto increment address and move cursor to the right
when MODE_SET =>

    LCD_EN          <= '1';
    LCD_RS          <= '0';
    LCD_RW_INT      <= '0';
    DATA_BUS_VALUE <= X"06";
    state           <= DROP_LCD_E;
    next_command    <= Print_String;

-- Write ASCII hex character in first LCD character location
when Print_String =>

    state <= DROP_LCD_E;
    LCD_EN          <= '1';
    LCD_RS          <= '1';
    LCD_RW_INT      <= '0';

-- ASCII character to output
if Next_Char(7 downto 4) /= X"0" then
    DATA_BUS_VALUE <= Next_Char;
else
    if Next_Char(3 downto 0) < 10 then

```

```

        DATA_BUS_VALUE <= X"3" &
Next_Char(3 downto 0);
                                Else
-- qualsiasi numero superiore al nove è visualizzato come underscore
                                DATA_BUS_VALUE <= X"5F";
end if;
end if;

-- Loop to send out 32 characters to LCD Display (16 by 2 lines)
if (CHAR_COUNT < 31 ) and (Next_Char /= X"FE") then
    CHAR_COUNT <= CHAR_COUNT + 1;
else
    CHAR_COUNT <= "00000";
end if;

-- Jump to second line ?
if CHAR_COUNT = 15  then next_command <= line2;

-- return to first line ?
elsif (CHAR_COUNT = 31) or (Next_Char = X"FE") then
    next_command <= RETURN_HOME;
else
    next_command <= Print_String;
end if;

-- Set write address to line 2 character 1
when LINE2 =>
    LCD_EN          <= '1';
    LCD_RS          <= '0';
    LCD_RW_INT      <= '0';
    DATA_BUS_VALUE <= X"C0";
    state           <= DROP_LCD_E;
    next_command     <= Print_String;

-- Return write address to first character position on line 1
when RETURN_HOME =>
    LCD_EN          <= '1';
    LCD_RS          <= '0';
    LCD_RW_INT      <= '0';
    DATA_BUS_VALUE <= X"80";
    state           <= DROP_LCD_E;
    next_command     <= Print_String;

-- Drop LCD E line - falling edge loads inst/datato LCD controller
when DROP_LCD_E =>
    LCD_EN <= '0';
    state <= HOLD;

-- Hold LCD inst/data valid after falling edge of E line
when HOLD =>
    state <= next_command;
end case;
end if;
end if;
end process;
end beh;

```

### 3.2.10 Mux4\_1

In Fig. 3.2.10-1 il simbolo di Mux4\_1:

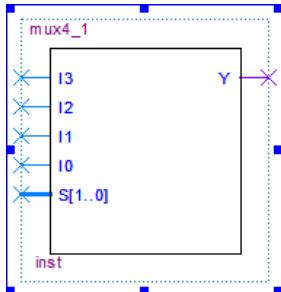


Fig. 3.2.10-1 Simbolo di Mux4\_1

**Porte:** in Tabella 3.2.10-1 gli input e output di Mux4\_1.

Pin	I/O	bit	Descrizione
I3	I	1	Ingresso, bit 3
I2	I	1	Ingresso, bit 2
I1	I	1	Ingresso, bit 1
I0	I	1	Ingresso, bit 0
S	I	2	Selettore
Y	O	1	Uscita

Tabella 3.2.10-1 Input e output di Mux4\_1

Questo blocco è un classico multiplexer 4 a 1 e ha quindi lo scopo di selezionare l'ingresso da portare in uscita. Il funzionamento è sintetizzato dalla seguente tabella (Tabella 3.2.10-2):

S	I->Y
00	I0
01	I1
10	I2
11	I3

Tabella 3.2.10-2 Tabella di verità di Mux4\_1

**Codice VHDL:**

```

library ieee;
use ieee.std_logic_1164.all;
entity mux4_1 is
Port(
    I3 : in STD_LOGIC; -- Ingressi
    I2 : in STD_LOGIC;
    I1 : in STD_LOGIC;
    I0 : in STD_LOGIC;
    S : in STD_LOGIC_VECTOR (1 downto 0); --segnale dal contatore
    Y : out STD_LOGIC); -- Uscita
end mux4_1;

architecture Behavioral of mux4_1 is
begin
with S select
    Y <= I0 when "00", --diventa uno solo se l'input connesso è uguale a uno
    I1 when "01",
    I2 when "10",
    I3 when others;
end Behavioral;

```

### 3.2.11 Packet\_Transfer

In Fig. 3.2.11-1 il simbolo di Packet\_Transfer:

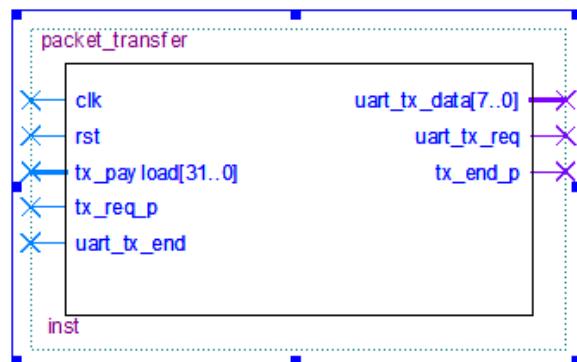


Fig. 3.2.11-1 Simbolo di packet\_transfer

**Porte:** in Tabella 3.2.11-1 gli input e output di Packet\_Transfer.

Pin	I/O	bit	Descrizione
clk	I	1	clock
rst	I	1	reset
Tx_payload[31..0]	I	32	Dati da trasmettere
Tx_req_p	I	1	Richiesta di inizio trasmissione
Uart_tx_end	I	1	Fine della trasmissione (uart)
Uart_tx_data[7..0]	O	1	Byte di dati da trasmettere
Uart_tx_req	O	1	Richiesta di inizio trasmissione (uart)
Tx_end_p	O	1	Fine trasmissione

Tabella 3.2.11-1 Input e output di packet\_transfer

Questo blocco ci è stato fornito dal Prof. Santinelli ma è stato modificato.

Per questo progetto è necessario inviare due tipi di dati: l'alive, che è un singolo byte, e alcuni pacchetti di dati da 6 byte ciascuno. Il blocco fornito aveva al suo interno il blocco UART, ciò non ci avrebbe permesso di usarlo per inviare l'alive perciò è stato eliminato. A tale scopo sono state modificate le porte di ingresso e uscita, che ora devono interagire con il blocco UART esterno. È stata eliminata anche la parte che gestiva la ricezione dei pacchetti, ora questo blocco è in grado solo di inviare. Per i fini di questo progetto è sufficiente che questo blocco si occupi dell'invio di 2 pacchetti di dati quando viene attivato "tx\_req\_p" (corrispondente al precedente "tx\_req").

Il pacchetto dati è composto da un header di 2 byte, AA 55 (esadecimale) e da un payload di 4 byte che è il dato ricevuto in ingresso. Questi 6 byte vengono inviati alla UART uno alla volta in sequenza.

Per vedere come il Packet\_Transfer è stato collegato alla UART e per conoscere i messaggi che invia si rimanda al blocco COM\_PIC.

**Codice VHDL:**

```
library ieee;
use ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
-- Send data packets. Each packet has:
-- 2 bytes header, 0xAA 0x55
-- 4 bytes payload
--   Packet format:
--   | header | 4 bytes payload (bits 31 to 0) |
```

```

-- header = 0xAA 0x55
--
-- eg:
--
-- payload = 0x01020304
-- packet: 0xAA 0x55 0x01 0x02 0x03 0x04
--
-- payload = 0x01AA0304
-- packet: 0xAA 0x55 0x01 0xAA 0x00 0x03 0x04
--
-- When a packet is sent, a 0x00 byte stuffing is added after
-- each 0xAA occurring in payload.
-----
```

---

```

entity packet_transfer is
    generic (
        CLK_FREQ           : integer :=50;          -- Main frequency (MHz)
        TIMEOUT            : integer :=500;         -- Rx Time out (ms)
        BAUD               : integer :=9600;        -- Baud rate (bps)
    );
    Port (
        clk                : in std_logic;        -- Main clock
        rst                : in std_logic;        -- Main reset
        tx_payload         : in std_logic_vector(31 downto 0); -- Data to transmit (32 bits)
        uart_tx_data       : out std_logic_vector(7 downto 0);
        tx_req_p           : in std_logic;
        uart_tx_req         : out std_logic;        -- Request send of data
        uart_tx_end         : in std_logic;        -- Data sended (pulsed, one clock wide)
        tx_end_p            : out std_logic
    );
end packet_transfer;
```

---

```

architecture Behavioral of packet_transfer is
```

---

```

    -- Constants

    constant RST_LVL      : std_logic := '1';
    constant HEAD0          : std_logic_vector(7 downto 0) := X"AA";
    constant HEAD1          : std_logic_vector(7 downto 0) := X"55";
    constant STUFF          : std_logic_vector(7 downto 0) := X"00";

    -- Types
    type state is (idle, header0, header1, payload, wait_tx_end_rising, wait_tx_end_falling, wait_state,
stop);

    -- TX Signals

    signal tx_fsm           : state;
    -- Control of transmission
    signal tx_fsm_next       : state;
```

```

signal tx_req_int : std_logic;
signal tx_payload_reg: std_logic_vector(31 downto 0);
signal tx_payload_int: std_logic_vector(31 downto 0);
signal tx_data_cnt : std_logic_vector(1 downto 0);

begin
-----
-- Data packet Transmitter

tx_proc:process(clk)
begin
if clk'event and clk = '1' then

-- Reset condition

if rst = RST_LVL then

    tx_fsm          <= idle;
    tx_fsm_next     <= idle;
    tx_end_p        <= '0';
    tx_req_int      <= '0';
    uart_tx_req    <= '0';
    uart_tx_data   <= (others=>'0');
    tx_payload_int <= (others=>'0');
    tx_data_cnt     <= (others=>'1');

else

    if(tx_req_p = '1') then
        tx_req_int <='1';
        tx_payload_reg <= tx_payload;
    end if;

-- FSM description
    case tx_fsm is

        -- Wait to transfer data
        when idle =>
            tx_data_cnt          <= (others=>'1');
            uart_tx_req          <= '0';
            tx_end_p              <= '0';
            if tx_req_int = '1' then
                tx_fsm          <= header0;
            end if;

        -- Send header 0
        when header0 =>

            uart_tx_data  <= HEAD0;
            uart_tx_req    <= '1';

end if;

```

```

tx_req_int           <='0';
tx_fsm               <= wait_tx_end_rising;
tx_fsm_next         <= header1;

-- Send header 1
when header1 =>
    tx_payload_int <= tx_payload_reg;
    uart_tx_data   <= HEAD1;
    uart_tx_req     <= '1';
    tx_fsm          <= wait_tx_end_rising;
    tx_fsm_next     <= payload;

when payload =>
    uart_tx_data   <= tx_payload_int(31 downto 24);
    uart_tx_req     <= '1';

-- Byte stuffing: insert an extra X"00" byte
if tx_payload_int(31 downto 24) = HEAD0 then
    tx_payload_int <= X"00"&tx_payload_int(23
downto 0);
    tx_fsm           <=
wait_tx_end_rising;
    tx_fsm_next     <= payload;

elsif tx_data_cnt = 0 then
    tx_data_cnt    <= (others=>'1');
    tx_fsm           <=
wait_tx_end_rising;
    tx_fsm_next     <= stop;

else
    tx_payload_int <= tx_payload_int(23
downto 0)& X"00";
    tx_data_cnt    <= tx_data_cnt - 1;
    tx_fsm           <=
wait_tx_end_rising;
    tx_fsm_next     <= payload;
end if;

when stop =>
    tx_end_p        <= '1';
    tx_fsm           <= idle;

when wait_tx_end_rising =>
    uart_tx_req     <= '0';

```

```
if uart_tx_end = '1' then
    tx_fsm      <= wait_state;
end if;

when wait_state =>
    if uart_tx_end = '0' then
        tx_fsm      <= tx_fsm_next;
    end if;

when others => null;
end case;
end if;
end if;
end process;
end behavioral;
```

### 3.2.12 RAM

In Fig. 3.2.12-1 il simbolo di RAM:

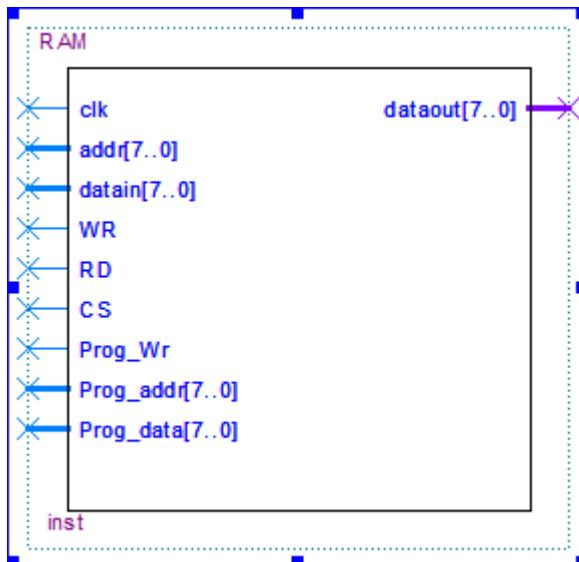


Fig. 3.2.12-1 Simbolo di RAM

**Porte:** in Tabella 3.2.12-1 gli input e output di RAM.

Pin	I/O	bit	Descrizione
clk	I	1	Clock
addr	I	8	Indirizzo
datain	I	8	Dato in ingresso per scrittura
WR	I	1	Segnale di abilitazione scrittura
RD	I	1	Segnale di abilitazione lettura
CS	I	1	Segnale di abilitazione
Prog_Wr	I	1	/
Prog_addr	I	1	/
Prog_data	I	8	/
dataout	O	8	Dato in uscita per lettura

Tabella 3.2.12-1 Input e output di RAM

Questo blocco è utilizzato per scrivere e leggere nella memoria RAM interna all'FPGA. Per poter memorizzare un carattere si deve abilitare la scrittura portando a 1 il segnale "WR"; in questo modo viene memorizzato nell'indirizzo "addr" il dato in "datain". Per poter leggere un dato salvato in memoria bisogna abilitare la lettura portando a 1 il segnale "RD", in questo modo sulla porta "dataout" sarà disponibile il dato memorizzato nell'indirizzo "addr". Essendo questo un blocco già fornito non sono previsti ulteriori approfondimenti.

#### Codice VHDL:

```
-- Quartus II VHDL Template
--Dual-port RAM
-- doppio canale: da una parte posso leggere e scrivere
-- dall'altra solo scrivere (canale priorario)
-- capacita della memoria: 256 celle da 8 bit
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

```
entity RAM is
```

```
Port
```

```
(
```

```
    clk      : in std_logic;
    -- lettura/scrittura
    addr    : in std_logic_vector(7 downto 0);
    datain  : in std_logic_vector(7 downto 0);
    dataout : out std_logic_vector(7 downto 0);
    WR      : in std_logic := '0';
    RD      : in std_logic := '1';
    CS      : in std_logic := '1';

    -- sola scrittura
    Prog_Wr : in std_logic := '0';
    Prog_addr : in std_logic_vector(7 downto 0);
    Prog_data : in std_logic_vector(7 downto 0)
```

```
);
```

```
end RAM;
```

```
architecture rtl of RAM is
```

```
-- Build a 2-D array type for the RAM
```

```
subtype word_t is std_logic_vector(7 downto 0);
type memory_t is array(2**8-1 downto 0) of word_t;
```

```
function init_ram
```

```
    return memory_t is
```

```
    variable tmp : memory_t := (others => (others => '0'));
```

```
begin
```

```
    for addr_pos in 0 to 2**8 - 1 loop
```

```
        -- Initialize each address with the address itself
```

```
        tmp(addr_pos) := std_logic_vector(to_unsigned(addr_pos, 8));
```

```
    end loop;
```

```
    return tmp;
```

```
end init_ram;
```

```
-- Declare the RAM signal and specify a default value. Quartus II
```

```
-- will create a memory initialization file (.mif) based on the
```

```
-- default value.
```

```
signal dataram : memory_t; -- := init_ram;
```

```
attribute ram_init_file : string;
```

```
attribute ram_init_file of dataram :
```

```
    signal is "RAMContent.mif";
```

```

-- Register to hold the address
signal addr_reg : natural range 0 to 2**8-1;

begin

process(clk)
begin
if(rising_edge(clk)) then
    -- scrittura secondaria prioritaria
    if (Prog_Wr='1') then
        dataram(to_integer(unsigned(Prog_addr))) <= Prog_data;
    elsif(WR = '1' and CS='1') then
        dataram(to_integer(unsigned(addr))) <= datain;
    end if;

-- Register the address for reading
    addr_reg <= to_integer(unsigned(addr));

end if;
end process;

dataout <= dataram(addr_reg) when (RD='1' and CS='1') else
(others=> 'Z');

end rtl;

```

### 3.2.13 Registro

In Fig. 3.2.13-1 il simbolo di Registro:

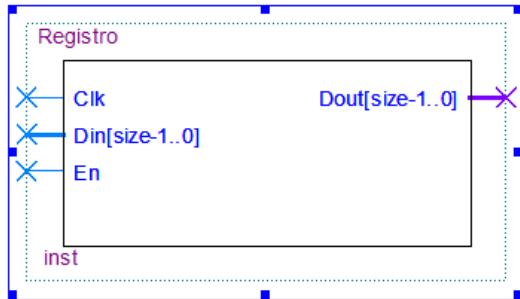


Fig. 3.2.13-1 Simbolo di Registro

**Porte:** in Tabella 3.2.13-1 gli input e output di Registro.

Pin	I/O	bit	Descrizione
clk	I	1	Clock
Din[size-1..0]	I	variabile	Dato in ingresso
En	I	1	Segnale di enable
Dout[size-1..0]	O	variabile	Dato in uscita

Tabella 3.2.13-1 Input e output di Registro

Lo scopo di questo blocco è quello di salvare i dati che riceve in ingresso.

Il salvataggio funziona tramite enable ("en"). Il dato in uscita è sempre uguale al dato salvato internamente "InternalValue", mentre quest'ultimo è uguale al dato in ingresso solo quando l'enable è a 1, altrimenti rimane uguale a sé stesso.

La dimensione del vettore è variabile in base al dato che si vuole memorizzare.

**Codice VHDL:**

```

library ieee;
use ieee.std_logic_1164.all;

entity Registro is
generic (size : integer:= 4);
Port( Clk : in STD_LOGIC;                                -- Clock
      Din : in STD_LOGIC_VECTOR (size-1 downto 0);        -- Data Input
      Dout : out STD_LOGIC_VECTOR (size-1 downto 0);       -- Data Output
      En : in STD_LOGIC);                                 -- Enable
end Registro;

architecture beh of Registro is
signal InternalValue : STD_LOGIC_VECTOR (size-1 downto 0) := (others => '0');
-- Il valore interno del registro è inizializzato a 0
signal NextValue : STD_LOGIC_VECTOR (size-1 downto 0);
begin
process (Clk)
begin
  if (Clk='1' and Clk'event) THEN
    InternalValue <= NextValue;
  end if;
end process;

```

```
process (En,Din,InternalValue) is
begin
-- Controllo enable
    NextValue <= InternalValue ;
    if (En = '1') then
        NextValue <= Din; -- input da Din
    end if;
end process;

Dout <= InternalValue ;
end architecture beh;
```

### 3.2.14 Tast2Bin

In Fig. 3.2.14-1 il simbolo di Tast2Bin:

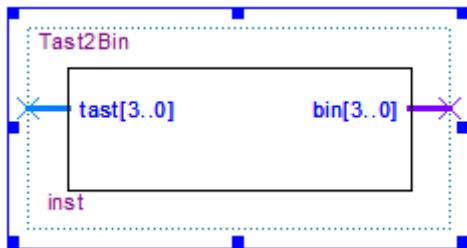


Fig. 3.2.14-1 Simbolo di Tast2Bin

**Porte:** in Tabella 3.2.14-1 gli input e output di Tast2Bin.

Pin	I/O	bit	Descrizione
tast	I	4	Posizione del carattere corrispondente sul tastierino
bin	O	4	Codifica in esadecimale del carattere convertito

Tabella 3.2.14-1 Input e output di Tast2Bin

Questo blocco combinatorio ha lo scopo di far corrispondere ad un determinato pulsante del tastierino 4x4 un carattere esadecimale.

La tabella di verità, già vista nella discussione del macro-blocco Tastierino, per la conversione tast – bin è la seguente (Tabella 3.2.14-2):

Tastierino	tast	Bin
1	1111	0001
2	1011	0010
3	0111	0011
A	0011	1010
4	1110	0100
5	1010	0101
6	0110	0110
B	0010	1011
7	1101	0111
8	1001	1000
9	0101	1001
C	0001	1100
*	1100	1110
0	1000	0000
#	0100	1111
D	0000	1101

Tabella 3.2.14-2 Conversione effettuata dal blocco Tast2Bin

**Codice VHDL:**

```
library ieee;
use ieee.std_logic_1164.all;

-- Questo blocco serve per convertire i valori della matrice del tastierino nei valori numerici corrispondenti.
-----
--1-2-3-A-
--4-5-6-B-
--7-8-9-C-
--*-0 #-D
--
```

--Le colonne partono da destra (0) a sinistra (3).  
--Le righe partono da giù (0) a su (3).  
--Es: D(0,0)... 2(3,2)... 1(3,3)  
--Il conteggio da 0 a 3 viene effettuato con 2 bit. Col = (Col1, Col0). Rig = (Rig1, Rig0).  
--hex = (Col1, Col0, Rig1, Rig0)

```
entity Tast2Bin is
Port( tast: in std_logic_vector(3 downto 0);
      bin: out std_logic_vector(3 downto 0));
end Tast2Bin;

architecture dflow of Tast2Bin is
begin
with tast select
bin <= "1101" when "0000", -- d = 13
      "1100" when "0001", -- C = 12
      "1011" when "0010", -- b = 11
      "1010" when "0011", -- A = 10
      "1111" when "0100", -- # = 15
      "1001" when "0101", -- 9
      "0110" when "0110", -- 6
      "0011" when "0111", -- 3
      "0000" when "1000", -- 0
      "1000" when "1001", -- 8
      "0101" when "1010", -- 5
      "0010" when "1011", -- 2
      "1110" when "1100", -- * = 14
      "0111" when "1101", -- 7
      "0100" when "1110", -- 4
      "0001" when "1111", -- 1
      "1111" when others;

end dflow;
```

### 3.2.15 Timer

In Fig. 3.2.15-1 il simbolo di Timer:

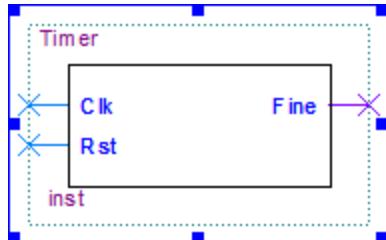


Fig. 3.2.15-1 Simbolo di Timer

**Porte:** in Tabella 3.2.15-1 gli input e output di Timer.

Pin	I/O	bit	Descrizione
Clk	I	1	Clock
Rst	I	1	Reset
Fine	O	1	Uscita a gradino. Timer attivo (0). Timer disattivo (1).

Tabella 3.2.15-1 Input e output di Timer

Questo blocco ha lo scopo di realizzare un contatore temporale che, raggiunto il tempo prestabilito, porta a 1 il segnale di uscita "Fine", il quale è rimasto a 0 durante il conteggio. L'ingresso reset "Rst" è il segnale che permette di far ripartire il contatore e azzerare l'uscita. Il segnale di Reset deve essere alto solo per un ciclo di clock. Nel programma sono utilizzati 3 segnali chiave:

- **count:** il segnale "count" si incrementa ad ogni ciclo di clock. Quando si raggiunge il valore di un secondo, in base alla frequenza del clock, si azzerà e incrementa il segnale "secondi".
- **secondi:** il segnale è incrementato di 1 quando il segnale "count" è azzerato.
- **state:** rappresenta il valore che dovrà assumere l'uscita "Fine".

**Codice VHDL:**

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;

-- Quando parte il segnale di reset (Rst) il segnale Fine si porta a 0
-- In quel momento, il segnale di count inizia a incrementare
-- Dopo un certo numero di secondi dettato dal Timeout, il segnale Fine si porta a 1 in attesa di un nuovo
reset

entity Timer is
    generic (
        Timeout : integer := 5; -- Tempo [sec]
        ClkFrq : integer := 50000000); -- Frequenza del Clock [Hz]
    Port (
        Clk : in std_logic; -- Clock
        Rst : in std_logic; -- Reset
        Fine : out std_logic); -- Uscita a gradino (0 <= Timer Attivo, 1 <= Fine Timer o Spento)
end Timer;

architecture Beh of Timer is
    signal count : integer := 0;           -- count: conta ogni ciclo di clock a partire dal reset

```

```

signal secondi : integer := 0;           -- secondi: conta i secondi
signal state : std_logic := '0';        -- state: stato interno

begin
process(Clk, Rst) is
begin
    if(Clk = '1' and Clk'event) then
        -- Quando il segnale di reset si attiva, Fine e il contatore si azzerano
        if Rst = '1' then
            count <= 0;
            secondi <= 0;
            state <= '0';
        else
            count <= count + 1;
            if(count = ClkFrq) then
                secondi <= secondi + 1;
                count <= 0;
                -- Al superamento del Timeout, lo stato futuro di si porta a '1'.
                if secondi = Timeout then
                    state <= '1';
                end if;
            end if;
        end if;
    end if;
end process;

Fine <= state;
end architecture Beh;

```

### 3.2.16 Transition\_Finder

In Fig. 3.2.16-1 il simbolo di Transition\_Finder:

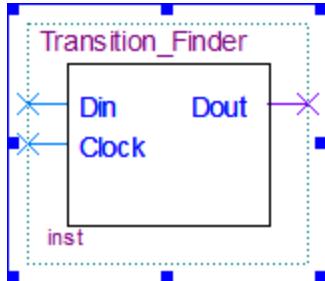


Fig. 3.2.16-1 Simbolo di Transition\_Finder

**Porte:** in Tabella 3.2.16-1 gli input e output di Transition\_Finder.

Pin	I/O	bit	Descrizione
Din	I	1	Segnale in ingresso
Clock	I	1	Clock
Dout	O	1	Segnale in uscita

Tabella 3.2.16-1 Input e output di Transition\_Finder

Questo blocco è una rete sequenziale in grado di rilevare la presenza di un gradino. Da un impulso di durata indefinita crea un impulso di durata di 1 ciclo di clock. È descritto tramite una macchina a stati finiti di Mealy, che prevede l'uso di un solo Flip Flop D. Il diagramma di Timing del Transition\_Finder è il seguente (Fig. 3.2.16-2):

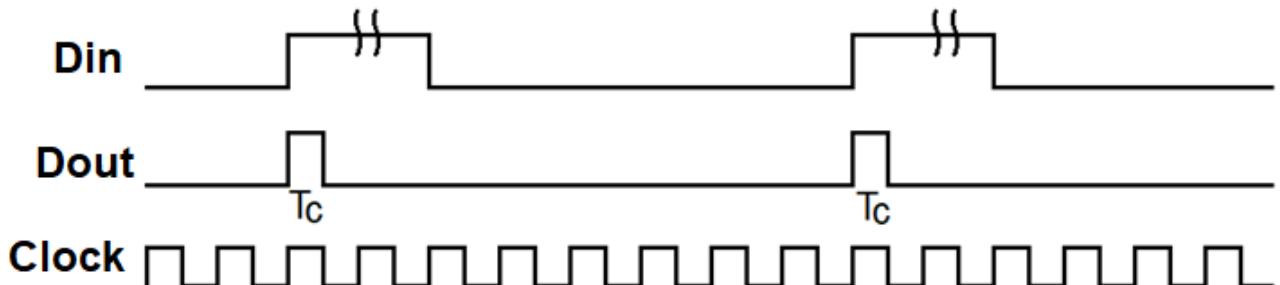


Fig. 3.2.16-2 Risposta nel tempo di Transition\_Finder

Questo sistema può essere descritto dal seguente diagramma degli stati per un automa di Mealy (Fig. 3.2.16-3):

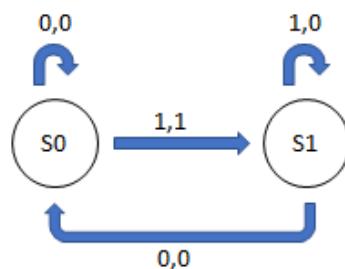


Fig. 3.2.16-3 Diagramma degli stati di Transition\_Finder

(Es: nella freccia 1,0 significa Din=1 e Dout=0)

La tabella di transizione degli stati è la seguente (Tabella 3.2.16-2):

	<b>y</b>	<b>Din</b>	<b>Y</b>	<b>Dout</b>
S0	0	0	0	0
	0	1	1	1
S1	1	0	0	0
	1	1	1	0

Tabella 3.2.16-2 Tabella di verità di Transition\_Finder

Dove Y è lo stato futuro (ingresso al Flip Flop) e y lo stato presente (uscita al Flip Flop). Per fare la sintesi in termini di somme di prodotti si ricavano le mappe di Karnaugh per Y (a sinistra) e "Dout" (a destra) (Tabella 3.2.16-3):

<b>Din\y</b>	0	1	<b>Din\y</b>	0	1
0	<b>0</b>	<b>0</b>	0	<b>0</b>	<b>0</b>
1	<b>1</b>	<b>1</b>	1	<b>1</b>	<b>0</b>

Tabella 3.2.16-3 Mappe di Karnaugh di Transition\_Finder

Da cui:

$$Y = D_{in}; D_{out} = \bar{y} D_{in}$$

Si realizza così la rete sequenziale dell'automa a stati finiti (Fig. 3.2.16-4).

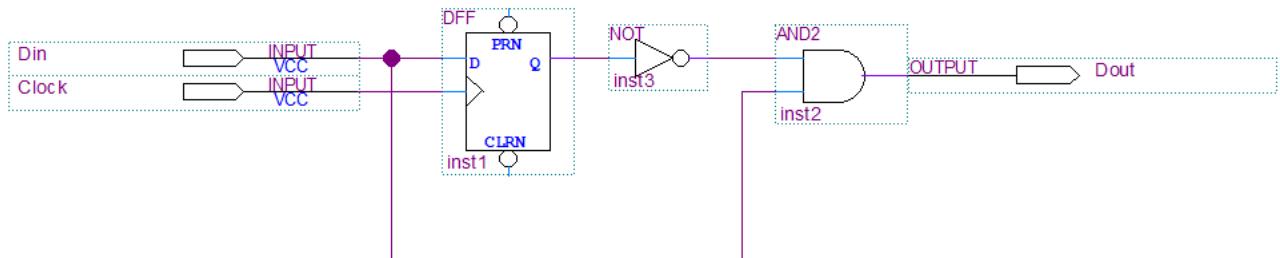


Fig. 3.2.16-4 Diagramma a blocchi di Transition\_Finder

### 3.2.17 UART

In Fig. 3.2.17-1 il simbolo di UART:

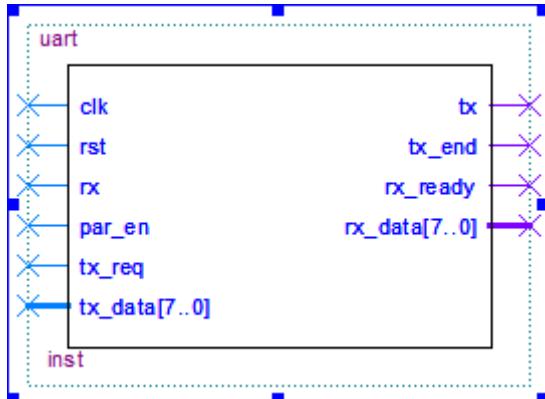


Fig. 3.2.17-1 Simbolo di UART

**Porte:** in Tabella 3.2.17-1 gli input e output di UART.

Pin	I/O	bit	Descrizione
clk	I	1	Clock
rst	I	1	Reset
rx	I	1	Rx, Ricezione
par_en	I	1	Attivazione del controllo di parità (Non utilizzato nel programma)
tx_req	I	1	Permette di iniziare la trasmissione del carattere presente sul bus tx_data
tx_data	I	8	Byte da trasmettere
tx	O	1	Tx, Trasmissione
tx_end	O	1	Segnale di avvenuta trasmissione del carattere
rx_ready	O	1	Segnala l'avvenuta ricezione di un nuovo carattere
rx_data	O	8	Byte ricevuto

Tabella 3.2.17-1 Input e output di UART

Questo blocco serve a controllare il modulo UART che gestisce la comunicazione secondo lo standard RS-232. Lo schema del sistema si può tradurre in due sottosistemi: un sistema logico che gestisce l'elaborazione e il trasferimento dei dati (Datapath), e una macchina a stati finiti (Control Unit) che serve a coordinare questa attività e a muovere i segnali di controllo correttamente. Il blocco è già fornito dal prof. Santinelli e, non essendo stato modificato, non verrà trattato ulteriormente.

**Codice VHDL:**

```

library ieee;
use ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-----
-- ver. 1.5, 15/04/2019
-- uart
-- 
-- 15/04/2019:
-- Santinelli
-- Ora il segnale tx_end si attiva per la durata di un ciclo di clock per segnalare l'avvenuta trasmissione di un carattere.
-- Vedere linee file seorgenre (15/04/2019)

```

```

-- Ver. 1.4, 29/04/2016:
-- Santinelli
-- Aggiunto clear sincrono del segnale tx_end nel processo tx_proc.
-- Il segnale tx_end non veniva resettato all'attivarsi del segnale rst.

-- Ver. 1.3, 16/10/2014:
-- Vezzani
-- Corretto problema su reset di tx_req_int nello stato stop3:
-- Spostato tx_req_int <= '0'; dallo stato stop3 allo stato stop2
--
```

---

```

entity uart is
generic (CLK_FREQ      : integer :=50;           -- Main frequency (MHz)
         SER_FREQ       : integer :=9600          -- Baud rate (bps)
);
Port (
    -- Control
    clk            : in   std_logic;           -- Main clock
    rst            : in   std_logic;           -- Main reset
    -- External Interface
    rx             : in   std_logic;           -- RS232 received serial data
    tx             : out  std_logic;           -- RS232 transmitted serial data
    -- RS232/UART Configuration
    par_en         : in   std_logic;           -- Parity bit enable
    -- uPC Interface
    tx_req         : in   std_logic;           -- Request SEND of data
    tx_end         : out  std_logic;           -- Data SENDED
    tx_data        : in   std_logic_vector(7 downto 0); -- Data to transmit
    rx_ready       : out  std_logic;           -- Received data ready to uPC read
    rx_data        : out  std_logic_vector(7 downto 0); -- Received data
);
end uart;

```

```

architecture Behavioral of uart is

    -- Constants
    constant UART_IDLE    : std_logic := '1';
    constant UART_START   : std_logic := '0';
    constant PARITY_EN    : std_logic := '1';
    constant RST_LVL      : std_logic := '1';

    -- Types
    type state is (idle,ck_start_bit,data,parity,stop1,stop2,stop3);           -- Stop1 and Stop2
    are inter frame gap signals

    -- RX Signals
    signal rx_fsm         : state;
    Control of reception

```

```

    signal rx_clk_en          : std_logic;                      --
Received clock enable
    signal rx_par_bit         : std_logic;                      --
Calculated Parity bit
    signal rx_data_deb       : std_logic;                      -- Debounce
RX data
    signal rx_data_tmp        : std_logic_vector(7 downto 0); -- Serial to parallel converter
    signal rx_data_cnt        : std_logic_vector(2 downto 0); -- Count received bits
    signal rx_clk_cnt         : std_logic_vector(4 downto 0); -- Count rx_clk cycles

-- TX Signals
    signal tx_fsm              : state;                         --
Control of transmission
    signal tx_clk_en           : std_logic;                      --
Transmited clock enable
    signal tx_par_bit          : std_logic;                      --
Calculated Parity bit
    signal tx_req_int          : std_logic;                      --
Internal tx request
    signal tx_data_tmp         : std_logic_vector(7 downto 0); -- Parallel to serial converter
    signal tx_data_cnt          : std_logic_vector(2 downto 0); -- Count transmited bits

```

*begin*

```

tx_clk_gen:process(clk)
    variable counter      : integer range 0 to
conv_integer((CLK_FREQ*1_000_000)/SER_FREQ-1);
begin
    if clk'event and clk = '1' then
        -- Normal Operation
        if counter = (CLK_FREQ*1_000_000)/SER_FREQ-1 then
            tx_clk_en     <=  '1';
            counter      :=  0;
        else
            tx_clk_en     <=  '0';
            counter      :=  counter + 1;
        end if;
        -- Reset condition
        if rst = RST_LVL then
            tx_clk_en     <=  '0';
            counter      :=  0;
        end if;
    end if;
end process;

```

```

-- Serial data Transmitter
tx_proc:process(clk)
    variable data_cnt      : std_logic_vector(2 downto 0);
begin
    if clk'event and clk = '1' then
        -- Reset condition

```

```

if rst = RST_LVL then
    tx                                <=      UART_IDLE;
    tx_fsm                <=      idle;
    tx_par_bit            <=      '0';
    tx_req_int           <=      '0';
    tx_data_tmp           <=      (others=>'0');
    tx_data_cnt           <=      (others=>'0');
    tx_end                <=      '0';
else
    -- external tx request detection
    if(tx_req = '1') then
        tx_req_int <='1';
        tx_data_tmp  <=      tx_data;
    end if;
    -- Clear tx_end signal
    tx_end                <=      '0'; -- Added(RV 15/04/2019)

    if tx_clk_en = '1' then

        -- FSM description
        case tx_fsm is

            -- Wait to transfer data
            when idle =>
                -- Send Init Bit
                if tx_req_int = '1' then
                    tx                                <=
UART_START;
                    tx_fsm                <=      data;
                    tx_data_cnt           <=      (others=>'1');
                    tx_par_bit            <=      '0';
                end if;

            -- Data receive
            when data =>
                tx                                <=
tx_data_tmp(0);
                tx_par_bit            <=      tx_par_bit xor
tx_data_tmp(0);
                if tx_data_cnt = 0 then
                    if par_en = PARITY_EN then
                        tx_fsm <=      parity;
                    else
                        tx_fsm <=      stop1;
                    end if;
                    tx_data_cnt           <=      (others=>'1');
                else
                    tx_data_tmp           <=      '0' & tx_data_tmp(7
downto 1);
                    tx_data_cnt           <=      tx_data_cnt - 1;
                end if;

```

```

-- Send parity bit
when parity =>
    tx
        <=
tx_par_bit;
    tx_fsm
        <= stop1;

-- End of communication
when stop1 =>
    -- Send Stop Bit
    tx
        <=
UART_IDLE;
    tx_fsm
        <= stop2;

when stop2 =>
    -- Set transmission completed flag (tx_end <-- 1)
    tx_end
        <= '1';
    -- I can accept a new tx request from the following
fast clock (RV 16/10/2014)
    tx_req_int
        <= '0';
    tx
        <=
UART_IDLE;
    tx_fsm
        <= stop3; --
removed, (RV 15/04/2019)
    tx_fsm
        <= idle; -- tx_end
active only for one fast clock (clk) period, (RV 15/04/2019)

    -- Reset tx_end_signal
    --when stop3 =>
        -- Removed (RV 15/04/2019)
        -- Send Stop Bit
        -- tx_end
            <= '0';
        -- tx
            <=
UART_IDLE;
    -- tx_fsm
        <= idle;

    -- Invalid States
    when others => null;
    end case;
    end if;
end if;
end if;
end process;

rx_debounceer:process(clk)
    variable deb_buf : std_logic_vector(3 downto 0);
begin
    if clk'event and clk = '1' then
        -- Debounce logic
        if deb_buf = "0000" then
            rx_data_deb
                <= '0';
        elsif deb_buf = "1111" then
            rx_data_deb
                <= '1';
        end if;
    end if;

```

```

-- Data storage to debounce
deb_buf          :=      deb_buf(2 downto 0) & rx;
end if;
end process;

rx_clk_gen:process(clk)
variable counter      :      integer range 0 to
conv_integer(((CLK_FREQ*1_000_000)/(SER_FREQ*16))-1);
begin
if (clk'event and clk = '1') then
    -- Normal Operation
    if counter = ((CLK_FREQ*1_000_000)/(SER_FREQ*16))-1 then
        rx_clk_en     <=      '1';
        counter       :=      0;
    else
        rx_clk_en     <=      '0';
        counter       :=      counter + 1;
    end if;
    -- Reset condition
    if rst = RST_LVL then
        rx_clk_en     <=      '0';
        counter       :=      0;
    end if;
end if;
end process;

-- Serial data Receiver
rx_proc:process(clk)
begin
if clk'event and clk = '1' then
    -- Default values
    rx_ready        <=      '0';
    -- Enable on UART rate
    if rx_clk_en = '1' then
        -- FSM description
        case rx_fsm is
            -- Wait to transfer data
            when idle =>

                if rx_data_deb = UART_START then
                    rx_fsm        <=      ck_start_bit;
                end if;
                rx_par_bit     <=      '0';
                rx_data_cnt   <=      (others=>'0');
                rx_clk_cnt    <=      (others=>'0');

            -- check rx line in the middle of the start bit
            when ck_start_bit =>

                if(rx_clk_cnt = 7)      then
                    if(rx_data_deb = UART_START) then
                        rx_fsm        <=      data;

```

```

        rx_clk_cnt          <= (others=>'0');
      else
        rx_fsm <= idle;
      end if;
    else
      rx_clk_cnt <= rx_clk_cnt + 1;
    end if;
-- Data receive
when data =>

  if(rx_clk_cnt = 15)      then
    rx_clk_cnt          <= (others=>'0');

    -- Check data to generate parity
    if par_en = PARITY_EN then
      rx_par_bit          <=      rx_par_bit
xor rx;
    end if;

    if rx_data_cnt = 7 then
      -- Data path
      rx_data(7)           <=
rx;
      rx_data(6 downto 0)  <= rx_data_tmp(7
downto 1);

      -- With parity verification
      if par_en = PARITY_EN then
        rx_fsm          <=      parity;
      -- Without parity verification
      else
        rx_ready          <=      '1';
        rx_fsm          <=      idle;
      end if;
    else
      rx_data_tmp          <=      rx &
rx_data_tmp(7 downto 1);
      rx_data_cnt          <=      rx_data_cnt
+ 1;
    end if;

    else
      rx_clk_cnt <= rx_clk_cnt + 1;
    end if;
  when parity =>
    -- Check received parity
    rx_fsm          <=      idle;
    if rx_par_bit = rx then
      rx_ready          <=      '1';
    end if;
  when others => null;

```

```
        end case;
        -- Reset condition
        if rst = RST_LVL then
            rx_fsm          <=    idle;
            rx_ready        <=    '0';
            rx_data         <=    (others=>'0');
            rx_data_tmp     <=    (others=>'0');
            rx_data_cnt     <=    (others=>'0');
            rx_clk_cnt      <=    (others=>'0');
        end if;
    end if;
end process;
end Behavioral;
```

## 4 PIC

I successivi capitoli/paragrafi sono dedicati in toto all'analisi e alla spiegazione dettagliata del codice/programma implementato sui vari dispositivi PIC del nostro sistema. Tali dispositivi hanno diversi ruoli e mansioni all'interno del nostro sistema:

- PIC Master:

Questo PIC ha il ruolo di supervisore del network CAN che è stato implementato, nonché di interfacciarsi con il Master di sistema (FPGA o Raspberry) al fine di ricevere/trasferire/interpretare comandi. I suoi compiti sono molti e verranno descritti in modo esaustivo nel proseguo del documento (assieme al codice implementativo), tuttavia se ne riporta qui (per i principali) una sommaria descrizione:

- Gestione/controllo Alive CAN:

Questo dispositivo ha il compito di effettuare controlli periodici circa lo stato di salute dei nodi connessi alla rete CAN, e qualora uno o più di essi risultino non funzionanti dovrà segnarli al Master di sistema;

- Gestione/controllo Alive EUSART:

È responsabile di fare il check periodico della comunicazione verso il Master di sistema, eventualmente cambiando canale comunicativo per passare da Raspberry a FPGA (o viceversa) se il caso lo richiedesse;

- Ricezione richieste d'accesso:

È responsabile di ricevere (attraverso un ben preciso protocollo-pacchetto dati) le richieste d'accesso agli ambienti di cui in nostro sistema si fa garante e traferirle al nodo interessato;

Come detto in precedenza qui sono citati unicamente i compiti principali, il dispositivo ovviamente deve essere in grado di svolgere svariate altre mansioni utili al corretto funzionamento del sistema, che saranno introdotte e trattate nel dettaglio nei capitoli successivi;

- PIC Slave:

Il nostro sistema conta un totale di 6 dispositivi PIC nel sistema, 5 dei quali sono denominati e svolgono il ruolo di Slave. Questi 5 dispositivi sono (oltre al PIC Master) gli altri nodi del network CAN, ognuno di essi ha il compito di gestire un sottosistema (Porta + Accessori). Gli Slave sono stati numerati sulla base delle relative porte da gestire (lo Slave 5 è quello della porta principale che è dunque la porta 5). Di questi, quelli da 1 a 4 supervisionano le porte interne munite di un sensore di prossimità (per la rilevazione dello stato della porta), di un tastierino numerico per l'inserimento delle password d'accesso, svariati LED per poter restituire un feedback all'utente e una "serratura" che, comandata, permette di sbloccare o bloccare la porta. Ogni nodo è stato pensato per poter garantire l'accesso su richiesta solo a determinate condizioni e non troppo spesso, per motivi di sicurezza. Lo Slave numero 5 è invece leggermente diverso dai precedenti, essendo il gestore della porta di accesso principale. In quanto tale esso possiede un pool di accessori e funzioni ridotto rispetto agli altri slave (in quanto parte delle loro funzioni non servono o sono già svolte da altri dispositivi del sistema per questo nodo, ad esempio non è presente il tastierino, in quanto non serve). Gli Slave da 1 a 4, una volta ricevuta una richiesta d'accesso devono comunque interfacciarsi con quello della porta principale per chiedergli o di sbloccare la porta (accesso consentito) o segnalare che l'accesso è stato negato.

### Introduzione al software dispositivi PIC:

Questo paragrafo è dedicato alla “definizione” del software utilizzato per lo sviluppo del programma che implementa le funzioni precedentemente descritte nei dispositivi PIC del sistema e contiene una panoramica generale (ad alto livello) dello stesso.

Iniziamo col dire che per la programmazione è stato impiegato il programma MPLAB X IDE, versione 5.15, e come si noterà nel proseguo del documento si è scelto di scrivere l’intero codice utilizzando il linguaggio C. A tal scopo il compilatore impiegato è XC8 versione 2.05. Il PIC utilizzati sono del modello **PIC 18F4580**.

Ad ogni PIC del nostro sistema è stato dedicato un file/cartella di progetto MPLAB, dunque ogni PIC ha il proprio programma. Ognuno di questi programmi è stato suddiviso (in termini di codice) su più file, sia di tipo .c (Source file C) che .h (header file C) per ragioni che verranno spiegate a breve. Le successive figure (Fig. 4-1 e 4-2) riportano le viste espanso della struttura dei file di progetto di PIC Master e del PIC Slave 1.

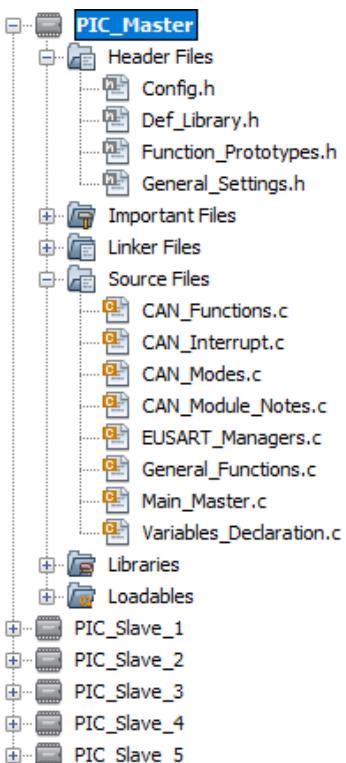


Fig. 4-1 Struttura Programma Master

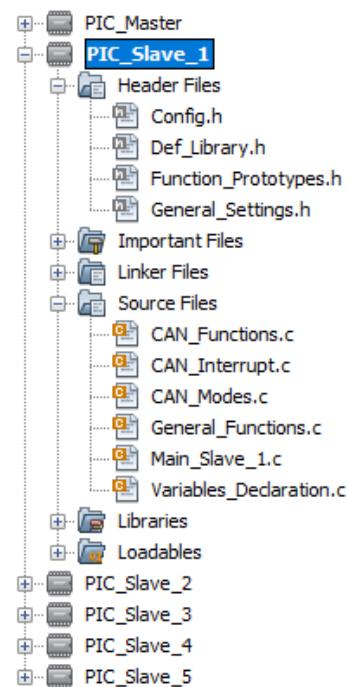


Fig. 4-2 Struttura Programma Slave 1

Come si può notare i due alberi dei file sono piuttosto simili, il PIC Master ne presenta uno più ricco in virtù del suo ruolo di supervisor della rete PIC e di tramite tra di essa ed il dispositivo Master del sistema (FPGA o Raspberry). Si noti inoltre che è stato riportato l’albero di progetto di uno solo degli Slave, in quanto, a parte differenze di codice che verranno trattate più avanti nel documento nelle opportune sezioni, essi presentano alberi (e struttura dei file) di progetto, assolutamente identici. Inoltre, a livello logico, il ruolo dei diversi file (aldilà del codice in essi contenuto) non varia tra nemmeno tra PIC Master e PIC Slave, infatti come si può notare, gli alberi presentano quasi gli stessi file.

Iniziamo con il considerare i file header:

- **Config.h:** Questo file è adibito al contenimento del codice relativo al settaggio dei Configuration bits del dispositivo considerato;
- **Def\_Library:** Contiene la ridefinizione di registri/bit-di-registri/pin (fatta attraverso la direttiva `#define`) atta a snellire il codice e a renderlo più leggibile, nonché la definizione delle variabili utilizzate. Si preannuncia che per scelte progettuali si è deciso di utilizzare tutte e sole variabili globali,

pertanto questo file contiene solo il richiamo alle variabili, tramite tipo *extern (extern type identifier)*, definite altrove in un altro file Source C. Ciò è stato fatto per permettere l'accesso a tali variabili a tutte le parti di programma, i cui file contengano l'inclusione di questo header;

- **Function\_Prootypes.h:** Contiene i prototipi di tutte le funzioni del programma;
- **General\_Settings.h:** Come sarà più chiaro in seguito, questo file contiene il codice utile per effettuare il settaggio della maggior parte dei registri del dispositivo;

Passiamo ai file Source C:

- **CAN\_Functions.c:** Contiene il corpo delle funzioni maggiormente legate all'attività del modulo CAN del PIC;
- **CAN\_Interrupt.c:** Il nome può apparire fuorviante, in quanto in realtà questo file contiene quasi tutti (tutti se consideriamo i PIC Slave) i blocchi codice da eseguire in risposta ad un interrupt di qualunque fonte. È stato rinominato usando la dicitura CAN per indicare il fatto che la stragrande maggioranza delle azioni da intraprendere a seguito di un interrupt coinvolgono in modo più o meno diretto anche il modulo CAN;
- **CAN\_Modes.c:** Contiene il corpo delle funzioni per la configurazione/riconfigurazione del modulo CAN;
- **General\_Functions.c:** Contiene il corpo di alcune funzioni di carattere generale, ossia non riferite in particolare ad un modulo ma utile al generale funzionamento del dispositivo;
- **Main\_"Nome\_Dispositivo".c:** Questo è, ovviamente, il file main del programma, diverso per ogni dispositivo. Rappresenta lo scheletro stesso del programma in quanto, come si vedrà nel prosieguo del documento, esso contiene relativamente poco codice, il cui ruolo è quasi solo chiamare funzioni definite altrove;
- **Variables\_Declaration.c:** Questo file contiene la dichiarazione e l'inizializzazione di tutte le variabili utilizzate nel progetto., ed è proprio il file Source C di cui si è parlato in precedenza;

In più nel progetto del dispositivo PIC Master:

- **EUSART\_Managers.c:** Contiene una serie di funzioni che utilizzano (si basano tu tale utilizzo) il modulo EUSART, al fine di implementare la connessione tramite protocollo RS-232 tra PIC Master e Master di sistema.
- **CAN\_Module\_Notes.c:** Questo file non contiene in realtà codice utile al funzionamento (solo codice commentato). È stato usato come archivio di registri e commenti su di essi in fase di sviluppo;

Questo conclude la presentazione della struttura del codice implementativo dei programmi. Nel prosieguo il codice vero e proprio verrà preso in considerazione ed analizzato.

#### Nota introduttiva di nomenclatura:

Nel seguito si potrà incontrare la dicitura *REGISTROn* (esempio: *TRISn*). Questa ricorrerà spesso, soprattutto nelle sezioni descrittive del documento e verrà usata per indicare/riferirsi a serie di registri che condividono un acronimo comune ma hanno un indice alfanumerico identificativo diverso. Ad esempio, la dicitura *PIEn* potrà essere utilizzata per riferirsi a TUTTI i registri di tipo *PIE* (*PIE1*, *PIE2*, *PIE3*). Motivo per cui nel testo, questo "parametro n" non sarà altro che un indice generico per indicare che in quella posizione dell'acronimo del nome del registro è di solito presente un numero (o una lettera in alcuni casi).

Allo stesso modo, alle volte ci si riferirà a bit (contenuto nel medesimo registro) aventi lo stesso acronimo ma differente numerazione (perché parte di un set di settaggio/configurazione) con la dicitura *BITn*. Ad esempio, *IRCFn* fa riferimento alla serie di 3 bit *IRCF0*, *IRCF1*, *IRCF2*) contenuta nel registro di controllo dell'oscillatore *OSCCON*, il cui valore (nel suo insieme) è utilizzato per impostare la frequenza di oscillazione.

## 4.1 PIC Master

### 4.1.1 Main Master

Cominciamo l'analisi del programma del PIC Master prendendo in considerazione il suo file main (**Main\_Master.c**).

Codice MPLAB:

```
#include "Def_Library.h"
#include "Config.h"
#include "General_Settings.h"
#include "Function_Protoypes.h"

void main(void)
{
    //Eseguo il reset iniziale del dispositivo
    Initial_Reset();

    //Configuro i registri generici
    general_settings();

    //Configuro il modulo CAN
    CAN_Config_Mode();

    //Configuro il CAN per la modalità normale
    CAN_Normal_Mode();

    //Delay per dar tempo a tutti gli Slave di configurarsi
    __delay_ms(50);

    //Funzione di test iniziale
    Initial_Test();

    Switch_manager(); //Effettuo il primo switch per settare il collegamento col raspberry all'avvio

    while (1)
    {
        //EUSART Routines
        if (Timer3OverflowFlag==1)
            {//Se i due registri del timer3 hanno raggiunto l'overflow...
                EUSART_Alive_check(); //Controllo se sono passati 8 secondi dall'ultimo ALIVE ricevuto; se sì,
                effettuo lo switch
            }

        if (SwitchStatusDisplay==1)
        {
            EUSART_Status_Display(); //Se ci sono state variazioni nei canali di comunicazione dello
            switch, lo mostro
        }

        if (read_RCREG==1)
```

```

{ //Se si è attivata la routine di interrupt...
    EUSART_FSM(); //Controllo se ho ricevuto un carattere; se sì, attivo la macchina a stati per la
    gestione del pacchetto dati
}

//CAN Routines

if (Alive_Check==1)
{
    if (Check_Req==1)
    {
        Are_you_there(); //Chiamo la routine di invio Alive
    }
}

if (Broken_Door==1)
{
    Broken_Door_Signal(); //Chiamo la routine di trasmissione porta rotta
    Broken_Door=0; //Azzero la variabile di stato
}

if (A_Door_is_Open==1)
{
    //Devo comunicare a Raspeberry/FPGA di bloccare gli accessi
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xDD; //Trasmetto il codice di porta aperta
    A_Door_is_Open=0; //Azzero la variabile
}

if (CAN_Invalid_Msg==1)
{
    if (Counter_Inv_Msg<10)
    {
        LEDRedCAN=1;
        __delay_ms(100);
        LEDRedCAN=0;
        __delay_ms(100);
        LEDRedCAN=1;
        __delay_ms(100);
        LEDRedCAN=0;
        __delay_ms(100);
        CAN_Invalid_Msg=0;
    }
    else
    {
        Counter_Inv_Msg=0; //Azzero il counter
        //Devo comunicare a Raspeberry/FPGA che è stato ricevuto (o trasmesso) ripetutamente un
        messaggio non valido
        while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    }
}

```

```

    TXREG=0xE1;      //Trasmetto il codice di "troppi messaggi non validi", Errore di Livello 1
    CAN_Invalid_Msg=0;
}
IRXIE=1; //Riattivo la possibilità di interrupt
}
if (CAN_Error_Check==1)
{
    CAN_Error_Handling();
}
}

void __interrupt() ISR(void) //Routine di interrupt
{
    di(); //Disattivo tutti gli interrupt
    CAN_ISR();

    //EUSART interrupt
    if(PIR1bits.RCIF==1)
    {
        //Il buffer RCREG è pieno:...
        read_RCREG=1; //Attivo la routine di verifica del carattere sul main...
        incoming_data=RCREG;
    }

    if(PEIE==0)
    {
        PEIE=1; //Riattivo gli interrupt periferici perchè all'azzeramento di GIE, viene azzerato
    }
}

}//Fine interrupt service routine

```

Non ci si soffermerà ora nella trattazione di tutte le funzioni esposte, cosa che verrà invece svolta nel seguito del documento, verranno però date alcune informazioni di carattere generale atte a comprendere meglio come è stato organizzato il programma.

Partendo dall'apice del codice, possiamo notare che le prime righe non sono altro che delle direttive di inclusione per file header. In questi file sono contenute diverse cose, in particolare:

- **Def\_Library.h**: Come detto in precedenza, contiene l'intero archivio di tutte le definizioni delle variabili (globali) del programma (dichiarate ed inizializzate nell'apposito file Source C) e la definizione degli shortcut (ossia ridefinizioni di variabili/pin/registri/bit interni a MPLAB tramite la direttiva `#define`) utilizzati per rendere più leggibile il codice e velocizzarne ovviamente la scrittura. Ad esempio, `TRMT` rappresenta lo shortcut per il bit `TXSTAbits.TRMT`.

Il codice completo di questo file header è contenuto in appendice al documento. Si tenga inoltre a mente che questo file contiene anche (in apice) le seguenti:

```
#define _XTAL_FREQ 1000000
#include <xc.h>
```

```
#include <stdio.h>
```

La prima riga serve a fornire al simulatore di sistema l'informazione riguardante la frequenza di oscillazione del dispositivo, la seconda e la terza invece fanno sì che vengano incluse due ulteriori librerie di progetto fondamentali per il corretto funzionamento del programma.

- **Config.h**: Contiene il codice per il settaggio dei Configuration Bits (verrà trattato nel seguito);
- **General\_Settings.h**: Contiene i settaggi generali del dispositivo (verrà trattato nel seguito);
- **Function\_Prototypes.h**: Contiene i prototipi di tutte le funzioni utilizzate nel programma (codice disponibile in appendice al documento);

Dato il loro contenuto, gli header **Function\_Prototypes.h** e **Def\_Library.h** sono stati in realtà inclusi in tutti i file di progetto che contengono le definizioni vere e proprie delle funzioni chiamate nel main.

Osservando il codice sopra-riportato contenuto nel ciclo **while**, si può notare come, a conti fatti, il file main del PIC Master non sia altro che una macchina a stati governata da una serie di variabili o flag di stato che vengono settati a seguito di determinati accadimenti, e in conseguenza dei quali è necessario prendere provvedimenti. Il ciclo **while** ha anche lo scopo di non far mai uscire il programma dalla funzione main, che quindi opererà ciclicamente fino allo spegnimento del dispositivo o al suo reset.

Il settaggio delle variabili/flag citati in precedenza avviene tramite l'utilizzo di segnali di interrupt, i quali fanno saltare il programma alla routine:

```
void __interrupt() ISR(void) //Routine di interrupt
```

Riportata in fondo al file main e che contiene poi opportune funzioni di cui si discuterà nel seguito atte a gestire le diverse fonti di interrupt.

L'ultima informazione di carattere generale che ha importanza evidenziare riguarda l'uso che è stato fatto di alcune funzioni predefinite del compilatore C di MPLAB:

- **\_delay\_ms**: è una funzione che permette di inserire un ritardo, ossia "trattiene" il programma a quella riga fino a quando il tempo in millisecondi specificato come argomento della funzione non è trascorso. Per l'uso di questa funzione è essenziale fornire al simulatore l'informazione riguardante la frequenza di clock del dispositivo (che poi andrà settata attraverso l'uso di opportuni registri di controllo) tramite la direttiva **\_XTAL\_FREQ** accennata in precedenza. Come si può notare subito prima della chiamata alla funzione di "Initial\_Test()" è presente un brevissimo delay, che è stato introdotto nel solo programma del PIC Master per assicurarsi che tutti gli Slave della rete abbiano il tempo di configurarsi completamente;
- **di()**: è una funzione che disattiva gli interrupt del dispositivo. Aprendone il file sorgente si può osservare che essa si limita ad azzerare il bit **GIE** (bit di abilitazione generale interrupt) del registro di controllo **INTCON**, che poi viene poi automaticamente settato nuovamente ad 1 dal programma, in uscita dalla routine di gestione interrupt. In realtà ci si è accorti sperimentalmente che disattivare il bit **GIE** provoca l'azzeramento anche del bit **PEIE** (bit di abilitazione interrupt periferici), motivo per cui è stato previsto il seguente blocco al termine della routine di **ISR**:

```
if(PEIE==0)
{
    PEIE=1; //Riattivo gli interrupt periferici perchè all'azzeramento di GIE, viene azzerato
}
```

Che si occupa, qualora fosse azzerato, di settarlo nuovamente al valore 1.

#### 4.1.2 Configuration bits

Il primo passo che è stato svolto per la programmazione dei dispositivi PIC usati nel sistema in esame, ha riguardato la scelta dei bit di configurazione dei dispositivi. Questo è stato fatto settando gli opportuni valori (discussi nel seguito) nell'apposita sezione di MPLAB e producendo poi il codice C corrispondente, contenuto per comodità e chiarezza nell'apposito file header **Config.h**, incluso poi nel file main di ogni PIC.

Codice MPLAB:

```
// PIC18F4580 Configuration Bit Settings

// 'C' source line config statements

// CONFIG1H
#pragma config OSC = IRCIO7      // Oscillator Selection bits (Internal oscillator block, CLKO function on RA6, port function on RA7)
#pragma config FCMEN = OFF        // Fail-Safe Clock Monitor Enable bit (Fail-Safe Clock Monitor disabled)
#pragma config IESO = OFF         // Internal/External Oscillator Switchover bit (Oscillator Switchover mode disabled)

// CONFIG2L
#pragma config PWRT = OFF        // Power-up Timer Enable bit (PWRT disabled)
#pragma config BOREN = OFF        // Brown-out Reset Enable bits (Brown-out Reset disabled in hardware and software)
#pragma config BORV = 3           // Brown-out Reset Voltage bits (VBOR set to 2.1V)

// CONFIG2H
#pragma config WDT = OFF          // Watchdog Timer Enable bit (WDT disabled (control is placed on the SWDTEN bit))
#pragma config WDTPS = 128         // Watchdog Timer Postscale Select bits (1:128)

// CONFIG3H
#pragma config PBADEN = OFF       // PORTB A/D Enable bit (PORTB<4:0> pins are configured as digital I/O on Reset)
#pragma config LPT1OSC = OFF       // Low-Power Timer 1 Oscillator Enable bit (Timer1 configured for higher power operation)
#pragma config MCLRE = OFF         // MCLR Pin Enable bit (RE3 input pin enabled; MCLR disabled)

// CONFIG4L
#pragma config STVREN = ON         // Stack Full/Underflow Reset Enable bit (Stack full/underflow will cause Reset)
#pragma config LVP = OFF           // Single-Supply ICSP Enable bit (Single-Supply ICSP disabled)
#pragma config BBSIZ = 1024         // Boot Block Size Select bit (1K words (2K bytes) boot block)
#pragma config XINST = OFF          // Extended Instruction Set Enable bit (Instruction set extension and Indexed Addressing mode disabled (Legacy mode))

// CONFIG5L
#pragma config CP0 = OFF           // Code Protection bit (Block 0 (000800-001FFFh) not code-protected)
#pragma config CP1 = OFF           // Code Protection bit (Block 1 (002000-003FFFh) not code-protected)
#pragma config CP2 = OFF           // Code Protection bit (Block 2 (004000-005FFFh) not code-protected)
#pragma config CP3 = OFF           // Code Protection bit (Block 3 (006000-007FFFh) not code-protected)
```

```

// CONFIG5H
#pragma config CPB = OFF      // Boot Block Code Protection bit (Boot block (000000-0007FFh) not code-protected)
#pragma config CPD = OFF      // Data EEPROM Code Protection bit (Data EEPROM not code-protected)

// CONFIG6L
#pragma config WRT0 = OFF     // Write Protection bit (Block 0 (000800-001FFFh) not write-protected)
#pragma config WRT1 = OFF     // Write Protection bit (Block 1 (002000-003FFFh) not write-protected)
#pragma config WRT2 = OFF     // Write Protection bit (Block 2 (004000-005FFFh) not write-protected)
#pragma config WRT3 = OFF     // Write Protection bit (Block 3 (006000-007FFFh) not write-protected)

// CONFIG6H
#pragma config WRTC = OFF     // Configuration Register Write Protection bit (Configuration registers (300000-3000FFh) not write-protected)
#pragma config WRTB = OFF     // Boot Block Write Protection bit (Boot block (000000-0007FFh) not write-protected)
#pragma config WRTD = OFF     // Data EEPROM Write Protection bit (Data EEPROM not write-protected)

// CONFIG7L
#pragma config EBTR0 = OFF    // Table Read Protection bit (Block 0 (000800-001FFFh) not protected from table reads executed in other blocks)
#pragma config EBTR1 = OFF    // Table Read Protection bit (Block 1 (002000-003FFFh) not protected from table reads executed in other blocks)
#pragma config EBTR2 = OFF    // Table Read Protection bit (Block 2 (004000-005FFFh) not protected from table reads executed in other blocks)
#pragma config EBTR3 = OFF    // Table Read Protection bit (Block 3 (006000-007FFFh) not protected from table reads executed in other blocks)

// CONFIG7H
#pragma config EBTRB = OFF    // Boot Block Table Read Protection bit (Boot block (000000-0007FFh) not protected from table reads executed in other blocks)

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

```

Come si può notare dal codice soprastante, la maggior parte dei settaggi ha riguardato lo spegnimento/disattivazione di moduli/modalità non utilizzate e/o che potevano andare in conflitto con la nostra applicazione perché facenti riferimento a PIN multiplexati. In particolare, i settaggi che sono stati considerati e modificati rispetto ai settaggi di default sono:

- *config OSC = IRCIO7*: Con questo andiamo a impostare l'oscillatore interno del PIC come fonte di Clock per il dispositivo (in conseguenza sono stati settati *FCMEN* e *IESO*);
- *config PWRT = OFF*: Disattivato perché inutilizzato;
- *config BOREN = OFF*: Disattivato perché inutilizzato (in conseguenza *BORV* è stato lasciato al valore di default);
- *config WDT = OFF*: Disattivato perché inutilizzato (in conseguenza *WDTPS* è stato lasciato al valore di default);

- *config LPT1OSC = OFF*: Disattivato in quanto il nostro progetto non ha preso in considerazione la possibilità di mandare i dispositivi del sistema “a riposo”, ossia in questa sede si è deciso di non considerare il problema dell’ottimizzazione dei dispendi energetici (si veda il capitolo 5. *Conclusioni e considerazioni di progetto* per ulteriori informazioni a riguardo);
- *config PBADEN = OFF*: Permette di utilizzare le porte B come in/out digitali e non analogici;
- *config MCLRE = OFF*: In fase di programmazione PIC, MPLAB automaticamente richiede l’utilizzo del PIN RE3 come PIN di programmazione, tuttavia ci siamo voluti riservare la possibilità di usarlo al di fuori della programmazione come ulteriore PIN di I/O;
- *config STVREN = ON*: Per motivi di sicurezza/robustezza del sistema abbiamo scelto di abilitare il reset del dispositivo qualora (per qualunque ragione/bug) il buffer del Program Counter dovesse riempirsi e generare overflow;
- *config LVP = OFF*: Disattivato per poter usufruire di un PIN aggiuntivo (RB5);
- *BBSIZ* e *XINST*: Sono stati lasciati al valore assegnato di default;
- Infine (per brevità non saranno riportati, ma sono comunque visibili nel codice sopraståte), sono stati disattivati tutti i bit di protezione dei registri in modo da renderli scrivibili/leggibili a seconda dei casi;

Si tenga comunque presente che per il settaggio di questi bits ci si è riferiti espressamente [alla sezione 25 del datasheet del dispositivo](#). Come si può infine notare non è presente l’istruzione di `#include <xc.h>`, questo perché essa è contenuta in un altro file (**Def\_Library.h**).

Si tenga inoltre presente che i precedenti settaggi sono stati usati per TUTTI i PIC del sistema considerato.

#### 4.1.3 Funzione Initial\_Reset()

La prima azione svolta da tutti i dispositivi PIC del sistema (Slave e Master) a seguito dell'accensione è quella di fare una chiamata nel main alla funzione “Initial\_Reset”.

//Eseguo il reset iniziale del dispositivo

*Initial\_Reset();*

Il cui prototipo è contenuto nel nell'apposito file header dei prototipi (**Function\_Prootypes.h**) e il/la cui corpo/definizione è invece contenuto/a nel file adibito alle funzioni “generali” del sistema (**General\_Functions.c**).

Codice MPLAB:

```
void Initial_Reset(void)
{
    if (RCONbits.NOT_RI==1)
    {
        RESET(); //Resetto il dispositivo
    }
    else
    {
        //Reset manuale dei registri non resettati al power-on o dopo comando

        //Reset manuale dei registri di trasmissione
        //TXB0
        TXB0DLC=0;
        TXB0EIDL=0;
        TXB0EIDH=0;
        TXB0SIDL=0;
        TXB0SIDH=0;
        TXB0DO=0;
        TXB0D1=0;
        TXB0D2=0;
        TXB0D3=0;
        TXB0D4=0;
        TXB0D5=0;
        TXB0D6=0;
        TXB0D7=0;
        //TXB1
        TXB1DLC=0;
        TXB1EIDL=0;
        TXB1EIDH=0;
        TXB1SIDL=0;
        TXB1SIDH=0;
        TXB1D0=0;
        TXB1D1=0;
        TXB1D2=0;
        TXB1D3=0;
        TXB1D4=0;
        TXB1D5=0;
        TXB1D6=0;
        TXB1D7=0;
```

```

//TXB2
TXB2DLC=0;
TXB2EIDL=0;
TXB2EIDH=0;
TXB2SIDL=0;
TXB2SIDH=0;
TXB2D0=0;
TXB2D1=0;
TXB2D2=0;
TXB2D3=0;
TXB2D4=0;
TXB2D5=0;
TXB2D6=0;
TXB2D7=0;

//Reset manuale dei registri di ricezione
//RXBO
RXB0DLC=0;
RXB0EIDL=0;
RXB0EIDH=0;
RXB0SIDL=0;
RXB0SIDH=0;
RXB0D0=0;
RXB0D1=0;
RXB0D2=0;
RXB0D3=0;
RXB0D4=0;
RXB0D5=0;
RXB0D6=0;
RXB0D7=0;
//RXB1
RXB1DLC=0;
RXB1EIDL=0;
RXB1EIDH=0;
RXB1SIDL=0;
RXB1SIDH=0;
RXB1D0=0;
RXB1D1=0;
RXB1D2=0;
RXB1D3=0;
RXB1D4=0;
RXB1D5=0;
RXB1D6=0;
RXB1D7=0;

//Reset manuale dei registri di maschere e filtri messaggi CAN
//Maschere
RXM0EIDL=0;

```

```

RXMOEIDH=0;
RXMOSIDL=0;
RXMOSIDH=0;
RXM1EIDL=0;
RXM1EIDH=0;
RXM1SIDL=0;
RXM1SIDH=0;
//Filtri
RXFOEIDL=0;
RXFOEIDH=0;
RXFOSIDL=0;
RXFOSIDH=0;
RXF1EIDL=0;
RXF1EIDH=0;
RXF1SIDL=0;
RXF1SIDH=0;
RXF2EIDL=0;
RXF2EIDH=0;
RXF2SIDL=0;
RXF2SIDH=0;
RXF3EIDL=0;
RXF3EIDH=0;
RXF3SIDL=0;
RXF3SIDH=0;
RXF4EIDL=0;
RXF4EIDH=0;
RXF4SIDL=0;
RXF4SIDH=0;
RXF5EIDL=0;
RXF5EIDH=0;
RXF5SIDL=0;
RXF5SIDH=0;

//Reset manuale delle porte
//PORTA
PORTA=0;
LATA=0;
TRISA=0;
//PORTB
PORTB=0;
LATB=0;
TRISB=0;
//PORTC
PORTC=0;
LATC=0;
TRISC=0;
//PORTD
PORTD=0;

```

```

LATD=0;
TRISD=0;
//PORTE
PORTE=0;
LATE=0;
TRISE=0;

//Reset manuale dei registri di priorità interrupt
IPR1=0;
IPR2=0;
IPR3=0;

//Reset manuale dei registri INT
INTCON=0;
INTCON2=0;
INTCON3=0;

//Reset manuale dei timer
T0CON=0;
TMR0L=0;
TMR1ON=1;
TMR1H=0;
TMR1L=0;
TMR1ON=0;
TMR3ON=1;
TMR3H=0;
TMR3L=0;
TMR3ON=0;

//Reset manuale variabili di processo

//Variabili globali utili per gestione degli stati del sistema
A_Door_is_Open=0; //Variabile associata alla rilezione di porta aperta da parte del sensore magnetico
Alive_Check=0; //Variabile associata all'avvio della routine di Alive
Check_Slave=0; //Variabile di stato associata alla routine di Alive
Check_Req=0; //Variabile utile per effettuare il check sicuro all'interno del ciclo di Alive
Broken_Door=0; //Variabile di stato per identificare la condizione di porta rotta
CAN_Invalid_Msg=0; //Variabile usata per gestire la situazione di messaggio CAN non valido
Counter_Inv_Msg=0; //Contatore per il numero di Messaggi CAN errati
CAN_Error_Check=0; //Variabile che abilita il Check degli errori CAN

//Contatori per implementazione handling errori CAN
RXBOVLF_Counter=0;

//Variabili di stato associate allo stato dei singoli PIC Slave (Fintanto che la variabile è 0 lo slave è funzionante)
Slave1_status=0; //Slave 1

```

```

Slave2_status=0;
Slave3_status=0;
Slave4_status=0;
Slave5_status=0;

//Variabili associate all'avvenuta (o meno) trasmissione della condizione di
malfunzionamento degli slave
S1_st_Transmit=0; //Slave 1
S2_st_Transmit=0;
S3_st_Transmit=0;
S4_st_Transmit=0;
S5_st_Transmit=0;

//Variabili EUSART
FSM_Status=0; //Variabile che definisce lo stato
incoming_data=0; //Variabile che memorizza il dato in arrivo sul registro RCREG
read_RCREG=0; //Flag che controlla la routine di gestione dei dati in arrivo
counter=0; //Variabile che definisce il valore del contatore relativo al timer3
switches=0; /*Variabile che tiene conto del numero consecutivo di switch avvenuti senza
l'invio di un ALIVE*/
SwitchStatusDisplay=0; // Variabile di stato della funzione di display stato connessioni switch
error_flag=0; //Flag che attiva la routine di errore
data_number=0; //Contatore relativo al numero di byte di dati ricevuti durante un pacchetto
dati
Timer3OverflowFlag=0; //Variabile di stato associata alla funzione di conteggio Overflow
Timer3
Raspberry=1; //Variabile di stato del Raspberry
FPGA=0; //Variabile di stato dell'FPGA
Double_Switch=0; //Variabile di stato associata al doppio switch
}

}

```

Lo scopo di questa funzione altro non è che effettuare un reset preliminare del dispositivo. Questa necessità deriva dal fatto che, come spiegato nel datasheet del dispositivo (in allegato al materiale del progetto) nella sezione 5.0: Reset esistono varie opzioni per il reset, ed ognuna di esse ha effetti diversi sui valori contenuti nei registri. Il Power-On Reset in particolare, reset che avviene quando la tensione di alimentazione Vdd supera una certa soglia (pagina 49/490 sezione 5.3 datasheet), porta diversi registri ad assumere valori randomici e/o invariati rispetto alla precedente sessione d'uso del dispositivo (per maggiori dettagli su quali e quali valori assumano i registri di rimanda per brevità al datasheet sezione 5.7 da pagina 55 a 66), per questo motivo, come si evince dal codice, si è scelto di seguire una procedura di reset iniziale articolata in tre passaggi:

**TABLE 5-3: STATUS BITS, THEIR SIGNIFICANCE AND THE INITIALIZATION CONDITION FOR RCON REGISTER**

Condition	Program Counter <sup>(1)</sup>	RCON Register						STKPTR Register	
		SBOREN	RI	TO	PD	POR	BOR	STKFUL	STKUNF
Power-on Reset	0000h	1	1	1	1	0	0	0	0
RESET Instruction	0000h	u <sup>(2)</sup>	0	u	u	u	u	u	u
Brown-out Reset	0000h	u <sup>(2)</sup>	1	1	1	u	0	u	u
MCLR Reset during Power-Managed Run modes	0000h	u <sup>(2)</sup>	u	1	u	u	u	u	u
MCLR Reset during Power-Managed Idle modes and Sleep mode	0000h	u <sup>(2)</sup>	u	1	0	u	u	u	u
WDT Time-out during Full Power or Power-Managed Run modes	0000h	u <sup>(2)</sup>	u	0	u	u	u	u	u
MCLR Reset during Full-Power execution	0000h	u <sup>(2)</sup>	u	u	u	u	u	u	u
Stack Full Reset (STVREN = 1)	0000h	u <sup>(2)</sup>	u	u	u	u	u	1	u
Stack Underflow Reset (STVREN = 1)	0000h	u <sup>(2)</sup>	u	u	u	u	u	u	1
Stack Underflow Error (not an actual Reset, STVREN = 0)	0000h	u <sup>(2)</sup>	u	u	u	u	u	u	1
WDT Time-out during Power-Managed Idle or Sleep modes	PC + 2	u <sup>(2)</sup>	u	0	0	u	u	u	u
Interrupt Exit from Power-Managed modes	PC + 2	u <sup>(2)</sup>	u	u	0	u	u	u	u

Legend: u = unchanged

Note 1: When the wake-up is due to an interrupt and the GIEH or GIEL bits are set, the PC is loaded with the interrupt vector (008h or 0018h).

2: Reset state is '1' for POR and unchanged for all other Resets when software BOR is enabled (BOREN<1:0> Configuration bits = 01 and SBOREN = 1); otherwise, the Reset state is '0'.

*Fig. 4.1.3-3 Tabella valori dei bit RCON (datasheet pag.54/490)*

- Avviene il Power-On Reset a seguito dell'accensione;
- La funzione “Initial\_Reset()” fa il check nel registro **RCON** (che contiene i bit di status dei reset) del bit **NOT\_RI** (bit di stato associato ai reset da riga di comando), e nel caso esso abbia valore 1 (ovvero non si è verificato alcun reset di quel tipo), allora viene avviene il reset del dispositivo da riga di comando(**RESET()**). Per completezza è stata riportata una tabella che riporta schematicamente lo stato dei bit di **RCON** a seguito delle condizioni di Reset verificatesi (Fig. 4.1.3-1);

#### REGISTER 5-1: RCON: RESET CONTROL REGISTER

R/W-0	R/W-1 <sup>(1)</sup>	U-0	R/W-1	R-1	R-1	R/W-0 <sup>(2)</sup>	R/W-0
IPEN	SBOREN	—	RI	TO	PD	POR	BOR
bit 7							bit 0

*Fig. 4.1.3-4 RCON Register (datasheet pag.48/490)*

3. Infine, dopo il reset da riga di comando la funzione eseguirà la sezione di codice contenuta nell'istruzione `else` in quanto a seguito del suddetto il bit `NOT_RI` avrà cambiato il proprio valore, motivo per cui la condizione richiesta dall'istruzione `if` non sarà più soddisfatta. Nella suddetta sezione viene effettuato l'azzeramento manuale di tutti quei registri (quelli di nostro interesse) non precedentemente azzerati (o i cui valori risultavano incerti).

È importante sottolineare, (come detto in precedenza) che questa procedura di reset è stata implementata su TUTTI i dispositivi PIC del sistema, indipendentemente dal loro ruolo. Questo è importante anche perché, come si nota dal codice sopraffatto, viene fatto in ultimo anche il reset delle variabili di stato dei processi/programmi del dispositivo (le quali saranno diverse in base al ruolo ricoperto). Inoltre, si noti che viene effettuato anche il reset dei contatori (Timer) del dispositivo, per la cui descrizione si rimanda al seguito della trattazione.

#### 4.1.4 Funzione general\_settings() (Master)

Eseguito il reset iniziale del dispositivo ora si devono eseguire i settaggi e le configurazioni necessarie al funzionamento del dispositivo. Si anticipa fin da ora che si è tentato di mantenere i settaggi di tutti i dispositivi PIC il più simili possibile, pertanto quando verrà il momento di commentare i settaggi dei PIC diversi dal Master si evidenzieranno unicamente le differenze, qualora dovessero essercene.

Nel file main del dispositivo Master viene quindi effettuata una chiamata alla funzione “general\_settings”:

```
//Configuro i registri generici  
general_settings();
```

Il cui corpo è contenuto in un secondo file header denominato **General\_Settings.h**, incluso in apice al main.

Codice MPLAB:

```
void general_settings(void) //Funzione dei settaggi  
{  
    //Inizializzazione PORT  
  
    //PORTB  
    TRISB=0b00001000; //Setto tutto le porte, in particolare RB2 dovrà essere  
                      //usata come CANTX (viene settata in automatico quando  
                      //si inizializzano i regitri CAN, mentre RB3 va manualmente  
                      //settato come input (1) in modo da essere il CANRX)  
    LATB=0x00;  
  
    //PORTA  
    LATA=0x00;  
    TRISA=0; //Setto tutti i pin del PORT A come output  
  
    //PORTC  
    LATC=0x00;  
    TRISC=0;  
    TRISCbits.RC7=1; //RC7 input perché fa le veci di RX  
    TRISCbits.RC6=0; //RC6 output perché fa le veci di TX  
  
    //PORTD  
    LATD=0x00;  
    TRISD=0x00;  
  
    //Settaggi aggiuntivi  
    //I PIN delle porte A sono analogiche di default, il nostro sistema gestisce segnali digitali  
    ADCON1=0b00001111; //Setto tutti i pin analogici come digitali  
    ADCON0bits.ADON=0; //Spengo il convertitore A/D  
  
    //Inizializzazione Registro di clock interno del sistema (datasheet 36/490)  
    //Frequenza di lavoro a 1Mhz  
    OSCCON=0b01000110;  
  
    //Disattivo il modulo CCP (datasheet 167/490)  
    CCP1CON=0b00000000; //Usa solo gli ultimi 4 bit  
  
    //Disabilito la priorità degli interrupt
```

```

RCONbits.IPEN=0;

//Configuro il Timer0 (datasheet slide 151/490)
T0CON=0b10000111;
//Overflow in circa 67 secondi

//Configuro il Timer1 (datasheet slide 155/490)
T1CON=0b10110000;
//Overflow in circa 2 secondi

//Configuro il Timer3 (datasheet slide 163/490)
T3CON=0b10110001; //Attivo il timer3, modalità a 16 bit, clock come sorgente, prescaler 1:8

//Configurazione Modulo EUSART per comunicazione RS-232
RCSTA=0b10010000; //Porte TX e RX abilitate, ricezione 8n1, ricezione dati abilitata
TXSTA=0b00100100; //Modalità asincrona, baud rate high speed, trasmissione 8n1, trasmissione dati
abilitata
BAUDCON=0b00001000; //Baud rate definito dai due registri SPBRGH e SPBRG
SPBRGH=0;
SPBRG=25; //Definisco un baud rate pari a 9600 per Fosc=1MHz (vedi tabella pag. 238 del datasheet)

//Setto i registri di ricezione per i messaggi CAN
RXB1CON=0b00100000;
RXB0CON=0b00100000;

//Impostazione/abilitazione interrupt
INTCON=0b11100000; //Abilito anche il timer0 (che userò per il messaggio di alive)

//Registro di abilitazione interrupt periferici 1
PIE1=0b00100001; // (datasheet slide 127/490)
//bit 0: Enable dell'interrupt di overflow Timer1
//bit 4: Disabilito l'interrupt sulla trasmissione dati modulo EUSART
//bit 5: Abilito l'interrupt sulla ricezione dati modulo EUSART

//Registro di abilitazione interrupt periferici 2
PIE2=0b00000010; // (datasheet slide 128/490)

//Registro di abilitazione interrupt periferici (CAN)
PIE3=0b10100001; //(datasheet slide 129/490 o 322/490)
//I bit 7,6,5 settati a 0 disattivano i 3 interrupt CAN di (IRXIE, WAKIE, ERRIE)
//I bit 1 e 0 gestiscono gli interrupt di ricezione rispettivamente sui Buffer CAN 1 e 0
//Per ora ho abilitato solo l'interrupt del registro di ricezione 0
}

```

Passando ora alla valutazione del codice sopra riportato si può notare come i primi settaggi svolti riguardino i registri dei pin I/O del dispositivo, in particolare viene fatto uso dei registri:

1. **TRISn**: Permettono l'impostazione dei pin come input (valore bit=1) o output (valore bit=0);

- 1.1. Nel caso del PIC Master gli unici pin impostati come input sono i pin relativi al CANRX (canale di ricezione CAN, **RB3**) ed alla ricezione seriale (ossia il pin **RC7**, che funge da pin di ricezione per il modulo EUSART);
- 1.2. Di default, tutti i pin non settati esplicitamente come input, sono di output;
2. **LATn**: Permettono di svolgere le operazioni di lettura/scrittura dati da e/o verso il pin desiderato;
- 2.1. Sono inizializzati a 0 (sono normalmente utilizzati per la scrittura e l'invio di dati);

Subito dopo le porte troviamo i settaggi relativi al modulo di conversione A/D, che nel nostro caso in realtà consistono fondamentalmente in una sua disattivazione dopo aver reso digitali tutti i pin del dispositivo (in quanto nel nostro progetto non è previsto il trasferimento di dati/valori analogici):

**ADCON1**=0b00001111; //Setto tutti i pin analogici come digitali

**REGISTER 20-2: ADCON1: A/D CONTROL REGISTER 1**

U-0	U-0	R/W-0	R/W-0	R/W-0 <sup>(1)</sup>	R/W-q <sup>(1)</sup>	R/W-q <sup>(1)</sup>	R/W-q <sup>(1)</sup>
—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

Fig. 4.1.4-1 Registro ADCON1 (datasheet slide 254/490)

**ADCON0**bits.**ADON**=0; //Spengo il convertitore A/D

**REGISTER 20-1: ADCON0: A/D CONTROL REGISTER 0**

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/DONE	ADON	
bit 7								bit 0

Fig. 4.1.4-2 Registro ADCON0 (datasheet slide 253/490)

La prima delle due precedenti righe di codice risulta particolarmente importante in quanto, all'accensione del dispositivo diversi pin sono di default settati come analogici. Nel seguito si riporta la tabella relativa al settaggio dei primi 4 bit del registro **ADCON1**.

bit 3-0

**PCFG<3:0>**: A/D Port Configuration Control bits:

PCFG<3:0>	AN10	AN9	AN8	AN7 <sup>(2)</sup>	AN6 <sup>(2)</sup>	AN5 <sup>(2)</sup>	AN4	AN3	AN2	AN1	AN0
0000 <sup>(1)</sup>	A	A	A	A	A	A	A	A	A	A	A
0001	A	A	A	A	A	A	A	A	A	A	A
0010	A	A	A	A	A	A	A	A	A	A	A
0011	A	A	A	A	A	A	A	A	A	A	A
0100	A	A	A	A	A	A	A	A	A	A	A
0101	D	A	A	A	A	A	A	A	A	A	A
0110	D	D	A	A	A	A	A	A	A	A	A
0111 <sup>(1)</sup>	D	D	D	A	A	A	A	A	A	A	A
1000	D	D	D	D	A	A	A	A	A	A	A
1001	D	D	D	D	D	A	A	A	A	A	A
1010	D	D	D	D	D	D	A	A	A	A	A
1011	D	D	D	D	D	D	D	A	A	A	A
1100	D	D	D	D	D	D	D	D	A	A	A
1101	D	D	D	D	D	D	D	D	D	A	A
1110	D	D	D	D	D	D	D	D	D	D	A
1111	D	D	D	D	D	D	D	D	D	D	D

A = Analog input

D = Digital I/O

Fig. 4.1.4-3 Tabella settaggio bit 3-0 registro ADCON1

Si passa poi al settaggio del registro di controllo dell'oscillatore di sistema, che servirà ad imporre la frequenza di clock del dispositivo:

*OSCCON=0b01000110;*

#### REGISTER 3-2: OSCCON: OSCILLATOR CONTROL REGISTER

R/W-0	R/W-1	R/W-0	R/W-0	R <sup>(1)</sup>	R-0	R/W-0	R/W-0
IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0
bit 7							bit 0

*Fig. 4.1.4-4 Registro OSCCON (datasheet slide 36/490)*

Il registro è settato per imporre una frequenza di oscillazione di 1MHz (bit *IRCFn*) e seleziona come fonte primaria di clock per il dispositivo, l'oscillatore interno di cui è provvisto PIC del modello scelto (bit *SCSn*).

Viene poi disattivato il modulo CCP del PIC (perché non utilizzato):

*CCP1CON=0b00000000; //Usa solo gli ultimi 4 bit*

#### REGISTER 16-1: CCP1CON: CAPTURE/COMPARE/PWM CONTROL REGISTER

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
bit 7							bit 0

*Fig. 4.1.4-5 Registro CCP1CON (datasheet slide 167/490)*

Viene disattivata la priorità degli interrupt del dispositivo (in quanto non è risultato necessario introdurla) azzerando l'apposito bit del registro *RCON*.

*RCONbits.IPEN=0;*

Successivamente vengono configurati in sequenza tre moduli Timer interni al dispositivo, che verranno poi utilizzati per lo svolgimento di alcune funzioni del programma (che verranno trattate nel seguito):

*T0CON=0b10000111;*

*T1CON=0b10110000;*

*T3CON=0b10110001;*

I tre moduli, come si evince dal codice sono i Timer 0,1 e 3. Tutti e tre sono settati per funzionare a 16 bit, in questa modalità di funzionamento ogni timer fa uso di 2 registri di conteggio (di 8 bit l'uno) denominati *TMRnL* e *TMRnH* (n come sempre è l'indice alfanumerico del registro). È bene tener presente fin da ora che il reset dei valori di questi contatori deve avvenire secondo una certa sequenza (di cui si parlerà a breve, dopo la presentazione dei Timer impiegati).

Il Timer0 verrà utilizzato per imporre una ciclicità temporale a quella che definiremo successivamente come la routine di Alive CAN del sistema (routine che va a monitorare lo stato delle comunicazioni/funzionalità dei dispositivi connessi alla rete CAN del sistema), ed è settato per utilizzare come fonte di clock, il clock interno del dispositivo, con un prescaler assegnato di 1:256 (il conteggio avviene sui fronti di salita).

#### REGISTER 12-1: T0CON: TIMER0 CONTROL REGISTER

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

*Fig. 4.1.4-6 Registro T0CON (datasheet slide 151/490)*

La differenza fondamentale tra questo timer ed il successivo è che questo è impostato per attivarsi immediatamente dopo essere stato configurato (bit 7 = *TMROON* settato ad 1).

Il Timer1 verrà impiegato anch'esso all'interno della routine di Alive CAN ed è impostato similmente al precedente per utilizzare il clock interno, con un prescaler assegnato di 1:8 (non viene attivato subito dopo la configurazione).

#### REGISTER 13-1: T1CON: TIMER1 CONTROL REGISTER

R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

Fig. 4.1.4-7 Registro T1CON (datasheet slide 155/490)

Infine, il Timer3 troverà impiego nella routine di Alive relativa alla connessione RS-232 tra PIC Master e master di sistema (FPGA o Raspberry), ed è settato similmente ai precedenti per utilizzare il segnale di clock derivante dall'oscillatore interno, con un prescaler assegnato di 1:8, e come il Timer0 si attiva immediatamente dopo il settaggio.

#### REGISTER 15-1: T3CON: TIMER3 CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	T3ECCP1 <sup>(1)</sup>	T3CKPS1	T3CKPS0	T3CCP1 <sup>(1)</sup>	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON
bit 7							bit 0

Fig. 4.1.4-8 Registro T3CON (datasheet slide 163/490)

Riprendendo ora il discorso sulla sequenza di reset dei suddetti, essa si articola (ed è uguale per tutti i Timer) nei seguenti passaggi:

1. A Timer in funzione (bit  $\text{TMRnON}=1$ ) si deve azzerare l'high byte del contatore ( $\text{TMRnH}$ );
2. Si azzerà poi il low byte del contatore ( $\text{TMRnL}$ );
3. Successivamente il Timer può essere disattivato (bit  $\text{TMRnON}=0$ ) in modo da interrompere il conteggio;

Proseguendo, vengono settati i registri di configurazione del modulo EUSART che sarà utilizzato per la comunicazione tramite protocollo RS-232:

- $\text{RCSTA}=0b10010000; //Porte TX e RX abilitate, ricezione 8n1, ricezione dati abilitata$

#### REGISTER 19-2: RCSTA: RECEIVE STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

Fig. 4.1.4-9 Registro RCSTA (datasheet slide 233/490)

Con questi valori assegnati al registro  $\text{RCSTA}$  viene abilitata la ricezione di messaggi ad 8 bit sul modulo (vengono settate quindi come TX ed RX le porte  $\text{RC6}$  ed  $\text{RC7}$  rispettivamente);

- $\text{TXSTA}=0b00100100; //Modalità asincrona, baud rate high speed, trasmissione 8n1, trasmissione dati abilitata$

### REGISTER 19-1: TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN <sup>(1)</sup>	SYNC	SENDDB	BRGH	TRMT	TX9D
bit 7							bit 0

Fig. 4.1.4-10 Registro TXSTA (datasheet slide 232/490)

Con questi valori di TXSTA il modulo viene poi settato per operare in modalità asincrona. Viene inoltre abilitata la trasmissione di messaggi ad 8 bit con alti valori di Baud Rate (impostazione high speed);

- **BAUDCON**=0b00001000; //Baud rate definito dai due registri SPBRGH e SPBRG

### REGISTER 19-3: BAUDCON: BAUD RATE CONTROL REGISTER

R/W-0	R-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN
bit 7							bit 0

Fig. 4.1.4-11 Registro BAUDCON (datasheet slide 234/490)

L'unico valore rilevante del registro **BAUDCON** settato è quello del bit **BRG16**, che abilità l'impostazione del valore del Baud Rate tramite 16 bit (scrivendo sui registri **SPBRGH** e **SPBRG**)

- **SPBRGH**=0;
- **SPBRG**=25; //Definisco un baud rate pari a 9600 per Fosc=1MHz (vedi tabella pag. 238 del datasheet)

I valori che sono stati imposti per i registri ai punti **SPRGn** sono stati presi dalla seguente tabella ([datasheet slide 238/490](#)), al fine di ottenere il valore di Baud Rate desiderato (9600 bit/s).

BAUD RATE (K)	<b>SYNC = 0, BRGH = 1, BRG16 = 1 or SYNC = 1, BRG16 = 1</b>								
	FOSC = 4.000 MHz			FOSC = 2.000 MHz			Fosc = 1.000 MHz		
	Actual Rate (K)	% Error	SPBRG value (decimal)	Actual Rate (K)	% Error	SPBRG value (decimal)	Actual Rate (K)	% Error	SPBRG value (decimal)
0.3	0.300	0.01	3332	0.300	-0.04	1665	0.300	-0.04	832
1.2	1.200	0.04	832	1.201	-0.16	415	1.201	-0.16	207
2.4	2.404	0.16	415	2.403	-0.16	207	2.403	-0.16	103
9.6	9.615	0.16	103	9.615	-0.16	51	9.615	-0.16	25
19.2	19.231	0.16	51	19.230	-0.16	25	19.230	-0.16	12
57.6	58.824	2.12	16	55.555	3.55	8	—	—	—
115.2	111.111	-3.55	8	—	—	—	—	—	—

Fig. 4.1.4-12 Tabella valori di SPRG ed SPRGH utilizzata

Successivamente alla configurazione del modulo EUSART viene effettuato un preliminare pre-settaggio di alcuni registri relativi al modulo CAN (modulo, la cui configurazione verrà effettuata più avanti) (fare riferimento alla [sezione 4.1.5](#) per maggiori dettagli):

**RXB1CON**=0b00100000;

**RXBOCON**=0b00100000;

Queste due righe di codice hanno lo scopo di inizializzare/configurare i due registri di controllo dei buffer di ricezione messaggi CAN, in particolare:

- Il registro di controllo del buffer 0 **RXB0CON** è settato per la ricezione di soli messaggi CAN in formato standard;

**REGISTER 24-13: RXB0CON: RECEIVE BUFFER 0 CONTROL REGISTER**

Mode 0	R/C-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
	RXFUL <sup>(1)</sup>	RXM1	RXM0	—	RXRTRRO	RXB0DBEN	JTOFF <sup>(2)</sup>	FILHITO
Mode 1,2	R/C-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
	RXFUL <sup>(1)</sup>	RXM1	RTRRO	FILHIT4	FILHIT3	FILHIT2	FILHIT1	FILHITO

Fig. 4.1.4-13 Registro di controllo RXB0CON (datasheet slide 293/490)

- Il secondo registro (**RXB1CON**) è stato settato similmente al primo per la ricezione di soli messaggi standard CAN, tuttavia nel nostro sistema il suo uso non è stato contemplato (tutti i messaggi sono gestiti dal buffer 0).

**REGISTER 24-14: RXB1CON: RECEIVE BUFFER 1 CONTROL REGISTER**

Mode 0	R/C-0	R/W-0	R/W-0	U-0	R-0	R/W-0	R-0	R-0
	RXFUL <sup>(1)</sup>	RXM1	RXM0	—	RXRTRRO	FILHIT2	FILHIT1	FILHITO
Mode 1,2	R/C-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
	RXFUL <sup>(1)</sup>	RXM1	RTRRO	FILHIT4	FILHIT3	FILHIT2	FILHIT1	FILHITO

Fig. 4.1.4-14 Registro di controllo RXB1CON (datasheet slide 295/490)

Infine, l'ultima parte dei settaggi è dedicata all'abilitazione delle fonti di interrupt di interesse per il nostro sistema, tuttavia prima di procedere è bene tenere a mente quanto segue. L'abilitazione, il controllo e la gestione degli interrupt si basa sull'uso di un variegato numero di registri, di cui nel seguito è riportata una sommaria descrizione:

- I registri **INTCONn** contengono gli enabler di varie fonti di interrupt e i loro rispettivi flag, nonché alcuni bit di settaggio per impostazioni generali;
- I registri **PIEn** contengono gli enabler degli interrupt periferici;
- I registri **PIRn** contengono i flag relativi agli interrupt periferici (sono quindi il corrispettivo dei **PIEn**) e sono quelli che ne permettono la gestione;
- I registri **IPRn**, che permettono la gestione/assegnazione dei livelli di priorità delle varie fonti di interrupt (non gestiti nel nostro progetto, in quanto non necessari);

Passando ora ai settaggi veri e propri:

- INTCON=0b11100000;**

**REGISTER 10-1: INTCON: INTERRUPT CONTROL REGISTER**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMROIE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF <sup>(1)</sup>
bit 7							bit 0

Fig. 4.1.4-15 Registro di controllo interrupt globali INTCON (datasheet slide 121/490)

Prima di tutto viene settato il registro **INTCON**, nel quale i bit 7 (**GIE**) e 6 (**PEIE**) sono gli enabler generali rispettivamente degli interrupt globali e di quelli periferici (questo vale nel nostro caso perché è stata disattivata la possibilità di avere più livelli di priorità degli interrupt, altrimenti quanto appena detto andrebbe leggermente rivisto, per maggiori dettagli si rimanda al [datasheet del dispositivo slide 121/490](#)). Settare ad 1 **GIE** corrisponde quindi ad abilitare la possibilità di ricevere interrupt, mentre settare ad 1 **PEIE** corrisponde

ad abilitare la possibilità di ricevere interrupt periferici (Timer diversi dallo 0, modulo CAN, modulo EUSART etc...). Con questi valori assegnati ai bit del registro viene inoltre abilitato l'interrupt su Overflow del Timer0 (**TMROIE**). Successivamente si passa all'abilitazione delle altre fonti di interrupt attraverso il settaggio dei registri **PIEn**, registri adibiti all'abilitazione degli interrupt periferici specifici dei vari moduli interni al PIC.

- **PIE1**=0b000100001;

#### REGISTER 10-7: PIE1: PERIPHERAL INTERRUPT ENABLE REGISTER 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE <sup>(1)</sup>	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							
bit 0							

Fig. 4.1.4-16 Registro di abilitazione interrupt periferici **PIE1** (datasheet slide 127/490)

Con questi settaggi vengono abilitati gli interrupt sulla ricezione di un messaggio da parte del modulo EUSART (**RCIE**) e sull'Overflow del Timer1 (**TMR1IE**).

- **PIE2**=0b000000010;

#### REGISTER 10-8: PIE2: PERIPHERAL INTERRUPT ENABLE REGISTER 2

R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
OSCFIE	CMIE <sup>(1)</sup>	—	EEIE	BCLIE	HLVDIE	TMR3IE	ECCP1IE <sup>(2)</sup>
bit 7							
bit 0							

Fig. 4.1.4-17 Registro di abilitazione interrupt periferici **PIE2** (datasheet slide 128/490)

Viene abilitato l'interrupt su Overflow del Timer3 (**TMR3IE**).

- **PIE3**=0b10100001;

#### REGISTER 10-9: PIE3: PERIPHERAL INTERRUPT ENABLE REGISTER 3

<b>Mode 0</b>	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IRXIE	WAKIE	ERRIE	TXB2IE	TXB1IE <sup>(1)</sup>	TXB0IE <sup>(1)</sup>	RXB1IE	RXB0IE
<b>Mode 1,2</b>	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IRXIE	WAKIE	ERRIE	TXBnIE	TXB1IE <sup>(1)</sup>	TXB0IE <sup>(1)</sup>	RXBnIE	FIFOWMIE <sup>(1)</sup>

Fig. 4.1.4-18 Registro di abilitazione interrupt periferici **PIE3** (datasheet slide 129/490)

In questo registro infine vengono abilitati gli interrupt relativi al modulo CAN di nostro interesse, in particolare vengono abilitati l'interrupt sulla ricezione/trasmissione di messaggi non validi (**IRXIE**), quello relativo agli errori generici del modulo CAN (**ERRIE**) ed infine l'interrupt sulla ricezione di messaggi CAN sul buffer di ricezione 0 (**RXB0IE**, l'unico, come detto in precedenza, utilizzato nel nostro programma).

Si tenga presente comunque che tutti i bit aventi valore 0 riportati nei settaggi dei registri **PIEn** precedenti corrispondono a fonti di interrupt non monitorate/gestite e quindi disattivate.

#### 4.1.5 CAN Configuration Mode

Dopo aver configurato i registri “generali” del PIC, è ora necessario affrontare la configurazione del modulo CAN del dispositivo. Ancora una volta è importante tenere presente che questa procedura accomuna tutti i dispositivi PIC presenti nel sistema ed è svolta nello stesso modo, tuttavia per chiarezza nel prosieguo di questa sezione verrà messo in evidenza ciò che è necessario cambiare da un dispositivo all’altro in fase di configurazione.

Al fine di configurare il modulo CAN del dispositivo risulta necessario richiedere al modulo di entrare in modalità di configurazione (i registri di controllo/settaggio non sono accessibili per la scrittura nelle altre modalità di funzionamento, di cui si parlerà nel seguito, al fine di arginare problemi derivanti da riconfigurazioni non volute). Per fare ciò viene effettuata nel main una chiamata alla funzione “CAN\_Config\_Mode” contenuta nell’apposito file (che come si vedrà è stato adibito alle sole funzioni di riconfigurazione CAN per motivi di ordine/chiarezza) **CAN\_Modes.c**. Si riporta nel seguito il codice della funzione, che verrà poi commentato.

```
//Configuro il modulo CAN  
CAN_Config_Mode();
```

Codice MPLAB:

```
void CAN_Config_Mode(void)  
{  
    //Inizializzazione modulo CAN  
  
    //Registro di controllo CAN (R/W)  
    //Richiedo la modalità di configurazione del modulo CAN  
    CANCONbits.REQOP2=1;  
  
    //Aspetto che il modulo CAN sia in Config. Mode  
    while(CANSTATbits.OPMODE2 != 1 || CANSTATbits.OPMODE1 != 0 || CANSTATbits.OPMODE0 != 0);  
  
    //Lettura Registro Status CAN (R only)  
    //CANSTAT=0b00000000; //(vedi datasheet slide 283/490)  
  
    //Imposto la modalità MODE 0  
    //Registro di controllo ECAN (R/W) //(vedi datasheet slide 286/490)  
    ECANCONbits.MDSEL1=0;  
    ECANCONbits.MDSEL0=0;  
  
    //Settaggio Baud Rate  
    //Primo Registro di settaggio del baud rate  
    BRGCON1=0b00000000; //(datasheet slide 317/490)  
    //Secondo Registro di settaggio del baud rate  
    BRGCON2=0b11000010; //(datasheet slide 318/490)  
    //Terzo Registro di settaggio del baud rate  
    BRGCON3=0b11000010; //(datasheet slide 319/490)  
    //Velocità di trasmissione nominale = 312500 bps  
    //SJW=1 ; BRP=1 ; SEG1PH=1 ; PRSEG=3 ; SEG2PH=3  
  
    //Registro di Stato Comunicazione (R/C)  
    //COMSTAT=0b00000000; //(vedi datasheet slide 287/490)
```

```

//Registro buffer per trasmissione CAN (TXB1CON E TXB2CON SONO CONFIGURATI SIMILMENTE)
(R/partially W)
//TXB0CON=0b00000011; //(vedi datasheet slide 288/490)
//TXB1CON=0b00000010;
//TXB2CON=0b00000001;

//Registro identificativi di trasmissione messaggi CAN, High Byte (TXB1SIDH E TXB2SIDH SONO
CONFIGURATI SIMILMENTE) (R/W)
//Abbiamo 7 casi differenti, quindi dovrebbero bastare i 3 bit del Low Byte
//TXB0SIDH=0b00000000; //(datasheet slide 289/490)
//TXB1SIDH=0b00000000;
//TXB2SIDH=0b00000000;

//Registro identificativi di trasmissione messaggi CAN, Low Byte (TXB1SIDL E TXB2SIDL SONO
CONFIGURATI SIMILMENTE) (R/W)
//TXB0SIDL=0b00000000; (datasheet slide 289/490)
//TXB1SIDL=0b00000000;
//TXB2SIDL=0b00000000;

//Registro di impostazione Data Field in buffer selezionato;
//TXBnDm=0b00000000; (datasheet slide 290/490)

//Registro di impostazione Data Length (TXB1DLC E TXB2DLC SONO CONFIGURATI SIMILMENTE)
(R/W)
//TXB0DLC=0b00000101; //(datasheet slide 291/490 esempi nelle successive pagine)
//TXB1DLC=0b00000101;
//TXB2DLC=0b00000101;

//Registro di controllo buffer di ricezione 0 (Puoi settare solo alcuni bit)
//RXB0CON=0b00100100; //(datasheet slide 293/490)
//Registro di controllo buffer di ricezione 1 (Puoi settare solo alcuni bit)
//RXB1CON=0b00100100; //(datasheet slide 295/490)

//Registro identificativi di ricezione messaggi CAN, High Byte (R-only)
//RXBxSIDH=0b00000000; //(datasheet slide 296/490)

//Registro identificativi di ricezione messaggi CAN, Low Byte (R-only)
//RXBxSIDL=0b00000000; (datasheet slide 297/490)

//Registro filtri di ricezione CAN (R-only)
//RXBxDLC=0b00000000; (datasheet slide 298/490)

//Registro di impostazione Data Field in buffer selezionato; (R-only)
//RXBnDm=0b00000000; (datasheet slide 298/490)

//Registro di Controllo errori CAN; (R-only)
//RXERRCNT=0b00000000; (datasheet slide 299/490)

```

```

//Registri di settaggi filtri di accettazione messaggi CAN, High Byte
//Setto a 0 i registri dei filtri High Byte perchè mi bastano i 3 bit
//del Low Byte per identificare 5 porte + master + Initial_Test
//In MODE 0 sono disponibili solo i registri da 0 a 5
RXF0SIDH=0b00000000; //(datasheet slide 308/490)
RXF1SIDH=0b00000000;
RXF2SIDH=0b00000000;
RXF3SIDH=0b00000000;
RXF4SIDH=0b00000000;
RXF5SIDH=0b00000000;

//Registri di settaggi filtri di accettazione messaggi CAN, Low Byte
//In MODE 0 sono disponibili solo i registri da 0 a 5
//Ogni dispositivo avrà solo 2 filtri, quello per il messaggio di test iniziale e quello
//che identifica i messaggi ad esso indirizzati

RXF0SIDL=0b11000000; // Filtro di identificazione dispositivo (Master=6)
RXF1SIDL=0b00000000; // Test Iniziale
RXF2SIDL=0b00000000; // (datasheet slide 308/490)
RXF3SIDL=0b00000000; //
RXF4SIDL=0b00000000; //
RXF5SIDL=0b00000000; //

//RXF1SIDL=0b11100000; Initial_Test
//RXF1SIDL=0b00100000; Slave 1
//RXF1SIDL=0b01000000; Slave 2
//RXF1SIDL=0b01100000; Slave 3
//RXF1SIDL=0b10000000; Slave 4
//RXF1SIDL=0b10100000; Slave 5
//RXF1SIDL=0b11000000; Master

//Registro di settaggio maschera di accettazione identificativi CAN, High Byte
//Setto a 0 i registri dei filtri High Byte perchè mi bastano i 3 bit
//del Low Byte per identificare 5 porte + master + Initial_Test
RXMOSIDH=0b00000000; //(datasheet slide 309/490)
RXM1SIDH=0b00000000;

//Registro di settaggio maschera di accettazione identificativi CAN, Low Byte
RXMOSIDL=0b11100000; //(datasheet slide 310/490)
RXM1SIDL=0b11100000;

//Registro di controllo I/O CAN
CIOCON=0b00100000; //(datasheet slide 320/490)
}

```

Come appare evidente osservando il codice, sono presenti moltissime righe commentate, che sono state inserite in fase di progetto come “appunto” /riferimento per il programmatore. Si tenga inoltre presente che tra i file di progetto sarà presente un file denominato **CAN\_Module\_Notes** che non verrà descritto in questo

documento perché non effettivamente funzionale al progetto. Il file sopracitato è stato pensato in fase di progetto come un macro-archivio di riferimenti e note di programmazione, in quanto contiene solo righe di codice commentate.

Cominciando ora l'analisi del codice, si può notare come le prime istruzioni non siano altro che la richiesta fatta al modulo CAN di entrare in modalità di configurazione:

`CANCONbits.REQOP2=1;`

`while(CANSTATbits.OPMODE2 != 1 || CANSTATbits.OPMODE1 != 0 || CANSTATbits.OPMODE0 != 0);`

In realtà è bene tenere a mente che per impostare la richiesta di passaggio ad una certa modalità di funzionamento dovrebbero essere settati 3 bit del registro di controllo CAN (`REQOP2`, `REQOP1`, `REQOP0`), che sono appunto i bit adibiti alla funzione di cambio di modalità del modulo. Nel nostro caso è stato settato solo il primo perché (come riportato sul [datasheet sezione 24.2 pagina 282/490](#)) il modulo passa alla modalità di configurazione quando il bit `REQOP2` assume valore 1, indipendentemente dal valore degli altri due.

La seconda riga di istruzione ha lo scopo di “trattenere” il programma in questo punto del codice fino a quando il modulo CAN non sarà effettivamente entrato nella modalità voluta. Questo risulta necessario in quanto come descritto nel [datasheet](#) del dispositivo nella [sezione 24](#), per riconfigurare il modulo CAN è prima necessario settare i bit `REQn` contenuti nel registro di controllo `CANCON`, che hanno la funzione di RICHIEDERE la configurazione.

#### REGISTER 24-1: CANCON: CAN CONTROL REGISTER

Mode 0	R/W-1	R/W-0	R/W-0	R/S-0	R/W-0	R/W-0	R/W-0	U-0
	REQOP2	REQOP1	REQOP0	ABAT	WIN2	WIN1	WIN0	—
Mode 1	R/W-1	R/W-0	R/W-0	R/S-0	U0	U-0	U-0	U-0
	REQOP2	REQOP1	REQOP0	ABAT	—	—	—	—
Mode 2	R/W-1	R/W-0	R/W-0	R/S-0	R-0	R-0	R-0	R-0
	REQOP2	REQOP1	REQOP0	ABAT	FP3	FP2	FP1	FP0
bit 7								
bit 0								

Fig. 4.1.5-1 Registro CANCON

#### REGISTER 24-2: CANSTAT: CAN STATUS REGISTER

Mode 0	R-1	R-0	R-0	R-0	R-0	R-0	R-0	U-0
	OPMODE2 <sup>(1)</sup>	OPMODE1 <sup>(1)</sup>	OPMODE0 <sup>(1)</sup>	—	ICODE3	ICODE2	ICODE1	—
Mode 1,2	R-1	R-0	R-0	R-0	R-0	R-0	R-0	R-0
	OPMODE2 <sup>(1)</sup>	OPMODE1 <sup>(1)</sup>	OPMODE0 <sup>(1)</sup>	EICODE4	EICODE3	EICODE2	EICODE1	EICODE0
bit 7								
bit 0								

Fig. 4.1.5-2 Registro CANSTAT

Tale richiesta può richiedere tempo per essere onorata, ed è quindi necessario aspettare prima di effettuare i successivi settaggi, monitorando i bit `OPMODE2`, `OPMODE1`, `OPMODE0` del registro `CANSTAT`, i quali descrivono l'effettivo stato del modulo.

Come si può notare nelle precedenti figure è evidenziata la MODE 0, che in realtà identifica la modalità Legacy del dispositivo, che coincide in sintesi con la modalità di funzionamento base (e default) del modulo CAN. La ragione per cui abbiamo scelto di operare con questa modalità è che, ai fini del nostro progetto, non abbiamo ritenuto necessario usufruire delle features aggiuntive offerte dalle altre modalità di funzionamento. Per impostare la Mode 0 del modulo CAN è necessario settare alcuni bit del registro di controllo `ECANCON` ([datasheet sezione 24.2 pagina 286/490](#)).

`ECANCONbits.MDSEL1=0;`

`ECANCONbits.MDSEL0=0;`

### REGISTER 24-3: ECANCON: ENHANCED CAN CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-1	R/W-0	R/W-0	R/W-0	R/W-0
MDSEL1 <sup>(1)</sup>	MDSEL0 <sup>(1)</sup>	FIFOWM <sup>(2)</sup>	EWIN4	EWIN3	EWIN2	EWIN1	EWIN0
bit 7							bit 0

Fig. 4.1.5-3 Registro ECANCON

Seguiti questi passaggi preliminari è ora possibile cominciare a svolgere i settaggi desiderati. Come prima cosa è necessario impostare il Baud Rate del modulo (tutti i nodi di una rete per CAN per poter comunicare correttamente devono utilizzare lo stesso Baud Rate). Questo viene fatto settando i tre registri *BRGCONn* del modulo:

*BRGCON1*=0b00000000;

*BRGCON2*=0b11000010;

*BRGCON3*=0b11000010;

### REGISTER 24-52: BRGCON1: BAUD RATE CONTROL REGISTER 1

| R/W-0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| SJW1  | SJW0  | BRP5  | BRP4  | BRP3  | BRP2  | BRP1  | BRP0  |
| bit 7 |       |       |       |       |       |       | bit 0 |

Fig. 4.1.5-4 Registro BRGCON1

### REGISTER 24-53: BRGCON2: BAUD RATE CONTROL REGISTER 2

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
SEG2PHTS	SAM	SEG1PH2	SEG1PH1	SEG1PH0	PRSEG2	PRSEG1	PRSEG0
bit 7							bit 0

Fig. 4.1.5-5 Registro BRGCON2

### REGISTER 24-54: BRGCON3: BAUD RATE CONTROL REGISTER 3

R/W-0	R/W-0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0
WAKDIS	WAKFIL	—	—	—	SEG2PH2 <sup>(1)</sup>	SEG2PH1 <sup>(1)</sup>	SEG2PH0 <sup>(1)</sup>
bit 7							bit 0

Fig. 4.1.5-6 Registro BRGCON3

La scelta dei valori dei segmenti temporali ha seguito i passaggi/vincoli indicati sul [datasheet sezione 24.9](#), e per chiarezza/ordine sono stati svolti utilizzando un documento Live Script di Matlab, riportato e discusso nel seguito (file **Baud\_Rate** in allegato al materiale di progetto):

```

Live Editor - C:\Users\Mattia\Desktop\Calcolo Baud Rate CAN\Baud_Rate mlx
Baud_Rate.mlx + 1

Vincoli:
PR_SEG + SEG_1_PH >= SEG_2_PH;
SEG_2_PH >= SJW

Procedura: Impongo a priori la velocità di trasmissione e ricavo di conseguenza il tempo di bit:

1 %Parametri noti/imposti a priori%
2 FOSC=1; %Frequenza oscillatore (in Mhz)
3 %Scelgo gli altri parametri%
4 SJW=1; %Tempo di jump per sincro (multiplo di Tq)
5 SEG_1_PH=1; %Tempo phase 1 (multiplo di Tq)
6 PR_SEG=3; %Tempo propagation phase (multiplo di Tq)
7 SEG_2_PH=3; %Tempo phase 2 (multiplo di Tq)

8 %Impongo il Nominal bit rate (=1/Tbit)*%
9 N_B_R=31250; %bit/s%
10 disp(N_B_R) 31250

11 %Ricavo il nominal bit time Tbit, come inverso del bit rate voluto%
12 Tbit=1/N_B_R;
13 disp(Tbit) 3.2000e-05

14 %Calcolo il numero di Time quanta per bit in base alla scelta dei valori precedenti%
15 n_Tq=(SJW+SEG_1_PH+PR_SEG+SEG_2_PH); %Numero di Time quanta
16 disp(n_Tq) 8

17 %Ricavo il tempo di bit Tq (Tbit=Tq x n_Tq)%
18 Tq=(Tbit/n_Tq);
19 disp(Tq) 4.0000e-06

20 %Calcolo il prescaler necessario, ossia BRP%
21 %Tq_p2 va in microsecondi!!!!%
22 BRP=((FOSC*(Tq*10^6))/2)-1
23
24
25
26
27

```

Fig. 4.1.5-7 Calcolo parametri Baud Rate CAN

È essenziale notare che la procedura riportata svolge a ritroso i passaggi presenti sul datasheet, ma ne usa i medesimi vincoli e considerazioni, ossia la scelta “a priori” dei multipli di Tq che compongono il tempo di bit, non è stata svolta casualmente, ma secondo quanto riportato sul [datasheet sezione 24.9](#). Gli unici due vincoli che non sono stati riportati nel documento live script (ma presenti sul datasheet e comunque soddisfatti) riguardano:

- Il valore minimo assumibile da Tq (tempo di bit), che non può essere, per definizione inferiore ad 1 microsecondo, valore che corrisponderebbe ad un Baud Rate di 1 Mb/s;
- Il secondo vincolo deriva da quello precedente, ossia per definizione l'estensione in Time Quanta del tempo di bit (numero di Time Quanta per bit) deve essere un numero intero non maggiore di 25 e non minore di 8, nel nostro caso come viene mostrato nella Fig. soprastante, esso è 8;

Come si può notare i calcoli (e i conseguenti settaggi) portano ad ottenere un valore perfettamente intero di BRP (valore del prescaler BRG).

Il passo successivo all'impostazione del Baud Rate riguarda la configurazione di filtri e maschere CAN. In Mode 0 il modulo vanta la possibilità di abilitare fino a 6 possibili diversi filtri. Ogni filtro si compone (considerando solo identificativi standard) fondamentalmente di due parti (High Byte = [RXFnSIDH](#) e Low Byte = [RXFnSIDL](#)).

#### **REGISTER 24-15: RXBnSIDH: RECEIVE BUFFER n STANDARD IDENTIFIER REGISTERS, HIGH BYTE [0 ≤ n ≤ 1]**

R-x	R-x	R-x	R-x	R-x	R-x	R-x	R-x
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

Fig. 4.1.5-8 Registro RXBnSIDH

**REGISTER 24-16: RXBnSIDL: RECEIVE BUFFER n STANDARD IDENTIFIER REGISTERS,  
LOW BYTE [0 ≤ n ≤ 1]**

R-x	R-x	R-x	R-x	R-x	U-0	R-x	R-x
SID2	SID1	SID0	SRR	EXID	—	EID17	EID16
bit 7							bit 0

Fig. 4.1.5-9 Registro RXBnSIDL

Per gli scopi del nostro progetto (come si evince anche dal codice sopra riportato) è stato sufficiente usare la sola parte Low Byte di ogni filtro, in quanto il numero di messaggi da gestire è risultato contenuto, o quantomeno il tipo di gestione è che stato implementato ha permesso di ridurre il numero di identificativi necessari. Questo metodo consiste fondamentalmente nell'associare ogni dispositivo PIC (nodo del network CAN) ad un identificativo (consapevoli del fatto che l'identificativo sia in realtà associato al messaggio e non al nodo) e inserire tra i byte di dati del messaggio anche cifre particolari che dovranno essere elaborate dal nodo interessato per estrarne il comando/azione richiesto/corrispondente. In quest'ottica gli identificativi individuati sono i seguenti (riportati anche nel codice sopra):

- //RXF1SIDL=0b11100000; Initial\_Test
- //RXF1SIDL=0b00100000; Slave 1
- //RXF1SIDL=0b01000000; Slave 2
- //RXF1SIDL=0b01100000; Slave 3
- //RXF1SIDL=0b10000000; Slave 4
- //RXF1SIDL=0b10100000; Slave 5
- //RXF1SIDL=0b11000000; Master

Come si nota dalla disposizione degli identificativi sopra riportati, le scelte fatte pongono il Master nella condizione di essere sempre il nodo a minore priorità in fase di trasmissione, questo in realtà non è propriamente un problema in quanto (come sarà discusso più diffusamente nel prosieguo del documento) il nostro sistema prevede un gran numero di messaggi inviati dal Master agli Slave e solo un ristretto numero di messaggi aventi "direzione" opposta, i quali peraltro come sarà chiaro successivamente, identificano situazioni particolari, la cui gestione deve essere prioritaria per il corretto funzionamento della rete. I registri *RXFnSIDL* o meglio i loro valori sono fondamentalmente l'unico parametro che varia in tutta la procedura di configurazione CAN da un modulo all'altro.

Osservando l'elenco si nota anche che in realtà il primo identificativo non è relativo ad un PIC, questo perché esso è dedicato ad una particolare funzione di test/monitoraggio, per la cui descrizione si rimanda alla sezione 4.1.7 del documento.

L'ultima cosa da tenere presente circa i filtri CAN, riguarda la loro gestione. Il modulo CAN dispone in Mode 0 di 2 buffer di ricezione *RXB1* ed *RXB0*, identici (per maggiori dettagli sul loro funzionamento si rimanda alla sezione 4.1.11 del documento). In Mode 0 al buffer *RXB0* sono associati i primi 2 registri di filtraggio (i restanti sono invece riferiti al secondo buffer). Questo, ancora una volta non è un problema in quanto ognuno dei PIC del nostro sistema necessita di 2 soli filtri, uno per la funzione *Initial\_Test()* ed uno per i messaggi aventi l'identificativo corrispondente (infatti, nel codice riportato in precedenza, tutti i registri di filtraggio aventi indice maggiore di 1 sono semplicemente azzerati, inoltre, dato il quantitativo di messaggi CAN scambiati, abbiamo ritenuto che l'uso del solo *RXB0* fosse sufficiente).

Settati i filtri è necessario impostare le maschere. Sono disponibili un totale di 2 macchere, configurabili attraverso i corrispondenti 4 registri (*RXMnSIDH*, *RXMnSIDL*), 2 per maschera.

**REGISTER 24-41: RXMnSIDH: RECEIVE ACCEPTANCE MASK n STANDARD IDENTIFIER MASK REGISTERS, HIGH BYTE [0 ≤ n ≤ 1]**

| R/W-x |
|-------|-------|-------|-------|-------|-------|-------|-------|
| SID10 | SID9  | SID8  | SID7  | SID6  | SID5  | SID4  | SID3  |
| bit 7 |       |       |       |       |       |       | bit 0 |

Fig. 4.1.5-10 Registro RXMnSIDH

**REGISTER 24-42: RXMnSIDL: RECEIVE ACCEPTANCE MASK n STANDARD IDENTIFIER MASK REGISTERS, LOW BYTE [0 ≤ n ≤ 1]**

R/W-x	R/W-x	R/W-x	U-0	R/W-0	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDEN <sup>(1)</sup>	—	EID17	EID16
bit 7							bit 0

Fig. 4.1.5-11 Registro RXMnSIDL

Come riportato nel codice presente all'inizio di questa sezione, le due maschere sono settate in modo tale da permettere ai nodi l'elaborazione di soli messaggi aventi identificativi compresi tra 0 e 7 (in decimale) ed in formato standard:

*RXMOSIDH=0b00000000; // (datasheet slide 309/490)*

*RXM1SIDH=0b00000000;*

*RXMOSIDL=0b11100000; // (datasheet slide 310/490)*

*RXM1SIDL=0b11100000;*

In merito proprio al formato dei messaggi CAN è bene precisare che nel presente sistema sono stati utilizzati SOLO messaggi CAN standard, filtri e maschere prevedono (come accennato in precedenza) registri aggiuntivi che permettono la gestione dei messaggi CAN estesi, che non sono stati però utilizzati (si rimanda al [datasheet sezione 24.2.3](#) per informazioni aggiuntive su di essi).

Come ultimo passo per la configurazione del modulo, si passa ora al settaggio del Registro [CIOCON](#) ossia:

*CIOCON=0b00100000; // (datasheet slide 320/490)*

**REGISTER 24-55: CIOCON: CAN I/O CONTROL REGISTER**

U-0	U-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0
—	—	ENDRHI <sup>(1)</sup>	CANCAP	—	—	—	—
bit 7							bit 0

Fig. 4.1.5-12 Registro CIOCON

Nel nostro caso, il settaggio di questo registro consiste nella necessità di settare i bit [CANCAP](#) (bit 4) ed [ENDRHI](#) (bit 5), il primo abilita la "cattura" del messaggio CAN (opzione non utilizzata, quindi disattivata), il secondo invece è stato settato ad 1, come suggerito nelle note del registro contenuto nel [datasheet pagina 320/490 sezione 24.2.5](#).

Questo conclude la configurazione del modulo CAN. Per maggiori informazioni ed un elenco completo di tutti i registri utilizzabili da questo modulo, si rimanda ancora una volta alla [sezione 24.2 del datasheet del dispositivo](#).

#### 4.1.6 CAN Normal Mode + altre configurazioni

Completata la configurazione del modulo CAN in Configuration Mode è ora possibile utilizzarlo, modificandone lo stato, in particolare questo viene fatto nel file main di progetto, chiamando una serie di funzioni adibite proprio alla richiesta di diverse modalità di funzionamento e contenute nel file **CAN\_Modes.c**.

Codice MPLAB:

```
void CAN_Normal_Mode(void)
{
    //Richiedo al CAN la modalità di normale funzionamento
    CANCONbits.REQOP2=0;
    CANCONbits.REQOP1=0;
    CANCONbits.REQOP0=0;

    //Aspetto che il modulo CAN sia in modalità Normale
    while(CANSTATbits.OPMODE2 != 0 || CANSTATbits.OPMODE1 != 0 || CANSTATbits.OPMODE0 != 0);
}

/*
void CAN_Loopback_Mode(void)
{
    //Richiedo al CAN la modalità Loopback
    CANCONbits.REQOP2=0;
    CANCONbits.REQOP1=1;
    CANCONbits.REQOP0=0;

    //Aspetto che il CAN entri in modalità Loopback
    while(CANSTATbits.OPMODE2 != 0 || CANSTATbits.OPMODE1 != 1 || CANSTATbits.OPMODE0 != 0);
}

void CAN_Sleep_Mode(void)
{
    //Richiedo al CAN la modalità Sleep
    CANCONbits.REQOP2=0;
    CANCONbits.REQOP1=0;
    CANCONbits.REQOP0=1;

    //Aspetto che il CAN entri in modalità Loopback
    while(CANSTATbits.OPMODE2 != 0 || CANSTATbits.OPMODE1 != 0 || CANSTATbits.OPMODE0 != 1);
}
*/
```

Il codice sopra riportato mostra le 3 funzioni usate per richiedere di volta in volta 3 modalità di funzionamento diverse:

- Normal Mode: Modalità di funzionamento standard del modulo CAN;
- Loopback Mode: Modalità usata soprattutto in fase di test in cui il modulo “simula” la presenza di una rete CAN inviando il contenuto del proprio registro di trasmissione al proprio registro di ricezione (test del modulo);

- Sleep Mode: Modalità a consumo ridotto, ossia se il modulo, per qualunque ragione, dovesse rimanere in stato di Idle per diverso tempo, sarebbe possibile prevedere che esso entri in modalità sospesa, abbassandone i consumi (per poi uscirne quando richiesto sulla base di opportune strategie software);

Si può osservare tuttavia che l'unico blocco non commentato risulta quello relativo alla modalità normale, questo perché in realtà la modalità Loopback è stata usata solo preliminary per i test dei moduli, mentre la modalità Sleep non è in realtà stata usata in quanto come precedentemente citato, il nostro sistema non considera (al momento) l'aspetto dell'ottimizzazione dei consumi.

Soffermandosi ora sul codice associato alla richiesta della modalità normale, chiamata dal main tramite prototipo contenuto nell'apposito file header **Function\_Prootypes.h**:

```
//Configuro il CAN per la modalità normale
```

```
CAN_Normal_Mode();
```

Ci si rende subito conto che la funzione è di per sé molto semplice, ossia come visto in precedenza per la modalità di configurazione, prima viene richiesto il cambio di stato tramite il cambio dei valori dei bit **REQn** del registro **CANCON**, dopodiché si attende (tramite il ciclo **while**) che il cambiamento diventi effettivo, monitorando il bit **OPMODEn** del registro **CANSTAT**.

Quando il programma esce da questa funzione, il modulo CAN è pronto per essere utilizzato.

#### 4.1.7 Funzione Initial\_Test() (Lato Master)

Subito dopo i settaggi e le configurazioni generali, è stata predisposta nel file main del PIC Master (**Main\_Master.c**) una chiamata ad una funzione denominata “Initial\_Test”.

//Funzione di test iniziale

*Initial\_Test();*

Il cui prototipo è contenuto nel nell'apposito file header dei prototipi (**Function\_Prootypes.h**) e il/la cui corpo/definizione è invece contenuto/a nel file adibito alle funzioni CAN (**CAN\_Functions.c**).

Codice MPLAB:

*void Initial\_Test(void)*

{

//Imposto la priorità dei messaggi

*TXB0CONbits.TXPRIO=1;*

*TXB0CONbits.TXPRI1=1;*

//Imposto l'identificativo messaggio

*TXBOSIDH=0b00000000;*

*TXBOSIDL=0b11100000; //Messaggio di test iniziale (a tutti)*

//Setto la lunghezza del messaggio

*TXB0DLC=0b00000101;*

//Imposto il contenuto dei registri da inviare

//TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)

*TXB0D0=10; //Scrivo nel registro 0 del buffer 0*

*TXB0D1=20; //Scrivo nel registro 1 del buffer 0*

*TXB0D2=30; //Scrivo nel registro 2 del buffer 0*

*TXB0D3=40; //Scrivo nel registro 3 del buffer 0*

*TXB0D4=50; //Scrivo nel registro 4 del buffer 0*

//Setto il bit di trasmissione

*TXREQ0=1;*

}

Prima di procedere con l'analisi del codice, è opportuno chiarire quale sia il procedimento per effettuare una trasmissione CAN e quali registri siano da esso interessati:

1. Il presupposto per poter trasferire un messaggio CAN è che il modulo CAN del PIC si trovi in modalità di funzionamento normale (o loopback se il nodo ricevente è lo stesso che trasmette);
2. Una volta certi che il modulo CAN sia nello stato corretto è necessario settare i registri per la trasmissione con le necessarie informazioni (dei buffer che si desidera trasmettere);
  - 2.1. Si tenga a mente che in Mode 0 il modulo CAN vanta la presenza di 3 buffer di trasmissione CAN uguali e utilizzabili separatamente, la procedura per la trasmissione è identica per tutti e 3 i buffer (in questa sede essa verrà riportata in relazione al buffer 0);
    - 2.1.1.Ogni buffer è composto da:

2.1.1.1. 1 registro di controllo;

**REGISTER 24-5: TXBnCON: TRANSMIT BUFFER n CONTROL REGISTERS [0 ≤ n ≤ 2]**

Mode 0	U-0	R-0	R-0	R-0	R/W-0	U-0	R/W-0	R/W-0
	—	TXABT <sup>(1)</sup>	TXLARB <sup>(1)</sup>	TXERR <sup>(1)</sup>	TXREQ <sup>(2)</sup>	—	TXPRI1 <sup>(3)</sup>	TXPRI0 <sup>(3)</sup>
Mode 1,2	R/C-0	R-0	R-0	R-0	R/W-0	U-0	R/W-0	R/W-0
	TXBIF	TXABT <sup>(1)</sup>	TXLARB <sup>(1)</sup>	TXERR <sup>(1)</sup>	TXREQ <sup>(2)</sup>	—	TXPRI1 <sup>(3)</sup>	TXPRI0 <sup>(3)</sup>
bit 7		bit 0						

Fig. 4.1.7-1 Registro TXBnCON

2.1.1.2. 4 registri per l'impostazione degli identificativi messaggi CAN (in realtà noi ne useremo solo 2, *TXBnSIDH* e *TXBnSIDL*, in quanto useremo solo identificativi standard e quindi non estesi, che comporterebbero l'uso dei registri *TXBnEIDH* ed *TXBnEIDL*);

**REGISTER 24-6: TXBnSIDH: TRANSMIT BUFFER n STANDARD IDENTIFIER REGISTERS, HIGH BYTE [0 ≤ n ≤ 2]**

R/W-x							
SID10	SID9	SID8	SID7	SID6	SID5	SID4	SID3
bit 7							bit 0

Fig. 4.1.7-2 Registro TXBnSIDH

**REGISTER 24-7: TXBnSIDL: TRANSMIT BUFFER n STANDARD IDENTIFIER REGISTERS, LOW BYTE [0 ≤ n ≤ 2]**

R/W-x	R/W-x	R/W-x	U-0	R/W-x	U-0	R/W-x	R/W-x
SID2	SID1	SID0	—	EXIDE	—	EID17	EID16
bit 7							bit 0

Fig. 4.1.7-3 Registro TXBnSIDL

2.1.1.3. 1 registro per l'impostazione della lunghezza del messaggio CAN;

**REGISTER 24-11: TXBnDLC: TRANSMIT BUFFER n DATA LENGTH CODE REGISTERS [0 ≤ n ≤ 2]**

U-0	R/W-x	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x
—	TXRTR	—	—	DLC3	DLC2	DLC1	DLC0
bit 7							bit 0

Fig. 4.1.7-4 Registro TXBnDLC

2.1.1.4. 8 registri di dati trasferibili (quindi al massimo 8 byte di dati);

**REGISTER 24-10: TXBnDm: TRANSMIT BUFFER n DATA FIELD BYTE m REGISTERS [0 ≤ n ≤ 2, 0 ≤ m ≤ 7]**

R/W-x							
TXBnDm7	TXBnDm6	TXBnDm5	TXBnDm4	TXBnDm3	TXBnDm2	TXBnDm1	TXBnDm0
bit 7							bit 0

Fig. 4.1.7-5 Registro TXBnDm

2.2. I bit *TXPRI0* e *TXPRI1* del registro *TXBnCON* servono ad impostare la priorità del messaggio e sono quindi la prima cosa da impostare;

2.2.1. Per essere precisi il settaggio di questi due bit serve a settare la priorità di trasmissione DEL BUFFER rispetto agli altri, pertanto se più buffer dovessero contemporaneamente richiedere l'invio del proprio contenuto, il modulo invierebbe i messaggi in ordine di priorità decrescente,

proprio sulla base dei valori di settaggio di questi bit (11 = priorità più alta possibile; 00=priorità più bassa possibile);

2.2.1.1. Nel caso più buffer, aventi la stessa priorità, dovessero richiedere l'invio messaggio nello stesso momento, verrebbe inviato prima il messaggio del buffer avente indice maggiore, ossia a parità di priorità impostata il contenuto dei buffer sarebbe spedito nell'ordine: TXB2 – TXB1- TXB0;

2.2.2. Il settaggio di questi due bit non alterna in alcun modo l'identificativo del messaggio CAN, a cui sono dedicati gli appositi registri citati in precedenza;

2.2.3. Nel nostro sistema la maggioranza dei messaggi CAN viene trasmessa attraverso il buffer 0, tranne che in un caso (come si vedrà), ossia nel trasferimento della password utente ad uno Slave;

2.3. Successivamente deve essere impostato l'identificativo messaggio, che deve essere contenuto nei registri *TXBnSIDH* o *TXBnSIDL* (si hanno a disposizione i 7 bit del SIDH e i bit 7-6-5 del SIDL);

2.3.1. Nel nostro sistema sono stati impiegati solo gli ultimi 3 del SIDL in quanto è stato deciso di indicare con ogni identificativo una porta (Master e Slave) + un identificativo che è stato riservato proprio al messaggio di test iniziale, per un totale di 7 identificativi utilizzati (come precedentemente discusso nella *sezione 4.1.5*);

2.4. Dopo l'identificativo è necessario impostare la lunghezza del messaggio che si vuole trasmettere, settando il registro *TXBnDLC*;

2.5. Ovviamente si devono poi settare i registri di dati *TXBnDm* con i valori del messaggio desiderato;

2.6. Infine, per richiedere l'invio del messaggio è necessario settare ad 1 il bit *TXREQ* contenuto nel registro di controllo del buffer utilizzato;

2.6.1. Settare ad 1 il bit di richiesta non implica la trasmissione immediata del messaggio;

2.6.2. È inoltre importante tenere a mente che questo bit sarà automaticamente azzerato nel momento in cui si avrà la conferma di corretto invio (bit di AKN settato ad uno da uno dei nodi);

Descritta la procedura generale di invio, si passa ora a commentare il codice sopra riportato.

Lo scopo di questa funzione non è altri che effettuare una preliminare verifica delle connessioni elettriche e del funzionamento minimo del sistema. Per fare ciò, quello che viene fatto è inizializzare un messaggio CAN di 5 byte, indirizzato a tutti i nodi CAN in quanto i primi 3 bit del registro di identificativo *TXBOSIDL* sono 111, e tutti i dispositivi della nostra rete CAN sono predisposti, tramite filtro per elaborare un messaggio con quell'identificativo.

Questo messaggio, una volta inviato ed elaborato dai nodi, causerà in risposta l'accensione di due LED (uno rosso ed uno verde) per la durata di 5 secondi su OGNI Slave correttamente funzionante.

Questa routine risulta importante sia in fase di test che di prototipazione/utilizzo per ovvi motivi.

Per i dettagli e la spiegazione del codice dei dispositivi Slave, fare riferimento al paragrafo relativo alla routine di ricezione slave (*sezione 4.2.3*).

#### 4.1.8 EUSART\_Alive\_check

Il carattere di ALIVE consiste in un byte (FF in esadecimale, ossia una sequenza di otto valori alti), che, in sostanza, circola in maniera più o meno periodica all'interno della rete. Quando uno dei due terminali riceve un carattere di ALIVE dalla rete RS-232, questo provvede a inviarlo di nuovo all'altro terminale. Il tempo limite imposto dal PIC Master per ricevere tale carattere è pari a circa 8,5 secondi.

La funzione che adempie a questo controllo temporale è denominata “EUSART\_Alive\_check”, definita in **Function\_Prootypes.h** e contenuta in **EUSART\_Managers.c**. Tale routine viene invocata nel main ogniqualvolta una variabile di stato dedicata (*Timer3OverflowFlag*) viene settata a 1.

*if (Timer3OverflowFlag==1)*

{ //Se i due registri del timer3 hanno raggiunto l'overflow...

EUSART\_Alive\_check(); /\*Controllo se sono passati 8,5 secondi dall'ultimo  
ALIVE ricevuto; se sì, effettuo lo switch\*/

}

Attraverso questa routine, il PIC Master deve semplicemente controllare se passano più o meno di 8,5 secondi dall'ultimo ALIVE ricevuto:

1. Se il PIC riceve un carattere di ALIVE entro 8,5 secondi da qualsivoglia terminale, allora non ci sono problemi; il PIC deve semplicemente inviare a sua volta il byte di ALIVE, e ricominciare a contare i secondi;
2. Se invece passano 8,5 secondi senza aver ricevuto un carattere di ALIVE, significa che il Raspberry/FPGA potrebbe avere qualche problema. Pertanto, attraverso la routine “EUSART\_Alive\_check”, il PIC Master deve controllare il multiplexer presente nella rete (d'ora in avanti definito switch per comodità) in modo che questo connetta lo stesso PIC Master con la scheda FPGA/Raspberry, escludendo di fatto il terminale difettoso dalla rete.

Nel PIC Master il controllo sul tempo della ricezione del carattere di ALIVE viene effettuato per mezzo del modulo Timer3, che si azzera ogniqualvolta raggiunge l'overflow. Su scala temporale, a partire da una frequenza di clock pari a 1 MHz, tra un overflow e il successivo passano all'incirca 2,1 secondi. Di seguito vengono riportati i conti nel dettaglio.

$$1) \text{ Frequenza di riferimento: } \frac{F_{osc}}{4} = 250 \text{ kHz}$$

$$2) \text{ Frequenza di conteggio del timer (prescaler 1:8): } \frac{250 \text{ kHz}}{8} = 31,25 \text{ kHz}$$

$$3) \text{ Tempo che intercorre tra un conteggio e l'altro: } \frac{1}{31,25 \text{ kHz}} = 32 \mu\text{s}$$

$$4) \text{ Tempo che intercorre tra due overflow (65536 incrementi): } 32 \mu\text{s} \times 65536 \approx 2.1 \text{ s}$$

Questo significa che ogni 2,1 secondi, a causa dell'avvenuto overflow del Timer3, un apposito flag (**TMR3IF**) viene settato a 1, e genera la seguente routine di interrupt (contenuta nel file **CAN\_Interrupt.c**).

Codice MPLAB:

*if (TMR3IF)*

{

*Timer3OverflowFlag=1; //Attivo la routine EUSART\_Alive\_check*

*TMR3IF=0; //Resetto il flag*

*TMR3IE=0; //Disattivo momentaneamente gli interrupt di questo tipo*

*//Resetto e stoppo il timer3*

*TMR3H=0;*

*TMR3L=0;*

*TMR3ON=0;*

```
}
```

All'interno di questa routine il timer e lo stesso flag di interrupt non vengono solo resettati, ma vengono anche spenti. Questo perché la routine “EUSART\_Alive\_check” deve essere conclusa prima che il Timer3 possa ripartire (e prima che l'overflow possa manifestarsi nuovamente). Inoltre, la routine di interrupt setta la variabile *Timer3OverflowFlag*. Da notare che è proprio quest'ultimo settaggio ad attivare la routine “EUSART\_Alive\_check” all'interno del main.

Si riporta dunque il codice relativo alla routine “EUSART\_Alive\_check”, sapendo che, come diretta conseguenza di quanto appena detto, viene attivata ad ogni overflow del Timer3. La descrizione accurata di ogni sua parte sarà fatta a partire dal paragrafo successivo.

Codice MPLAB:

```
void EUSART_Alive_check (void)
{ //Routine per il controllo sul tempo della ricezione del carattere di ALIVE
    if (counter!=3)
        { //Aggiorno il contatore (max 3 volte)
            counter=counter+1;
            TMR3IE=1; //Riabilito l'interrupt sull'overflow del Timer3...
            TMR3ON=1; //... e riattivo il Timer3 (ho finito la routine, quindi posso farlo qui)
        }
    else
        { /*Se il counter è già stato aggiornato 3 volte (ovvero sono passati 8,5 secondi), devo fare lo
           switch, nonché azzerare il contatore*/
            Switch_manager();
            counter=0;
        }
    Timer3OverflowFlag=0; //Azzero la variabile di attivazione
}
```

#### 4.1.8.1 Gestione dello switch: *Switch\_manager*

Quando avviene l'overflow del Timer3, all'interno della routine “EUSART\_Alive\_check” si aggiorna un contatore dedicato (*counter*), il quale è inizializzato a 0. Ad ogni aggiornamento di *counter*, la routine termina; il programma deve dunque ripristinare il funzionamento del Timer3 (settando *TMR3ON*) e del suo flag di interrupt (settando *TMR3IE*). Invece, quando *counter* raggiunge il valore 3, e passano ancora una volta 2,1 secondi (ovvero quando sono passati complessivamente circa 8,5 secondi), allora il programma deve cambiare le impostazioni dello switch, in modo da connettersi con il dispositivo di riserva. Se, ad esempio, il PIC Master è connesso al Raspberry (connessione di default), lo switch provvederà a collegarlo con la scheda FPGA. Ciò viene fatto invocando la funzione “*Switch\_manager*”. Tale subroutine è contenuta nel file **EUSART\_Managers.c** e definita nel file **Function\_Prootypes.h**. Da notare che il programma utilizza anche alcune variabili definite nel file **Def\_Library.h**

Codice MPLAB:

```
void Switch_manager (void)
{ /*Programma che gestisce lo switch presente nella comunicazione seriale, che
   consente di selezionare il dispositivo con cui comunica il PIC. Lo switch
   avviene ogniqualvolta il carattere di ALIVE non è stato ricevuto per 8,5
   secondi (questo controllo viene effettuato altrove; qui ci occupiamo di
   effettuare lo switch)*/
    if (Raspberry==1 && FPGA==0)
        { //Se il PIC è connesso al raspberry...
            CREN=0; //Disabilito la ricezione dati
        }
}
```

```

    TXEN=0; /*Disabilito anche la trasmissione dati (sto per scollegare la
              seriale, quindi è necessario disabilitare le due linee)*/
    S0=0;
    S1=0;
    FPGA=1; //... attacco l'fpga...
    Raspberry=0; //... e stacco il raspberry
    FSM_status=0; //Inoltre azzero lo stato; infatti i dati saranno irrimediabilmente corrotti
    switches=switches+1; //Controllo quanti switch consecutivi senza ALIVE sono stati effettuati
    if (switches>=2)
    { //Se sono già 2...
        Double_Switch=1; //...attivo la routine di visualizzazione dello stato di errore
    }
}
else if (Raspberry==0 && FPGA==1)
{ //In caso contrario, spengo la connessione con l'fpga e riattacco il raspberry
    CREN=0; //Disabilito la ricezione dati
    TXEN=0; /*Disabilito anche la trasmissione dati (sto per scollegare la
              seriale, quindi è necessario disabilitare le due linee)*/
    S0=1;
    S1=1;
    FPGA=0;
    Raspberry=1;
    FSM_status=0;
    switches=switches+1;
    if (switches>=2)
    {
        Double_Switch=1;
    }
}
else
{ //In tutti gli altri casi (es. all'avvio, in cui entrambe le variabili di stato siano nulle...)
    CREN=0; //Disabilito la ricezione dati
    TXEN=0; /*Disabilito anche la trasmissione dati (sto per scollegare la
              seriale, quindi è necessario disabilitare le due linee)*/
    S0=1; //...abilito il collegamento con il raspberry
    S1=1;
    Raspberry=1;
    FPGA=0;
}
SwitchStatusDisplay=1; //Attivo la variabile per il display dello stato
}

```

Nello specifico questa funzione gestisce tre diversi casi, determinati dalla combinazione delle variabili di stato dello switch (denominati *Raspberry* e *FPGA*, rispettivamente). In sostanza, ogniqualvolta il programma viene invocato, questo controlla lo stato attuale dello switch, e inverte di fatto la connessione, agendo direttamente sui segnali di controllo (*S0* e *S1*, che dal punto di vista hardware sono pin di uscita del PIC Master) e aggiornando di conseguenza le variabili di stato dello switch. Queste variabili di stato sono state implementate in funzione di un'altra routine denominata “EUSART\_Status\_Display”, la quale provvede a

visualizzare (attraverso un apposito LED RGB) l'attuale stato dello switch e, quindi, quale dispositivo è attualmente collegato con il PIC Master. Tale routine sarà trattata nel dettaglio nella *sezione 4.1.9*.

In aggiunta allo switch vero e proprio il programma:

1. disabilita la trasmissione/ricezione dei dati attraverso la rete RS-232, azzerando i rispettivi bit (*TXEN* e *CREN*). Questo deve essere fatto per prevenire l'invio/ricezione di eventuali dati durante lo switch effettivo. Tali bit saranno ripristinati (e quindi la rete seriale sarà nuovamente operativa) durante l'esecuzione della routine “EUSART\_Status\_Display” (per la trattazione della quale si rimanda alla *sezione successiva*).
2. azzerà lo stato della macchina a stati (“EUSART\_FSM”, la cui trattazione è rimandata alla *sezione 4.1.10*); questo perché è altamente probabile che gli eventuali dati in arrivo al momento dello switch risultino incompleti o corrotti.
3. aggiorna un contatore (*switches*), che viene incrementato ogni volta che viene effettuato uno switch. In sostanza, la variabile *switches* indica quanti switch sono stati effettuati dall'ultimo carattere di ALIVE ricevuto. Se la variabile *switches* raggiunge il valore 2, significa che sono stati effettuati due switch in questo modo; in altre parole, non è giunto un carattere di ALIVE entro il tempo stabilito né dal Raspberry né dalla scheda FPGA (ovvero sono passati 17 secondi, entro cui nessuno dei due terminali è riuscito a generare un ALIVE). Quando ciò accade, il programma setta la variabile di stato *Double\_Switch*. Questa variabile (come spiegato nella *prossima sezione*) attiva la condizione di accensione del LED blu di errore di “EUSART\_Status\_Display”; questa funzionalità è stata implementata per considerare eventuali routine future di gestione di questo tipo di problema.

Da notare che viene contemplato anche un terzo caso, ovvero quando entrambe le variabili di stato dello switch sono spente. Questa particolare condizione si verifica soltanto all'accensione del sistema, quando sia i LED di stato (e le variabili ad esso associate) sia lo switch sono spenti. In questo caso, il programma non fa altro che impostare la connessione di default, ossia con il Raspberry. Nel main, infatti, la funzione “Switch\_manager” viene invocata una volta, dopo il settaggio dei registri e l'inizializzazione delle variabili. Si riporta di seguito la parte di codice relativa al main in cui ciò accade.

*Switch\_manager(); //Effettuo il primo switch per settare il collegamento col raspberry all'avvio*

Come ultima cosa, il programma attiva la routine “EUSART\_Status\_Display”, attraverso il settaggio della variabile *SwitchStatusDisplay*.

#### *4.1.8.2 Gestione del carattere di ALIVE: EUSART\_ImAlive*

Per quanto detto finora, a prima vista questa routine potrebbe sembrare soltanto una sorta di cronometro, che ogni 8,5 secondi provvede a cambiare i segnali di controllo dello switch, senza curarsi dell'arrivo del carattere di ALIVE. Questo perché la gestione del carattere di ALIVE (e il conseguente azzeramento del Timer3 e del *counter*) viene fatta all'interno di un'altra routine del main (denominata “EUSART\_FSM”), che si attiva ogniqualvolta viene ricevuto un carattere in arrivo dal Raspberry/scheda FPGA. Si è deciso di implementare la gestione del carattere di ALIVE in “EUSART\_FSM” perché il carattere di ALIVE ha una priorità più alta rispetto al pacchetto dati; questo comporta la possibilità di trovare un carattere di ALIVE laddove il PIC dovrebbe aspettarsi un byte del pacchetto. In questo modo, il byte di ALIVE viene trattato sul software come uno dei possibili byte che il PIC Master può ricevere. Per questo motivo, si anticipa la trattazione relativa alla gestione del carattere di ALIVE in questo paragrafo (la gestione degli altri tipi di dato è rimandata alla *sezione 4.1.10*).

La funzione invocata in questo caso è denominata “EUSART\_ImAlive”, ed è implementata come subroutine all'interno di “Junk\_manager” e “Data\_manager”, a loro volta funzioni invocate in “EUSART\_FSM”. Il loro funzionamento sarà trattato rispettivamente nella *sottosezione 4.1.10.1* e nella *sottosezione 4.1.10.2*.

Di seguito viene riportato il codice sorgente di “EUSART\_ImAlive”, contenuto nel file **EUSART\_Managers.c**.

Codice MPLAB:

```
void EUSART_ImAlive (void)
```

```

{ //Funzione base per la trasmissione del carattere di ALIVE e azzeramento del Timer3
    while (TRMT==0);
    TXREG=ALIVE; //Trasmetto il carattere di ALIVE SENZA AGGIORNARE LO STATO
    counter=0; //Azzero il contatore relativo al Timer3
    switches=0; //Azzero il conteggio di switch consecutivi senza ALIVE
    if (Double_Switch==1)
        { //Se si è acceso il LED di errore per mancanza di ALIVE sia dal raspberry sia dalla scheda FPGA...
            Double_Switch=0; //...posso spegnerlo (qualunque dispositivo sia collegato al PIC, ora sta
                               funzionando)
        }
    TMR3H=0; //Infine azzero i due registri del Timer3
    TMR3L=0;
}

```

La subroutine “EUSART\_ImAlive” si limita a trasmettere il carattere di ALIVE, azzerando inoltre il Timer3 e tutti i flag e contatori ad esso associati (*counter* per ogni due secondi trascorsi senza ricevere un ALIVE; *switches* per il numero di switch consecutivi avvenuto senza la ricezione di almeno un ALIVE; *Double\_Switch* per spegnere il LED di errore associato alla variabile *switches*).

#### 4.1.9 EUSART\_Status\_Display

Come visto nella *sezione precedente*, la routine “EUSART\_Alive\_check” si occupa di controllare quanto tempo intercorre tra l’arrivo di un carattere di ALIVE e il successivo e, all’occorrenza, di cambiare l’altro terminale della rete quando quest’ultimo non invia un ALIVE dopo 8,5 secondi. Tuttavia, tale routine non consente di visualizzare in maniera concreta questi passaggi.

La visualizzazione di queste dinamiche viene deputata a un’altra routine del main, denominata “EUSART\_Status\_Display”, definita nel file **Function\_Protoypes.h** e contenuta in **EUSART\_Managers.c**. Come accennato altrove, questa routine viene attivata dopo aver commutato con successo la connessione tra PIC Master e Raspberry/scheda FPGA (attraverso la routine “Switch\_manager”, descritta nella *sottosezione 4.1.8.1*). Dal punto di vista software, ciò avviene quando la variabile *SwitchStatusDisplay* viene settata a 1.

```
if(SwitchStatusDisplay==1)
{
    EUSART_Status_Display(); //Se ci sono state variazioni nei canali di comunicazione dello switch, lo mostro
}
```

Si riporta il codice sorgente relativo a tale routine.

Codice MPLAB:

```
void EUSART_Status_Display (void)
{
    if (Raspberry==1 && FPGA==0 && Double_Switch==0)
    { //Visualizzazione collegamento raspberry
        LEDGreenPI=1;
        __delay_ms(50);
        LEDGreenPI=0;
        __delay_ms(50);
        LEDGreenPI=1;
        __delay_ms(50);
        LEDGreenPI=0;
    }
    else if (Raspberry==0 && FPGA==1 && Double_Switch==0)
    { //Visualizzazione collegamento fpga
        LEDRedFPGA=1;
        __delay_ms(50);
        LEDRedFPGA=0;
        __delay_ms(50);
        LEDRedFPGA=1;
        __delay_ms(50);
        LEDRedFPGA=0;
    }
    else if (Double_Switch==1)
    { //Visualizzazione stato d'errore
        LEDBlueSWITCH=1;
        __delay_ms(50);
        LEDBlueSWITCH=0;
        __delay_ms(50);
        LEDBlueSWITCH=1;
        __delay_ms(50);
    }
}
```

```

    LEDBlueSWITCH=0;
}
if (CREN==0)
{
    CREN=1; //Riabilito la ricezione
}
if (TXEN==0)
{
    TXEN=1; /*riabilito la trasmissione (lo switch è concluso e la rete
               ripristinata, quindi posso sia trasmettere che ricevere)*/
}
TMR3IE=1; //Riabilito l'interrupt sull'overflow del timer3...
TMR3ON=1; //... e riattivo il timer3 (ho finito la routine, quindi posso farlo qui)
SwitchStatusDisplay=0;
}

```

Una volta all'interno del programma, possono presentarsi tre casi, che corrispondono alle tre possibili condizioni di switch:

1. Lo switch ha appena collegato il PIC Master al Raspberry dopo 8,5 secondi senza ALIVE da parte della scheda FPGA. In questa condizione, la variabile *Raspberry* vale 1, mentre *FPGA* e *Double\_Switch* valgono 0, e viene fatto lampeggiare per due volte l'ingresso verde del LED RGB (collegato all'uscita denominata *LEDGreenPI*);
2. Lo switch ha appena collegato il PIC Master alla scheda FPGA dopo 8,5 secondi senza ALIVE da parte del Raspberry. In questo caso la variabile *FPGA* vale 1, mentre *Raspberry* e *Double\_Switch* valgono 0, e viene fatto lampeggiare per due volte l'ingresso rosso del LED RGB (collegato all'uscita denominata *LEDRedFPGA*);
3. Lo switch ha effettuato due commutazioni consecutive senza che il PIC abbia ricevuto un ALIVE. Quando questo avviene, *Double\_Switch* vale 1, indipendentemente dal valore delle altre variabili di stato. In questo caso viene fatto lampeggiare per due volte l'ingresso blu del LED RGB (collegato all'uscita denominata *LEDBlueSWITCH*).

Riassumendo:

1. Il LED RGB lampeggia di verde ogniqualvolta il PIC si collega al Raspberry. Questo avviene all'avvio, oppure a seguito di un guasto sulla rete di comunicazione con la scheda FPGA.
2. Il LED RGB lampeggia di rosso quando il PIC si collega alla scheda FPGA. Questo accade a seguito di un guasto sulla rete di comunicazione con il Raspberry.
3. Il LED RGB lampeggia di blu ad ogni cambio di collegamento, a partire dal secondo switch consecutivo senza ALIVE. Questo accade fintanto che uno dei due dispositivi non riprende a funzionare correttamente.

Infine, la routine ripristina la rete RS-232 (spenta da "Switch\_Manager") e il Timer3 (spento dalla routine di interrupt del timer stesso) settandone i relativi bit quando necessario (*TXEN*, *CREN*, *TMR3IE* e *TMR3ON*).

#### 4.1.10 EUSART\_FSM

Come già accennato altrove (*paragrafo 2.3*), un pacchetto dati è composto da una sequenza ordinata di dodici byte, comprensiva di identificativi (dati da scartare, definiti per semplicità junk) e di dati propriamente detti. Questo significa che è possibile identificare una variabile di stato che rappresenta in maniera univoca il tipo di byte che il PIC si aspetta di ricevere (stato 0 significa 1° byte del pacchetto; stato 1 significa 2° byte del pacchetto, ecc.).

La routine in questione, denominata “EUSART\_FSM”, altro non è che una macchina a stati costruita con questa logica, e si occupa della gestione di tutti i byte in arrivo al PIC attraverso il protocollo RS-232. Tale routine è direttamente invocata nel main.

```
if (read_RCREG==1)
{ //Se si è attivata la routine di interrupt...
    EUSART_FSM(); //Controllo se ho ricevuto un carattere; se sì, attivo la macchina a stati per la gestione
                    del pacchetto dati
}
```

Come si evince dal codice, il PIC Master entra in questa routine soltanto se il flag *read\_RCREG* è settato; questo accade nella routine di interrupt generata dalla ricezione di dati sul modulo EUSART del PIC (determinata dallo stato del flag *RCIF*). Nello specifico è stata implementata la seguente ISR (contenuta direttamente nella routine di interrupt nel file **Main\_Master.c**):

```
//EUSART interrupt
if (PIR1bits.RCIF==1)
{
    //Il buffer RCREG è pieno:...
    read_RCREG=1; //... attivo la routine di verifica del carattere sul main...
    incoming_data=RCREG; //... e leggo il dato appena ricevuto
}
```

Si riporta di seguito l'implementazione software della macchina a stati.

Codice MPLAB:

```
void EUSART_FSM (void)
{ //Routine per la gestione del pacchetto dati sulla seriale (implementazione con macchina a stati)
    switch (FSM_status)
    {
        case 0:
            Junk_manager (0xAA);
            break;
        case 1:
            Junk_manager (0x55);
            break;
        case 2:
            Junk_manager (0xB);
            break;
        case 3:
            Data_manager ();
            break;
        case 4:
            Data_manager ();
            break;
        case 5:
            Data_manager ();
```

```

        break;
case 6:
    Junk_manager (0xAA);
    break;
case 7:
    Junk_manager (0x55);
    break;
case 8:
    Junk_manager (0xC);
    break;
case 9:
    Data_manager ();
    break;
case 10:
    Data_manager ();
    break;
case 11:
    Junk_manager (0xD);
    if (error_flag==1)
    { //Se è avvenuto un errore di un qualche tipo...
        while (TRMT==0);
        TXREG=ERROR; //...trasmetto il messaggio di errore al raspberry/FPGA...
        error_flag=0; //...e azzero il flag d'errore
    }
    data_number=0; //Riazzero il contatore dati (ho finito i dati)
    break;
}
read_RCREG=0; //Disattivo la macchina a stati fino al prossimo carattere ricevuto
}

```

I byte ricevuti dal PIC Master possono essere raggruppati in tre categorie:

1. Byte di un pacchetto dati;
2. Carattere di ALIVE (FF);
3. Un carattere diverso (byte errato).

“EUSART\_FSM” fa uso della variabile di stato *FSM\_status*, che si può identificare come il numero di byte di un pacchetto dati che sono giunti dalla rete e sono già stati opportunamente trattati. A seconda della progressione dei dati all’interno di un pacchetto, ogni volta che viene attivata (attraverso il flag *read\_RCREG*), la macchina a stati svolge essenzialmente due operazioni:

1. Controlla il byte appena ricevuto (memorizzato nella variabile *incoming\_data*);
2. Invoca un’opportuna funzione a seconda dello stato (se il programma si aspetta un identificativo, invoca la funzione “Junk\_manager”; se si aspetta un dato, invoca la funzione “Data\_manager”).

Questo significa che, a seconda del valore della variabile *FSM\_status*, il PIC Master si aspetta un junk oppure un dato (a meno dell’ALIVE e di un byte errato), secondo lo schema riportato in tabella:

<b>Stato (FSM_status)</b>	0	1	2	3	4	5
<b>Tipo di byte (esadecimale)</b>	Junk (AA)	Junk (55)	Junk (B)	Dato (n°porta)	Dato (cifra psw)	Dato (cifra psw)
<b>Stato (FSM_status)</b>	6	7	8	9	10	11
<b>Tipo di byte (esadecimale)</b>	Junk (AA)	Junk (55)	Junk (C)	Dato (cifra psw)	Dato (cifra psw)	Junk (D)

Tabella 4.1.10-1 Struttura di un pacchetto dati

Una particolarità degna di nota si ha nell'ultimo stato (*FSM\_status*=11). In questo caso, dopo aver invocato "Junk\_manager", il programma effettua un controllo sul flag di errore (*error\_flag*): se vale 1, il PIC trasmette un messaggio di errore (per la precisione, il carattere EE in esadecimale) verso il Raspberry o la scheda FPGA (a seconda del dispositivo a cui è collegato), per poi azzerare tale flag; altrimenti non succede nulla. *error\_flag* è una variabile che, come spiegato più avanti, rappresenta un errore negli identificativi, oppure un dato non accettabile all'interno del pacchetto dati. Di conseguenza, si può già intuire come questo flag venga settato proprio in caso di dato corrotto in un qualche modo. Inoltre, in questo stato il programma azzerà la variabile *data\_number*, che tiene traccia del numero di dati del pacchetto già trattati dal programma. Questa istruzione costituisce una sicurezza in più, qualora il pacchetto dati contenga un byte errato.

In conclusione, la routine azzerà *read\_RCREG*; in questo modo, il programma esce dalla routine, per poi rientrare al prossimo carattere ricevuto.

#### 4.1.10.1 Gestione degli identificativi: Junk\_manager

Il primo tipo di byte che il pacchetto dati può fornire è quello degli identificativi (ovvero una serie di byte che indica l'inizio o la fine della trasmissione del pacchetto). La gestione di questi identificativi avviene attraverso la funzione "Junk\_manager", che svolge eventuali mansioni aggiuntive a seconda del byte a cui fa riferimento. Tale programma è contenuto nel file **EUSART\_Managers.c**, e viene dichiarato nel file **Function\_Prootypes.h**. Questa funzione sfrutta alcune delle variabili definite nel file **Def\_Library.h**.

Codice MPLAB:

```
void Junk_manager (unsigned char junk)
{ //Programma che gestisce i caratteri inutili (identificativi e fine msg)
    if (incoming_data==ALIVE)
        { //Se il carattere corrisponde a quello di ALIVE...
            EUSART_ImAlive(); //Ritrasmetto il carattere di ALIVE e azzero il Timer3
        }
    else
    {
        if (error_flag==1)
            { //Se si è già verificato un errore, aggiorno lo stato e non faccio nulla
                EUSART_FSM_refresh();
            }
        else
            { //Se invece non si è verificato alcun errore, svolgo la funzione di gestione dei junk
                if (incoming_data==junk)
                    { //Se il carattere corrisponde a quello dichiarato sopra...
                        switch (incoming_data)
                            { //Devo distinguere alcuni casi di carattere ricevuto, per poter svolgere delle
                                istruzioni aggiuntive relative al CAN
                                case 0xD: //junk=0xD
                                    TXREQ1=1;
                                    break;
                            }
            }
        }
    }
}
```

```

        EUSART_FSM_refresh();
    }
    else
    { //Se il carattere è qualcosa di diverso...
        error_flag=1; //...setto il flag d'errore e aggiorno lo stato
        EUSART_FSM_refresh();
    }
}
}

```

Il primo controllo effettuato dalla funzione è sull'arrivo di un carattere di ALIVE. La sua gestione avviene attraverso la routine “EUSART\_ImAlive”, di cui si è già ampiamente discusso nella *sottosezione 4.1.8.2*. Basti solo dire che questo è l'unico caso in cui lo stato della macchina a stati non viene aggiornato. Questo accade per il semplice fatto che nel Raspberry/scheda FPGA la routine di ALIVE, avendo una priorità più alta, può interrompere l'invio del pacchetto dati, per poi farlo ripartire una volta conclusa, ovvero il byte successivo all'ALIVE corrisponde a quello che sarebbe dovuto arrivare al suo posto. Se, ad esempio, il carattere di ALIVE arriva come 5° byte del pacchetto, il byte successivo corrisponde al 5° byte effettivo del pacchetto dati. Una volta terminata la routine “EUSART\_ImAlive”, termina anche “Junk\_manager”.

Qualora non sia sopraggiunto un carattere di ALIVE, la routine procede con un controllo sul valore del flag di errore (*error\_flag*). Se è settato, significa che si è precedentemente verificato un errore, e quindi il pacchetto dati in arrivo è già stato corrotto in un qualche modo: per questo motivo è fondamentale procedere con l'aggiornamento dello stato, senza svolgere altre funzioni. L'aggiornamento dello stato viene svolto dalla funzione “EUSART\_FSM\_refresh”, il cui codice sorgente (contenuto in **EUSART\_Managers.c**) è il seguente:

Codice MPLAB:

```

void EUSART_FSM_refresh (void)
{ //Funzione base per l'aggiornamento dello stato nella macchina a stati
    if (FSM_status<11)
        { //Se non mi trovo all'ultimo stato (n=11)...
            FSM_status = FSM_status +1; //...passo allo stato successivo
        }
    else
        { //Altrimenti...
            FSM_status =0; //...torno al primo stato
        }
}

```

Questa funzione implementa semplicemente un programma base che incrementa la variabile di stato *FSM\_status* (di fatto, aggiornandola) finché non raggiunge il valore 11 (12° e ultimo byte del pacchetto dati). Quando ciò avviene, la funzione azzerà questa variabile.

Se invece il flag di errore è azzerato, possono verificarsi due casi:

1. Il carattere ricevuto dalla rete RS-232 (*incoming\_data*) corrisponde al junk previsto dallo stato in cui viene invocata la funzione;
2. Il carattere è qualcosa di diverso, ovvero né un ALIVE (in quanto già discriminato in precedenza) né il junk previsto. In questo caso il byte giunto dal Raspberry/scheda FPGA è un carattere errato.

Nel primo caso, è giunto il carattere corretto: a seconda del byte ricevuto, il programma può svolgere alcune funzioni aggiuntive, che non hanno nulla a che vedere con il modulo EUSART, quanto piuttosto con il settaggio di alcuni bit relativi al modulo CAN. Questo controllo si effettua con un'istruzione *switch* sul carattere ricevuto. Nello specifico, quando il junk è il carattere D (13 in decimale; ultimo byte del pacchetto dati), il

programma setta il bit di richiesta di trasmissione dei dati caricati sul buffer 1 di trasmissione CAN ([TXREQ1](#)). Questo avviene perché i dati ricevuti in precedenza (e trattati dalla funzione “Data\_manager”, che verrà descritta a breve) sono già stati caricati sul buffer 1 di trasmissione CAN, e sono quindi pronti per essere inviati verso gli Slave.

Il secondo caso prevede la possibilità di un carattere errato, che non corrisponde né al carattere di junk associato allo stato né al carattere di ALIVE. In questo caso, il programma aggiorna lo stato e setta il flag di errore, il quale non viene azzerato fino alla fine dell’invio del pacchetto dati. In questo modo, quando la funzione viene invocata al prossimo carattere di junk, il programma si limiterà ad aggiornare lo stato fino alla fine dell’invio del pacchetto dati, prevenendo in questo modo la trasmissione di dati corrotti via CAN.

#### [4.1.10.2 Gestione dei dati: Data\\_manager](#)

Il secondo tipo di byte presente nel pacchetto dati in arrivo dalla comunicazione RS-232 è rappresentato dai dati veri e propri. Questi dati vengono gestiti dalla funzione “Data\_manager”, che svolge determinate funzioni a seconda del numero di dato a cui fa riferimento. Tale programma è contenuto nel file **EUSART\_Managers.c**, e viene dichiarato nel file **Function\_Prootypes.h**.

Codice MPLAB:

```
void Data_manager (void)
{ //Programma che gestisce i dati, caricandoli nei rispettivi buffer CAN
    if (incoming_data==ALIVE)
        { //Se il carattere corrisponde a quello di ALIVE...
            EUSART_ImAlive(); //...ritrasmetto il carattere di ALIVE e azzero il Timer3
        }
    else
    {
        if (error_flag==1)
            { //Se si è già verificato un errore, aggiorno lo stato e non faccio nulla
                EUSART_FSM_refresh();
            }
        else
            { //Se invece non si è verificato alcun errore, svolgo la funzione di gestione dei dati
                if (incoming_data<=0x9)
                    { //Se il carattere è accettabile...
                        switch (data_number)
                            { //...controllo quale dato è arrivato (se il primo, il secondo...)
                                case 0: //Primo dato (numero porta)
                                    TXB1SIDH=0;
                                    TXB1DLC=4;
                                    switch (incoming_data)
                                        { //Distinguo il dato ricevuto (a ogni dato corrisponde un numero porta, e quindi un diverso identificativo)
                                            case 1: //Porta 1
                                                TXB1SIDL=0b00100000;
                                                break;
                                            case 2: //Porta 2
                                                TXB1SIDL=0b01000000;
                                                break;
                                            case 3: //Porta 3
                                                TXB1SIDL=0b01100000;
                                                break;
                                        }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

        case 4: //Porta 4
            TXB1SIDL=0b10000000;
            break;
        default: //Un qualunque altro dato accettato prima
            (0, 5, 6, 7, 8, 9)
            error_flag=1;
            break;
    }
    data_number=data_number+1; //Aggiorno il contatore dati
    break;
case 1: //Secondo dato (prima cifra psw)
    TXB1D0=incoming_data; //Carico il dato sul registro CAN
    TXB1D0
    data_number=data_number+1; //Aggiorno il contatore dati
    break;
case 2: //Terzo dato (seconda cifra psw)
    TXB1D1=incoming_data; //Carico il dato sul registro CAN
    TXB1D1
    data_number=data_number+1; //Aggiorno il contatore dati
    break;
case 3: //Quarto dato (terza cifra psw)
    TXB1D2=incoming_data; //Carico il dato sul registro CAN
    TXB1D2
    data_number=data_number+1; //Aggiorno il contatore dati
    break;
case 4: //Quinto dato (quarta cifra psw)
    TXB1D3=incoming_data; //Carico il dato sul registro CAN
    TXB1D3
    data_number=0; //Riazzero il contatore dati (ho finito i dati)
    break;
}
EUSART_FSM_refresh();
}
else
{ //Se il carattere è qualcosa di diverso...
    error_flag=1; //Setto il flag d'errore e aggiorno lo stato
    EUSART_FSM_refresh();
}
}
}

```

Esattamente come nella sua controparte, il primo controllo che questa funzione effettua è relativo alla ricezione del carattere di ALIVE, e la sua gestione è esattamente identica: viene invocata la routine “EUSART\_ImAlive”, senza aggiornare lo stato. Quando ciò accade, la routine “Data\_manager” termina. Quando invece non è giunto alcun ALIVE, il programma controlla il valore del flag di errore (*error\_flag*), in perfetta analogia con “Junk\_manager”: se vale 1, il programma si limita ad aggiornare lo stato senza fare nulla; se vale 0, si possono presentare due casi:

1. Il carattere ricevuto (*incoming\_data*) è un dato accettabile, ovvero un valore compreso tra 0 e 9;

- Il carattere è errato (né un ALIVE né un dato).

Nel primo caso, il PIC Master ha ricevuto un dato ritenuto valido. In un pacchetto dati non corrotto sono sempre presenti 5 dati, nell'ordine:

- Il numero della porta a cui si vuole accedere (valore compreso tra 1 e 4);
- Prima cifra della password;
- Seconda cifra della password;
- Terza cifra;
- Quarta e ultima cifra.

A seconda del dato ricevuto, questo deve essere opportunamente trattato dal programma. Per questo motivo, occorre distinguere anche quale dei cinque dati è stato ricevuto. Questo avviene grazie al contatore *data\_number*, che viene aggiornato ogniqualvolta il dato è stato trattato nella maniera corretta. In ogni caso, dopo aver identificato il dato a cui si deve fare riferimento, viene effettuato un aggiornamento dello stato. In generale, per quanto riguarda le cifre della password (ovvero *data\_number* assume valori tra 1 e 4), queste devono essere caricate negli opportuni registri di trasmissione CAN, per poter essere ricevuti dal rispettivo PIC Slave. In particolare, tali dati vengono caricati sul buffer 1 di trasmissione CAN (registri *TXB1Dn*).

Nel caso il dato sia il numero di porta (ossia quando *data\_number* vale 0), il discorso è leggermente più complesso. Infatti, tale dato deve essere codificato come l'identificativo del messaggio CAN, in quanto i buffer di ricezione CAN dei diversi PIC Slave sono settati in modo da ricevere soltanto l'identificativo pari alla rispettiva porta (questo avviene nella routine “CAN\_Config\_Mode” dei PIC Slave stessi, di cui si è già ampiamente discusso nella sezione 4.1.5). Per questo motivo occorre distinguere anche tra i diversi dati possibili che indicano il numero della porta. Ciò viene svolto con l'istruzione *switch* sulla variabile *incoming\_data*: a seconda del dato ricevuto, il programma setta in maniera opportuna i registri relativi all'identificativo del buffer 1 di trasmissione CAN (*TXB1SIDH* e *TXB1IDL*). Se, ad esempio, l'utente seleziona la porta 3, l'identificativo del messaggio sarà quello che corrisponde al PIC Slave della porta 3. Inoltre, predispone la lunghezza del messaggio CAN da trasmettere (registro *TXB1DLC*) a 4 byte.

In aggiunta, il programma considera anche il caso in cui il numero di porta dovesse accidentalmente assumere valori sbagliati ma comunque accettati in precedenza (0, 5, 6, 7, 8, 9). In questo particolare caso, il programma setta il flag di errore e aggiorna lo stato, senza settare alcun identificativo.

Nel secondo caso, ovvero quando arriva un carattere invalido, il programma setta il flag di errore e aggiorna lo stato, esattamente come nella funzione “Junk\_manager”.

#### 4.1.11 Routine di ricezione CAN (Master)

Come detto in precedenza, ogni PIC del sistema è programmato per operare sulla base di una macchina a stati, gestita per la maggior parte attraverso segnali di interrupt, i quali permettono, al verificarsi di determinate condizioni di prendere gli opportuni provvedimenti.

È il caso anche della routine che gestisce la ricezione dei messaggi CAN in arrivo dalla rete. Quando un nuovo messaggio supera i filtri e viene salvato nel buffer di ricezione 0 per l'elaborazione, un segnale di interrupt viene generato ed il flag associato **RXB0IF** viene settato ad 1. Il blocco di codice nel file degli interrupt (**CAN\_Interrupt.c**) che gestisce questo accadimento e prende provvedimenti è il seguente.

Codice MPLAB:

```
if (RXB0IF)
{
    if (RXB0D0==80 && RXB0D1==80 && RXB0D2==80 && RXB0D3==80 && RXB0D4==80)
    {
        //C'è una porta aperta!
        A_Door_is_Open=1;
    }

    if (RXB0D0==11 && RXB0D1==11 && RXB0D2==11 && RXB0D3==11 && RXB0D4==11 &&
        Alive_Check==1)
    {
        //Il primo slave ha confermato di esserci!

        //Resetto il Timer1
        TMR1H=0; //Resetto la parte alta del Timer1
        TMR1L=0; //Resetto la parte bassa del Timer1
        TMR1ON=0; //Spengo il Timer1

        //Aggiorno lo stato
        Slave1_status=0;
        Check_Slave=2;
        Check_Req=1;
    }

    if (RXB0D0==22 && RXB0D1==22 && RXB0D2==22 && RXB0D3==22 && RXB0D4==22 &&
        Alive_Check==1)
    {
        //Il secondo slave ha confermato di esserci!

        //Resetto il Timer1
        TMR1H=0; //Resetto la parte alta del Timer1
        TMR1L=0; //Resetto la parte bassa del Timer1
        TMR1ON=0; //Spengo il Timer1

        //Aggiorno lo stato
        Slave2_status=0;
        Check_Slave=3;
        Check_Req=1;
    }
}
```

```

if (RXB0D0==33 && RXB0D1==33 && RXB0D2==33 && RXB0D3==33 && RXB0D4==33 &&
Alive_Check==1)
{
    //Il terzo slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Aggiorno lo stato
    Slave3_status=0;
    Check_Slave=4;
    Check_Req=1;
}
if (RXB0D0==44 && RXB0D1==44 && RXB0D2==44 && RXB0D3==44 && RXB0D4==44 &&
Alive_Check==1)
{
    //Il quarto slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Aggiorno lo stato
    Slave4_status=0;
    Check_Slave=5;
    Check_Req=1;
}
if (RXB0D0==55 && RXB0D1==55 && RXB0D2==55 && RXB0D3==55 && RXB0D4==55 &&
Alive_Check==1)
{
    //Il quinto slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Reset di tutte le variabili
    Slave5_status=0;
    Alive_Check=0; //Fine routine di Alive
    Check_Slave=0;
    Check_Req=0;

    //Resetto e riabilito il Timer0
    TMROL=0;
}

```

```

    TMROIE=1; //Riabilito gli interrupt del Timer0
    TMROON=1; //Riattivo il Timer0
}

//Azzero il Buffer di ricezione
RXBOD0=0;
RXBOD1=0;
RXBOD2=0;
RXBOD3=0;
RXBOD4=0;
RXBOD5=0;
RXBOD6=0;
RXBOD7=0;

RXBOFUL=0; //Azzero il flag, in modo da segnalare la possibilità di ricevere un nuovo messaggio
RXBOIF=0; //Azzero il flag che segnala l'arrivo di un nuovo messaggio
}

```

Come si evince facilmente dal codice, il check del messaggio ricevuto viene effettuato attraverso la valutazione del contenuto dei registri di ricezione contenuti nel buffer 0 con logica AND. L'insieme delle situazioni che il master deve gestire in termini di messaggi che in apparenza sembra comprendere molti casi, in realtà si riduce a due sole possibilità.

La prima eventualità riguarda il fatto che uno degli Slave del sistema stia segnalando la presenza di una porta aperta:

```
if (RXBOD0==80 && RXBOD1==80 && RXBOD2==80 && RXBOD3==80 && RXBOD4==80)
```

A seguito di ciò la variabile *A\_Door\_is\_Open* viene settata ad 1, in modo tale che nel main venga poi eseguita la seguente routine associata:

```

if (A_Door_is_Open==1)
{
    //Devo comunicare a Raspeberry/FPGA di bloccare gli accessi
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xDD;           //Trasmetto il codice di porta aperta
    A_Door_is_Open=0;      //Azzero la variabile
}

```

Il cui scopo è in definitiva informare il master di sistema che è presente appunto una porta già aperta (quindi un rischio per la sicurezza) attraverso la trasmissione di un ben definito carattere che identifica questa situazione.

La seconda eventualità riguarda invece la ricezione di messaggi che sono parte del meccanismo di richiesta-risposta che il PIC Master impiega per svolgere la routine di Alive CAN. Data la complessità e la ripartizione della suddetta, essa verrà trattata in una sezione a parte del documento.

//Azzero il Buffer di ricezione

```

RXBOD0=0;
RXBOD1=0;
RXBOD2=0;
RXBOD3=0;
RXBOD4=0;
RXBOD5=0;
RXBOD6=0;
RXBOD7=0;

```

```
RXB0FUL=0; //Azzero il flag, in modo da segnalare la possibilità di ricevere un nuovo messaggio  
RXB0IF=0; //Azzero il flag che segnala l'arrivo di un nuovo messaggio
```

Il rimanente codice si occupa di:

- Azzerare i registri del buffer di ricezione;
- Azzerare il bit *RXB0FUL* (che in realtà corrisponde al bit *RXFUL* del registro di controllo del buffer di ricezione 0) che è un flag che rappresenta lo stato del buffer 0, in particolare esso viene settato ad 1 nel momento in cui un nuovo messaggio viene salvato sul buffer (fintanto che resta ad 1 non è possibile ricevere altri messaggi su quel buffer);
- Azzerare il flag relativo a questo tipo di interrupt. Questo deve essere fatto solo dopo aver azzerato il bit *RXFUL* del buffer che si sta utilizzando, altrimenti il bit viene settato nuovamente in automatico;

#### 4.1.12 Routine di Alive CAN

Come precedentemente detto, alla routine di Alive è stato dedicato un paragrafo a parte del documento per via della sua ripartizione e complessità rispetto alle funzioni precedentemente analizzate, infatti come si vedrà a breve il codice che ne compone l'implementazione è stato ripartito su più file. Lo scopo di questa routine è effettuare periodicamente un controllo sullo stato di "salute"/operatività dei dispositivi connessi alla rete CAN. Nel nostro sistema questa periodicità viene scandita attraverso l'uso del Timer0, il quale, a seguito dell'Overflow, porta all'esecuzione della routine di Alive (ogni 67 secondi circa). Si tenga inoltre presente che il check dei dispositivi connessi alla rete CAN è stato implementato attraverso un meccanismo di Richiesta-Risposta sequenziale, ossia, il PIC Master scrive ad un dispositivo (Richiesta) per volta tramite rete CAN ed entro un certo tempo si aspetta di ricevere un messaggio contenente un particolare set di dati che identifica la condizione di salute del dispositivo stesso (Risposta). Se ciò non avviene per qualunque motivo, il PIC Master ha allora il compito di segnalare al master di sistema la presenza di un possibile guasto. Il caso di mancata risposta viene segnalato dall'Overflow di un secondo timer (Timer1) (Overflow in circa 2 secondi).

Passando ora al codice, il seguente blocco contiene il codice eseguito dal programma al momento dell'overflow del Timer0 (e conseguente interrupt):

Codice MPLAB:

```
if (TMROIF)
{
    Alive_Check=1; //E' il momento di fare il check!
    Check_Slave=1;
    Check_Req=1;
    TMROH=0;      //Azzero la parte alta del Timer0
    TMROL=0;      //Azzero la parte bassa del Timer0
    TMROON=0;     //Disattivo il Timer0
    TMROIE=0;     //Disattivo gli interrupt del Timer0
    TMROIF=0;
}
```

All'interno del blocco, vengono prima settate una serie di variabili necessarie al corretto svolgimento della routine, in particolare:

- *Alive\_Check*: è la variabile di stato associata alla routine di Alive, settarla ad 1 significa abilitare l'esecuzione del corrispondente blocco di codice contenuto nel file main;
- *Check\_Slave*: è la variabile di stato interna alla funzione, serve a identificare/aggiornare/tenere traccia del numero dello Slave da controllare;
- *Check\_Req*: è una variabile che viene settata quando è presente una effettiva richiesta di controllo, la sua introduzione è stata necessaria per (come si vedrà) evitare che il PIC Master invii più messaggi allo stesso slave mentre aspetta una sua eventuale risposta;

Si anticipa che per la corretta gestione di tutte le casistiche sono saranno poi introdotte altre variabili (di cui ora si riporta una sommaria descrizione):

- *Slave\_Statusn*: (La *n* al termine del nome ha la stessa funzione usata per i registri, ossia è un indice numerico). Questa variabile identifica lo stato di funzionamento (=0) o malfunzionamento (=1) di un dispositivo(*n*-esimo), e dunque ne è stata definita una per ogni Slave da controllare;
- *Broken\_Door*: Variabile che identifica la condizione per cui almeno uno dei dispositivi non ha risposto in tempo alla richiesta (ed è dunque malfunzionante);
- *Sn\_st\_Transmit*: Come nel caso delle variabili di stato, anche di queste ne esiste una per ogni Slave e servono sostanzialmente (se settate) a segnalare la necessità di trasmettere al master di sistema il malfunzionamento del o dei dispositivi che non hanno risposto in tempo alla richiesta;

Dopodiché avviene il reset e la disattivazione del Timer0, reset che come è stato anticipato deve avvenire secondo una certa modalità, in particolare:

1. Mentre il Timer è ancora attivo (on) deve essere azzerato prima *TMROH* ossia l'high byte del contatore;
2. Dopodiché si azzerà il low byte del contatore (*TMROL*=0);
3. Si disattiva il contatore (*TMROON*=0);

Si tenga presente che questa prassi va seguita in generale per ogni Timer.

Infine, le ultime due righe servono a disattivare temporaneamente gli interrupt di questo tipo e a resettare il corrispondente flag.

Questo porta nel main all'esecuzione del seguente blocco di codice:

```
if (Alive_Check==1)
{
    if (Check_Req==1)
    {
        Are_you_there(); //Chiamo la routine di invio Alive
    }
}
```

Che, come visto diverse volte in precedenza, non è altro che una chiamata ad una funzione il cui corpo è definito in un altro file (**CAN\_Functions.c** in questo caso), la quale in realtà viene chiamata solo se entrambe le variabili *Check\_Req* ed *Alive\_Check* sono settate ad 1. Nel seguito si riporta il codice della funzione di cui poi verrà svolta l'analisi/commento.

Codice MPLAB:

```
void Are_you_there(void)
{
    switch (Check_Slave)
    {
        case 1: //Scrivo al primo slave, se è attivo

            if (Slave1_status==0) //Se era attivo dal ciclo precedente, gli scrivo
            {
                //Imposto la priorità dei messaggi
                TXB0CONbits.TXPRIO=1;
                TXB0CONbits.TXPRI1=1;

                //Imposto l'identificativo messaggio
                TXB0SIDH=0b00000000;
                TXB0IDL=0b00100000;

                //Setto la lunghezza del messaggio
                TXB0DLC=0b00000101;

                //Imposto il contenuto dei registri da inviare
                //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
                TXB0D0=11; //Scrivo nel registro 0 del buffer 0
                TXB0D1=11; //Scrivo nel registro 1 del buffer 0
                TXB0D2=11; //Scrivo nel registro 2 del buffer 0
                TXB0D3=11; //Scrivo nel registro 3 del buffer 0
            }
    }
}
```

```

TXB0D4=11; //Scrivo nel registro 4 del buffer 0

Check_Req=0; //Setto a zero la richiesta in modo da non mandare più messaggi allo stesso slave

//Setto il bit di trasmissione
TXREQ0=1;

//Faccio partire il Timer1
TMR1ON=1;
}

else
{
    //Aggiorno lo stato
    Check_Slave=2; //Salto questo step e vado al successivo
    Check_Req=1;
    Slave1_status=0; //Riaggiorno la variabile di stato per riprovare l'invio al ciclo di Alive successivo
}

break;

case 2: //Scrivo al secondo slave, se è attivo

if (Slave2_status==0)
{
    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRI0=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXB0SIDH=0b00000000;
    TXB0IDL=0b01000000;

    //Setto la lunghezza del messaggio
    TXB0DLC=0b00000101;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=22; //Scrivo nel registro 0 del buffer 0
    TXB0D1=22; //Scrivo nel registro 1 del buffer 0
    TXB0D2=22; //Scrivo nel registro 2 del buffer 0
    TXB0D3=22; //Scrivo nel registro 3 del buffer 0
    TXB0D4=22; //Scrivo nel registro 4 del buffer 0

    Check_Req=0; //Setto a zero la richiesta in modo da non mandare più messaggi allo stesso slave
}

```

```

    //Setto il bit di trasmissione
    TXREQ0=1;

    //Faccio partire il Timer1
    TMR1ON=1;
}

else //Se invece al ciclo precedente è risultato malfunzionante.... per questo ciclo lo salto
{
    //Aggiorno lo stato
    Check_Slave=3; //Salto questo step e vado al successivo
    Check_Req=1;
    Slave2_status=0; //Raggiorno la variabile di stato per riprovare l'invio al ciclo di
    Alive successivo
}

break;

```

*case 3: //Scrivo al terzo slave, se è attivo*

```

if(Slave3_status==0)
{
    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRI0=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXB0SIDH=0b00000000;
    TXB0SIDL=0b01100000;

    //Setto la lunghezza del messaggio
    TXB0DLC=0b00000101;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=33; //Scrivo nel registro 0 del buffer 0
    TXB0D1=33; //Scrivo nel registro 1 del buffer 0
    TXB0D2=33; //Scrivo nel registro 2 del buffer 0
    TXB0D3=33; //Scrivo nel registro 3 del buffer 0
    TXB0D4=33; //Scrivo nel registro 4 del buffer 0

    Check_Req=0; //Setto a zero la richiesta in modo da non mandare più messaggi allo
    stesso slave

    //Setto il bit di trasmissione
    TXREQ0=1;

    //Faccio partire il Timer1
    TMR1ON=1;
}

```

```

        }
    else
    {
        //Aggiorno lo stato
        Check_Slave=4; //Salto questo step e vado al successivo
        Check_Req=1;
        Slave3_status=0; //Raggiorno la variabile di stato per riprovare l'invio al ciclo di
        Alive successivo
    }

    break;
}

```

*case 4: //Scrivo al quarto slave, se è attivo*

```

if(Slave4_status==0)
{
    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRI0=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXBOSIDH=0b00000000;
    TXBOSIDL=0b10000000;

    //Setto la lunghezza del messaggio
    TXB0DLC=0b00000101;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=44; //Scrivo nel registro 0 del buffer 0
    TXB0D1=44; //Scrivo nel registro 1 del buffer 0
    TXB0D2=44; //Scrivo nel registro 2 del buffer 0
    TXB0D3=44; //Scrivo nel registro 3 del buffer 0
    TXB0D4=44; //Scrivo nel registro 4 del buffer 0

    Check_Req=0; //Setto a zero la richiesta in modo da non mandare più messaggi allo
    stesso slave

    //Setto il bit di trasmissione
    TXREQ0=1;

    //Faccio partire il Timer1
    TMR1ON=1;
}

else
{
    //Aggiorno lo stato
    Check_Slave=5; //Salto questo step e vado al successivo
}

```

```

    Check_Req=1;
    Slave4_status=0; //Raggiorno la variabile di stato per riprovare l'invio al ciclo di
    Alive successivo
}

break;

case 5: //Scrivo al quinto slave, se è attivo

if(Slave5_status==0)
{
    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRI0=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXB0SIDH=0b00000000;
    TXB0SIDL=0b10100000;

    //Setto la lunghezza del messaggio
    TXB0DLC=0b00000101;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=55; //Scrivo nel registro 0 del buffer 0
    TXB0D1=55; //Scrivo nel registro 1 del buffer 0
    TXB0D2=55; //Scrivo nel registro 2 del buffer 0
    TXB0D3=55; //Scrivo nel registro 3 del buffer 0
    TXB0D4=55; //Scrivo nel registro 4 del buffer 0

    Check_Req=0; //Setto a zero la richiesta in modo da non mandare più messaggi allo
    stesso slave

    //Setto il bit di trasmissione
    TXREQ0=1;

    //Faccio partire il Timer1
    TMR1ON=1;
}

else
{
    //Reset di tutte le variabili
    Alive_Check=0; //Fine routine di Alive
    Check_Slave=0;
    Slave5_status=0; //Raggiorno la variabile di stato per riprovare l'invio al ciclo di
    Alive successivo

    //Resetto e riabilito il Timer0
}

```

```

    TMROL=0; //Resetto la parte bassa del Timer0
    TMROIE=1; //Riabilito gli interrupt del Timer0
    TMROON=1; //Riattivo il Time0
}

break;

}

}

```

La funzione sopra-riportata può sembrare, almeno in un primo momento, piuttosto articolata, in realtà è molto semplice in quanto i 5 *case* riportati sono molto simili, con la sola eccezione del quinto, in quanto esso chiude il ciclo della routine di Alive (almeno in un caso, si veda trattazione successiva). Successivamente sarà preso a riferimento il codice del primo *case* e in parte quello del quinto per l'analisi.

La prima cosa da notare è che la variabile di stato dello *switch*, non è altri che la variabile *Check\_Slave*, introdotta in precedenza, la quale identifica il numero dello Slave a cui devo fare richiesta di conferma di operatività.

```

switch (Check_Slave)
{

```

*case 1: //Scrivo al primo slave, se è attivo*

In ogni *case* viene poi fatto il check dello stato del dispositivo attraverso la variabile apposita (*Slave\_Statusn*):

*if (Slave1\_status==0) //Se era attivo dal ciclo precedente, gli scrivo*

Se il dispositivo risulta attivo sin dal precedente ciclo di richieste (o di default, se siamo al primo richiamo della routine) allora viene effettuata nuovamente la richiesta di conferma di operatività tramite la scrittura di un messaggio CAN (a cui ogni Slave deve rispondere con un proprio messaggio univoco, si veda la *sezione 4.2.3* per maggiori dettagli ed esempio) e subito dopo la richiesta di invio viene attivato il Timer1, il quale serve a dare il tempo massimo per la risposta dello Slave (circa 2 secondi). Si noti che prima di inviare il messaggio viene azzerata momentaneamente (in ogni *case*) la variabile *Check\_Req*, questo serve ad evitare che il PIC Master possa inviare più messaggi/richieste allo Slave considerato, mentre attende una sua conferma (o allo stesso modo mentre attende l'attivazione dell'interrupt di Overflow del Timer1 nel caso lo Slave fosse malfunzionante).

Nel caso invece (*else*) il dispositivo risultasse non attivo dal ciclo precedente, allora il sistema semplicemente salta il dispositivo, ma ne aggiorna lo stato riportandolo come "attivo" in modo tale che al ciclo di Alive successivo il PIC Master provi a verificare nuovamente lo stato dello Slave:

```

//Aggiorno lo stato
Check_Slave=2; //Salto questo step e vado al successivo
Check_Req=1;
Slave1_status=0; //Riaggiorno la variabile di stato per riprovare l'invio al ciclo di
Alive successivo

```

Come detto in precedenza tutti i *case* sono simili, ancor di più se si considera solo il caso in tutti i dispositivi siano attivi, a meno delle ovvie variazioni nell'identificativo del messaggio CAN e del contenuto dei dati (compreso il quinto). La trattazione del caso di "Dispositivo non attivo" cambia nel caso del quinto *case*, in quanto è quello che chiude di fatto il ciclo della routine Alive (nel caso esso sia attivo, a chiudere la routine di Alive è la sotto-funzione che ne elabora la risposta in realtà):

```

else
{
    //Reset di tutte le variabili
}

```

```

    Alive_Check=0; //Fine routine di Alive
    Check_Slave=0;
    Slave5_status=0; //Raggiorno la variabile di stato per riprovare l'invio al ciclo di
    Alive successivo

    //Resetto e riabilito il Timer0
    TMROL=0; //Resetto la parte bassa del Timer0
    TMROIE=1; //Riabilito gli interrupt del Timer0
    TMROON=1; //Riattivo il Time0
}

```

Come si può notare, nel caso appena descritto, viene fatto il reset di tutte le variabili del ciclo in modo da poterlo ripetere e vengono riattivati il Timer0 e la possibilità di interrupt su Overflow di quest'ultimo.

Ad ogni modo ora la trattazione deve, per forza di cose, biforcarsi nel senso che ovviamente i casi di "Dispositivo n conferma in tempo di essere attivo" e "Dispositivo n non conferma in tempo" dovranno essere trattati separatamente in quanto coinvolgono funzioni e azioni diverse.

Poniamoci ora nel caso semplice ed ottimale in cui tutto vada secondo le aspettative di progetto, per cui ogni Slave riesce a rispondere in tempo alla richiesta del PIC Master. La subroutine che si occupa di gestire la risposta in arrivo dagli Slave è contenuta nella routine di interrupt che gestisce tutti i messaggi in arrivo dalla rete CAN (si veda la sezione dedicata alla ricezione messaggi per maggiori informazioni). Questa routine include al proprio interno anche le casistiche relative alle risposte di ogni Slave del sistema, il cui corrispondente codice viene riportato nel seguito:

Codice MPLAB:

```

if (RXB0D0==11 && RXB0D1==11 && RXB0D2==11 && RXB0D3==11 && RXB0D4==11 && Alive_Check==1)
{
    //Il primo slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Aggiorno lo stato
    Slave1_status=0;
    Check_Slave=2;
    Check_Req=1;
}

if (RXB0D0==22 && RXB0D1==22 && RXB0D2==22 && RXB0D3==22 && RXB0D4==22 &&
Alive_Check==1)
{
    //Il secondo slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Aggiorno lo stato
    Slave2_status=0;
}

```

```

Check_Slave=3;
Check_Req=1;
}
if (RXB0D0==33 && RXB0D1==33 && RXB0D2==33 && RXB0D3==33 && RXB0D4==33 &&
Alive_Check==1)
{
    //Il terzo slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Aggiorno lo stato
    Slave3_status=0;
    Check_Slave=4;
    Check_Req=1;
}
if (RXB0D0==44 && RXB0D1==44 && RXB0D2==44 && RXB0D3==44 && RXB0D4==44 &&
Alive_Check==1)
{
    //Il quarto slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Aggiorno lo stato
    Slave4_status=0;
    Check_Slave=5;
    Check_Req=1;
}
if (RXB0D0==55 && RXB0D1==55 && RXB0D2==55 && RXB0D3==55 && RXB0D4==55 &&
Alive_Check==1)
{
    //Il quinto slave ha confermato di esserci!

    //Resetto il Timer1
    TMR1H=0; //Resetto la parte alta del Timer1
    TMR1L=0; //Resetto la parte bassa del Timer1
    TMR1ON=0; //Spengo il Timer1

    //Reset di tutte le variabili
    Slave5_status=0;
    Alive_Check=0; //Fine routine di Alive
    Check_Slave=0;
    Check_Req=0;
}

```

```

//Resetto e riabilito il Timer0
TMR0L=0;
TMROIE=1; //Riabilito gli interrupt del Timer0
TMROON=1; //Riattivo il Timer0
}

```

Come per la precedente funzione, anche in questa, le azioni prese ad ogni step sono piuttosto simili, e consistono in sostanza nel:

- Resetare il Timer1 (per poterlo riutilizzare alla prossima richiesta);

```

TMR1H=0; //Resetto la parte alta del Timer1
TMR1L=0; //Resetto la parte bassa del Timer1
TMR1ON=0; //Spengo il Timer1

```

- Aggiornare lo stato delle variabili in gioco per poi proseguire nell'esecuzione della routine, in particolare viene aggiornato/confermato lo stato del dispositivo, la variabile di stato del ciclo di check viene aggiornata ed infine viene settata ad 1 la variabile *Check\_Req* (caso del primo Slave ripotato):

```

Slave1_status=0;
Check_Slave=2;
Check_Req=1;

```

Ancora una volta, come si evince dal codice, nel caso ci trovassimo a valutare la quinta conferma (quindi l'ultimo Slave della lista), allora alle azioni precedenti si dovrebbero aggiungere le azioni atte al reset di tutte le variabili e dei Timer (nonché la riattivazione del Timer0) in modo da permettere il corretto svolgimento ciclico della routine in futuro (vedi codice sopra riportato). Nel caso quindi tutto funzioni correttamente la routine di Alive di riduce ad un sequenziale rimbalzo del codice tra le due funzioni "Are\_you\_there" e "Master\_Receive\_Routine".

Passiamo ora invece a considerare l'altro ramo della biforcazione, ossia il caso in cui almeno uno degli Slave non risponda entro il tempo limite alla richiesta del PIC Master. In questo caso in seguito all'Overflow del Timer1 viene generato il corrispondente interrupt e viene eseguita l'opportuna parte del seguente blocco di codice:

Codice MPLAB:

```

if (TMR1IF)
{
    Broken_Door=1; //Qualcosa non va quindi devo informare il Controllore di sistema
    switch (Check_Slave)
    {
        case 1:
            Slave1_status=1; //Il dispositivo è guasto o in ritardo
            S1_st_Transmit=1; //Devo informare il master di sistema che la porta è
                               //malfunzionante
            Check_Slave=2; //Aggiorno lo stato
            Check_Req=1;

            //Resetto il Timer1
            TMR1H=0; //Resetto la parte alta del Timer1
            TMR1L=0; //Resetto la parte bassa del Timer1
            TMR1ON=0; //Spengo il Timer1
            break;
    }
}

```

*case 2:*

```
Slave2_status=1; //Il dispositivo è guasto o in ritardo
S2_st_Transmit=1; //Devo informare il master di sistema che la porta è
malfunzionante
Check_Slave=3; //Aggiorno lo stato
Check_Req=1;

//Resetto il Timer1
TMR1H=0; //Resetto la parte alta del Timer1
TMR1L=0; //Resetto la parte bassa del Timer1
TMR1ON=0; //Spengo il Timer1
break;
```

*case 3:*

```
Slave3_status=1; //Il dispositivo è guasto o in ritardo
S3_st_Transmit=1; //Devo informare il master di sistema che la porta è
malfunzionante
Check_Slave=4; //Aggiorno lo stato
Check_Req=1;

//Resetto il Timer1
TMR1H=0; //Resetto la parte alta del Timer1
TMR1L=0; //Resetto la parte bassa del Timer1
TMR1ON=0; //Spengo il Timer1
break;
```

*case 4:*

```
Slave4_status=1; //Il dispositivo è guasto o in ritardo
S4_st_Transmit=1; //Devo informare il master di sistema che la porta è
malfunzionante
Check_Slave=5; //Aggiorno lo stato
Check_Req=1;

//Resetto il Timer1
TMR1H=0; //Resetto la parte alta del Timer1
TMR1L=0; //Resetto la parte bassa del Timer1
TMR1ON=0; //Spengo il Timer1
break;
```

*case 5:*

```
Slave5_status=1; //Il dispositivo è guasto o in ritardo
S5_st_Transmit=1; //Devo informare il master di sistema che la porta è
malfunzionante

//Resetto il Timer1
TMR1H=0; //Resetto la parte alta del Timer1
TMR1L=0; //Resetto la parte bassa del Timer1
```

```

TMR1ON=0; //Spengo il Timer1

//Reset di tutte le variabili
Alive_Check=0; //Fine routine di Alive
Check_Slave=0;
Check_Req=0;

//Resetto e riabilito il Timer0
TMROL=0;
TMROIE=1; //Riabilito gli interrupt del Timer0
TMROON=1; //Riattivo il Timer0
break;

}
TMR1IF=0; //Azzero il flag
}

```

Questo blocco si occupa in primis di settare la variabile *Broken\_Door* (del cui uso si parlerà più diffusamente nel seguito del paragrafo), dopodiché, tramite l'impiego un costrutto *switch* (la cui variabile di controllo è ancora una volta la variabile di stato del ciclo) vengono settate le opportune variabili di stato corrispondenti allo step del ciclo di *Alive* a cui il programma è arrivato.

Prendiamo per esempio il caso in cui il primo Slave non risponda entro il tempo limite alla richiesta del PIC Master, in questo caso al settarsi dell'interrupt verrebbe eseguito il codice contenuto nel primo *case* della lista, il quale in sequenza:

- Aggiorna la variabile di stato del dispositivo per evidenziarne la NON-operatività;
- Slave1\_status=1; //Il dispositivo è guasto o in ritardo*
- Setta ad 1 la corrispondente variabile di stato che indica la necessità di comunicare il guasto del dispositivo considerato al master di sistema;
- S1\_st\_Transmit=1; //Devo informare il master di sistema che la porta è malfunzionante*
- Aggiorna lo stato del ciclo di *Alive* (in modo da proseguire con il check);
- Check\_Slave=2; //Aggiorno lo stato*  
*Check\_Req=1;*
- Infine, resetta il Timer1 per il nuovo utilizzo allo step successivo;

Tutti i *case* sono, ancora una volta, molto simili tra loro, con la solita eccezione del quinto, il quale chiudendo il ciclo della routine deve preoccuparsi anche del reset di tutte le variabili/Timer in gioco in modo da assicurare il corretto comportamento ciclico dell'*Alive*. Alla fine della funzione (dopo il costrutto *switch*) viene poi azzerato il flag associato a questo tipo di interrupt:

*TMR1IF=0; //Azzero il flag*

Ora, in precedenza è stato detto che la prima cosa che avviene quando questo interrupt viene attivato è il settaggio della variabile *Broken\_Door*, essa ha come effetto l'esecuzione di un particolare blocco di codice nel main:

```

if(Broken_Door==1)
{
    Broken_Door_Signal(); //Chiamo la routine di trasmissione porta rotta
    Broken_Door=0; //Azzero la variabile di stato
}

```

Che ancora una volta coincide con la chiamata ad una funzione, il cui corpo è contenuto in un altro file (**General\_Functions.c** in questo caso) e una volta svolta la funzione viene poi resettata la variabile di stato associata.

Codice MPLAB:

```
void Broken_Door_Signal (void)
{
    if (S1_st_Transmit==1)
    {
        while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
        TXREG=0xD1;
        S1_st_Transmit=0;          //Ho informato il master della situazione, azzero la variabile
    }
    else if (S2_st_Transmit==1)
    {
        while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
        TXREG=0xD2;
        S2_st_Transmit=0;          //Ho informato il master della situazione, azzero la variabile
    }
    else if (S3_st_Transmit==1)
    {
        while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
        TXREG=0xD3;
        S3_st_Transmit=0;          //Ho informato il master della situazione, azzero la variabile
    }
    else if (S4_st_Transmit==1)
    {
        while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
        TXREG=0xD4;
        S4_st_Transmit=0;          //Ho informato il master della situazione, azzero la variabile
    }
    else if (S5_st_Transmit==1)
    {
        while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
        TXREG=0xD5;
        S5_st_Transmit=0;          //Ho informato il master della situazione, azzero la variabile
    }
}
```

Questa funzione ha lo scopo di capire quale dispositivo non ha risposto in tempo (attraverso il check delle variabili *Sn\_st\_Transmit* e in base al dispositivo, inviare un messaggio tramite modulo EUSART al master di sistema contenente una cifra identificativa che codifica il numero dello Slave bollato come malfunzionante:

```
while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
TXREG=0xD1;
```

Il ciclo **while** alla prima riga serve a trattenere il programma qualora un altro messaggio fosse già in fase di invio fino allo svuotamento del registro di trasmissione del modulo EUSART (che segnala la possibilità di inviare un nuovo messaggio, che coincide con la condizione: **TRMT!=1 && TXEN!=1**). Infine, viene resettata la variabile *Sn\_st\_Transmit* del dispositivo malfunzionante, in modo tale che la funzione non invii messaggi

ridondanti (essendo poi implementata con un costrutto *if – else if* un solo blocco di codice viene eseguito per volta).

Questo chiude l'analisi della routine di Alive del nostro sistema, la successiva Fig.4.1.12-1 contiene uno schema riassuntivo ad alto livello della routine appena descritta.

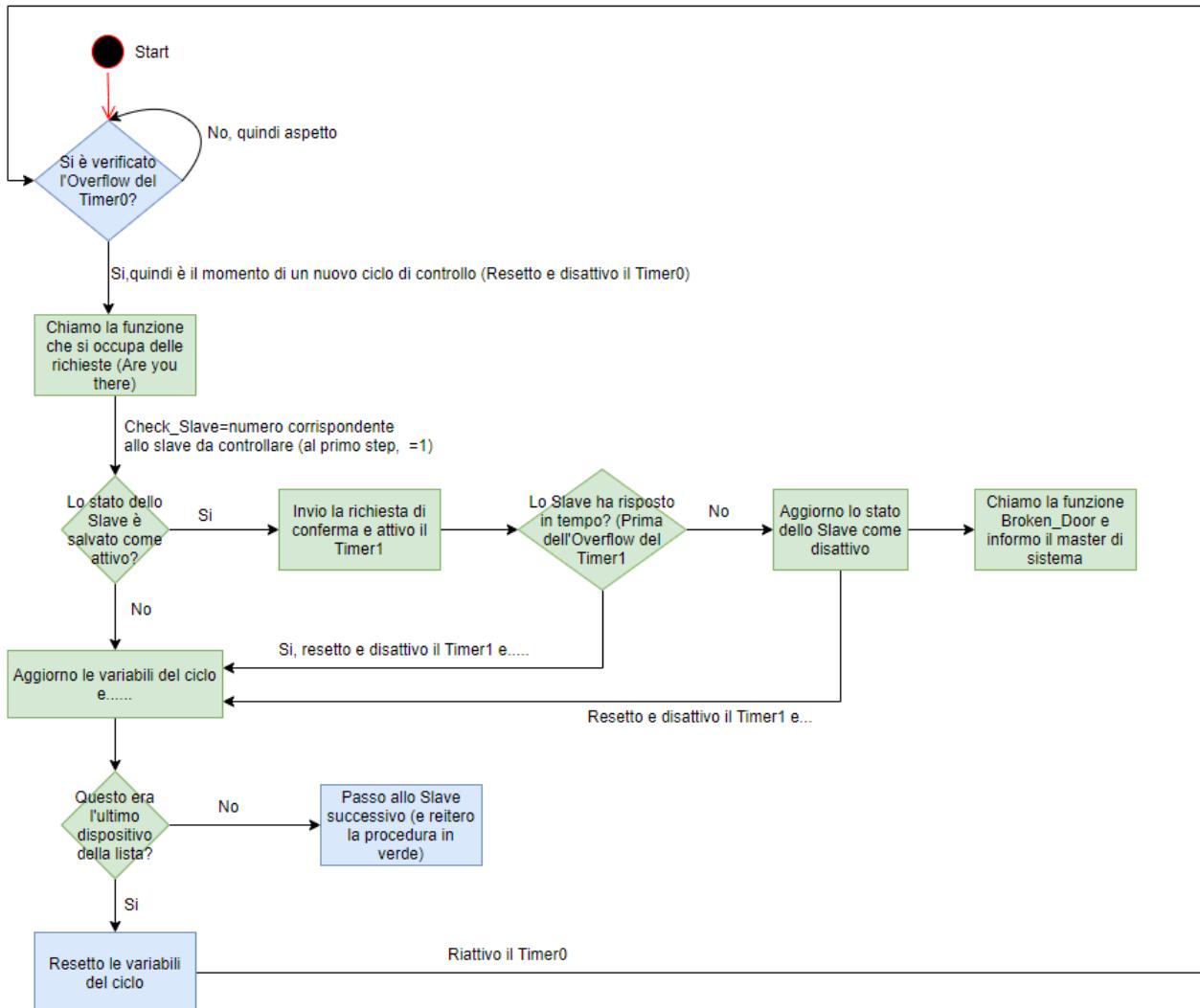


Fig.4.1.12-1 Schema riassuntivo routine di Alive CAN

#### 4.1.13 CAN Errors Handling (Master)

Come ulteriore aggiunta alle funzionalità di sistema è stato implementato su tutti i dispositivi PIC un meccanismo di gestione degli errori CAN (gestione degli errori del modulo CAN, diversa dalla gestione delle situazioni critiche). La gestione generale di questi errori, così come di quasi tutte le situazioni previste affrontate fino a questo punto, avviene attraverso l'uso di alcune variabili di stato che vengono settate a seguito del verificarsi di determinati interrupt, abilitati dal registro **PIE3**.

**REGISTER 10-9: PIE3: PERIPHERAL INTERRUPT ENABLE REGISTER 3**

Mode 0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IRXIE	WAKIE	ERRIE	TXB2IE	TXB1IE <sup>(1)</sup>	TXB0IE <sup>(1)</sup>	RXB1IE	RXB0IE
Mode 1,2	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	IRXIE	WAKIE	ERRIE	TXBnIE	TXB1IE <sup>(1)</sup>	TXB0IE <sup>(1)</sup>	RXBnIE	FIFOWMIE <sup>(1)</sup>

bit 7	bit 0
-------	-------

Fig. 4.1.13-1 Registro PIE3 per gestione errori (bit IRXIE e ERRIE)

Il codice relativo a questi settaggi è, come per i precedenti casi contenuto nell'apposita cartella **CAN\_Interrupt.c**:

Codice MPLAB:

```
if (IRXIF)
{
    //Cosa faccio se arriva un messaggio non valido?
    CAN_Invalid_Msg=1;
    Counter_Inv_Msg++;
    IRXIF=0; //Azzero il flag
    IRXIE=0; //Devo disattivare temporaneamente questi interrupt per poter agire
}
if (ERRIF)
{
    //Cosa faccio se si verifica un errore sul CAN?
    CAN_Error_Check=1;
    ERRIF=0; //Azzero il flag
    ERRIE=0; //Devo disattivare temporaneamente questi interrupt per poter agire
}
```

Come si può notare, dal punto di vista interrupt, la gestione degli errori è ridotta al check di due possibili fonti di malfunzionamento:

- La prima fonte sono gli errori che si verificano in fase di ricezione (o trasmissione) di messaggi CAN;
- La seconda raccoglie in realtà tutta una serie di altri possibili errori CAN di vario genere. Errori che comunque, a prescindere dalla propria natura, causano l'incremento dei registri di conteggio errori CAN.

Prima di affrontare nel dettaglio quello che sarà il funzionamento delle due routine si riporta nel seguito il codice contenuto nel file main del dispositivo Master (**Main\_Master.c**) (La gestione degli errori è presente anche su tutti gli Slave del sistema, ma in versione ridotta, come sarà spiegato nel seguito).

Codice MPALB:

```
if (CAN_Invalid_Msg==1)
{
    if (Counter_Inv_Msg<10)
    {
```

```

    LEDRedCAN=1;
    __delay_ms(50);
    LEDRedCAN=0;
    __delay_ms(50);
    LEDRedCAN=1;
    __delay_ms(50);
    LEDRedCAN=0;
    __delay_ms(50);
    CAN_Invalid_Msg=0;
}
else
{
    Counter_Inv_Msg=0; //Azzero il counter
    //Devo comunicare a Raspberry/FPGA che è stato ricevuto ripetutamente un
    messaggio invalido
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE1; //Trasmetto il codice di "troppi messaggi invalidi", Errore di
    Livello 1
    CAN_Invalid_Msg=0;
}
IRXIE=1; //Riattivo la possibilità di interrupt
}

if (CAN_Error_Check==1)
{
    CAN_Error_Handling();
}

```

Cominciamo l'analisi del codice a partire dalla routine di gestione degli errori in fase di trasmissione o ricezione. Quando un simile errore si verifica, il bit `IRXIF` viene settato ad 1, generando (in quanto abilitato da `IRXIE`) un interrupt. All'interno di questa routine di interrupt viene portata ad 1 la variabile di stato per l'handling di questo genere di errori:

`CAN_Invalid_Msg=1;`

Viene maggiorato un contatore (contatore degli errori di questo tipo):

`Counter_Inv_Msg++;`

Ed infine vengono resettati i due bit precedentemente citati:

`IRXIF=0; //Azzero il flag`

`IRXIE=0; //Devo disattivare temporaneamente questi interrupt per poter agire`

Tornati nel main del programma, verrà svolto il blocco di codice contenuto nel primo blocco condizionale:

`if (CAN_Invalid_Msg==1)`

{

...

}

Il codice contenuto all'interno di questi primi `if` è a sua volta ripartito in due parti;

- Se il contatore precedentemente citato non ha ancora raggiunto il valore 10, allora questo blocco porterà un LED di colore rosso a lampeggiare, per poi azzerare la variabile di stato:

`if (Counter_Inv_Msg<10)`

```

    {
        LEDRedCAN=1;
        __delay_ms(50);
        LEDRedCAN=0;
        __delay_ms(50);
        LEDRedCAN=1;
        __delay_ms(50);
        LEDRedCAN=0;
        __delay_ms(50);
        CAN_Invalid_Msg=0;
    }

```

- In alternativa, se il contatore dovesse assumere un valore di 10 o maggiore:

```

else
{
    Counter_Inv_Msg=0; //Azzero il counter
    //Devo comunicare a Raspeberry/FPGA che è stato ricevuto (o trasmesso)
    //ripetutamente un messaggio non valido
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE1; //Trasmetto il codice di "troppi messaggi non validi", Errore di Livello 1
    CAN_Invalid_Msg=0;
}

```

Ossia, viene resettato il contatore e viene impostata la trasmissione di un messaggio tramite protocollo RS-232, allo scopo di comunicare al master di sistema (FPGA o Raspeberry) che si è verificato un discreto numero di problemi (si vedano in merito le relative sezioni presenti nei *Capitolo 2* e *Capitolo 3* per maggiori informazioni circa la reazione ai messaggi di errore).

Infine, a prescindere dalla condizione verificatasi, viene riabilitata la possibilità di interrupt:

*IRXIE=1; //Riattivo la possibilità di interrupt*

Passiamo ora a trattare la seconda routine.

Come in precedenza detto la seconda fonte di errori, ha natura piuttosto variegata e in generale sono diverse le cause per cui il bit *ERRIF* potrebbe settarsi, come si noterà a breve. Ad ogni modo in termini di interrupt, questa routine è molto simile alla precedente (senza il contatore) ossia, viene settata ad 1 la corrispondente variabile di stato:

*CAN\_Error\_Check=1;*

E vengono azzerati sia il flag che l'enable di interrupt:

*ERRIF=0; //Azzero il flag*

*ERRIE=0; //Devo disattivare temporaneamente questi interrupt per poter agire*

Questo porta nel main alla chiamata alla funzione “CAN\_Error\_Handling”:

*if(CAN\_Error\_Check==1)*

{

*CAN\_Error\_Handling();*

}

Questa funzione è contenuta nel file **CAN\_Fuctions.c**:

Codice MPLAB:

*void CAN\_Error\_Handling(void)*

{

*if(RXB0OVFL==1 || RXB1OVFL==1)*

{

```

if(RXBOVLF_Counter<10)
{
    RXBOVLF_Counter++;
    RXB0OVFL=0;
    RXB1OVFL=0;
}
else
{
    RXBOVLF_Counter=0;
    //Devo comunicare a Raspberry/FPGA che si è verificato un Overflow ripetuto dei buffer di ricezione
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE1;                //Trasmetto il codice di "Segnalati troppi overflow registro di ricezione", Errore di Livello 1
    RXB0OVFL=0;
    RXB1OVFL=0;
}

if(EWARN==1)
{
    //Devo comunicare a Raspberry/FPGA che gli errori di ricezione e/o trasmissione cominciano ad essere diversi
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE2;                //Trasmetto il codice di "Segnalati diversi errori di RX/TX CAN", Errore di Livello 2
}
if(RXBP==1 || TXBP==1 || TXBO==1)
{
    //Devo comunicare a Raspberry/FPGA che gli errori di ricezione e/o trasmissione sono tanti o troppi, il master necessita di un reset
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE3;                //Trasmetto il codice di "Segnalati troppi errori di RX/TX CAN, il master deve resettarsi", Errore di livello 3
    RESET();
}

CAN_Error_Check=0; //Azzero la variabile di stato
ERRIE=1;          //Riabilito gli interrupt Errori CAN
}

```

Ora, prima di procedere con l'analisi del codice, è bene tenere a mente quanto segue circa la gestione errori CAN:

- La gestione degli errori CAN avviene attraverso il monitoraggio dei bit del registro di status delle comunicazioni CAN, denominato **COMSTAT** ([datasheet sezione 24.2.1 pagina 287/490](#)). In particolare, ognuno di questi bit è associato ad una fonte diversa di errore, tuttavia qualora uno o più di essi dovessero essere settati ad 1, il risultato comune sarebbe il trigger dell'interrupt associato ad **ERRIE**;

#### REGISTER 24-4: COMSTAT: COMMUNICATION STATUS REGISTER

Mode 0	R/C-0	R/C-0	R-0	R-0	R-0	R-0	R-0	R-0
	RXB0OVFL	RXB1OVFL	TXBO	TXBP	RXBP	TXWARN	RXWARN	EWARN
Mode 1	R/C-0	R/C-0	R-0	R-0	R-0	R-0	R-0	R-0
	—	RXBnOVFL	TXBO	TXBP	RXBP	TXWARN	RXWARN	EWARN
Mode 2	R/C-0	R/C-0	R-0	R-0	R-0	R-0	R-0	R-0
	FIFOEMPTY	RXBnOVFL	TXBO	TXBP	RXBP	TXWARN	RXWARN	EWARN
bit 7								bit 0

Fig. 4.1.13-2 Registro COMSTAT

- Come è possibile vedere leggendo il codice, non tutte le cause di errore sono state trattate e in altri casi alcune sono state accorpate:
  - Gli errori dovuti ad overflow di un registro di ricezione sono stati gestiti assieme (bit **RXB0OVFL** e **RXB1OVFL**) (e sono considerati come criticità di livello più basso rispetto alle successive);
  - Il bit **EWARN** è in realtà un bit che viene settato automaticamente qualora uno dei bit **TXWARN** o **RXWARN** si setti (nel nostro caso non è necessario fare distinzioni in quanto entrambi risultano in una criticità di Livello 2, anche se un ulteriore sviluppo potrebbe essere proprio questo);
  - Il caso in cui uno tra Receiver Bus e Transmitter Bus vada in modalità passiva è tanto grave come la condizione di Bus-off, motivo per cui tutte e 3 le condizioni sono trattate nello stesso modo.
  - Nel nostro sistema le condizioni di errore/malfunzionamento sono classificate in 3 livelli:
    - Livello 1: (Overflow registri di ricezione ed errori generici di comunicazione) Errori che inficiano localmente nel tempo le capacità operative del sistema, e che richiedono solo dopo un certo numero di accadimenti di essere trattati;
    - Livello 2: (Contatori di errore sopra il 95) Situazioni in cui i failure sono stati parecchi, quindi il sistema sta malfunzionando, con conseguente necessità di agire (ma sta ancora operando);
    - Livello 3: (Bus passivi od off) Il Sistema non è più in grado di operare correttamente, sono richiesti interventi immediati di rispristino;

Passando ora al codice, nel caso di errori dovuto ad Overflow di uno o di entrambi i registri di ricezione, il codice è molto simile a quello implementato in precedenza per gli errori di trasmissione o ricezione (a meno del LED, che qui non è presente), ossia è presente un contatore, che viene maggiorato ad ogni conferma di questo tipo di errore.

```
if(RXB0VLF_Counter<10)
{
    RXBOVLF_Counter++;
    RXB0OVFL=0;
    RXB1OVFL=0;
}
```

Se il contatore ha valore minore di 10, allora esso viene solo maggiorato e vengono resettati i flag di errore, altrimenti:

```
else
{
    RXBOVLF_Counter=0;
```

```

//Devo comunicare a Raspberry/FPGA che si è verificato un Overflow ripetuto dei buffer di
ricezione
while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
TXREG=0xE1; //Trasmetto il codice di "Segnalati troppi overflow registro di
ricezione", Errore di Livello 1
RXBOOVFL=0;
RXB1OVFL=0;
}

```

Viene azzerato il contatore, e viene effettuata la comunicazione al dispositivo master del sistema (FPGA o Raspberry) del verificarsi di un preoccupante numero di errori di overflow dei registri.

Se poi uno dei contatori di errore (Transmitter o Receiver) supera il valore di 95 (quindi setta rispettivamente **TXWARN** o **RXWARN**), allora avviene la comunicazione al master di sistema che il numero degli errori minaccia di minare seriamente le capacità di operare del sistema, ed è quindi necessario prendere provvedimenti.

```

if(EWARN==1)
{
    //Devo comunicare a Raspeberry/FPGA che gli errori di ricezione e/o trasmissione cominciano ad
    essere diversi
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE2; //Trasmetto il codice di "Segnalati diversi errori di RX/TX CAN", Errore di Livello 2
}

```

Se poi, se una tra le seguenti 3 situazioni si verifica:

- Receiver Bus in stato passivo (Receive error counter > 127);
- Transmitter Bus in stato passivo (Transmit error counter >127);
- Transmitter Bus in stato off (Transmit error counter >255);

Allora viene trasmesso al master di sistema un messaggio che identifica la condizione di impossibilità di operare del PIC Master a causa dei troppi errori CAN. A seguito di questa trasmissione il sistema tenta di ripristinarsi eseguendo un reset da riga di comando.

```

if(RXB==1 || TXBP==1 || TXBO==1)
{
    //Devo comunicare a Raspeberry/FPGA che gli errori di ricezione e/o trasmissione sono tanti o troppi,
    il master necessita di un reset
    while (TRMT!=1 && TXEN!=1); //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xE3; //Trasmetto il codice di "Segnalati troppi errori di RX/TX CAN, il master deve
    resettarsi", Errore di livello 3
    RESET();
}

```

Infine, come sempre, viene azzerata la variabile di stato della routine e vengono riabilitati di interrupt del tipo appena descritto:

```

CAN_Error_Check=0; //Azzero la variabile di stato
ERRIE=1; //Riabilito gli interrupt Errori CAN

```

Come detto all'inizio di questo paragrafo, anche i PIC Slave sono dotati di una basilare struttura di gestione degli errori CAN, tuttavia potendo essi comunicare tra loro e con il master solo tramite rete CAN, eventuali problemi risulterebbero gestibili solo fino ad un certo punto. Per questo motivo (come verrà spiegato successivamente nella sezione dedicata al prototipo degli Slave) ogni altro PIC diverso dal PIC Master possiede una ridotta struttura di gestione che si limita ad ignorare gli errori di livello 1 e 2, e procede al reset qualora dovever verificarsi una condizione di errore di livello 3.

È bene inoltre sottolineare che anche la presente strategia di gestione errori non è affatto esente da difetti/inefficienze, tuttavia, per motivi di tempo non ci si è spinti oltre nel suo sviluppo.

## 4.2 PIC SLAVE

### 4.2.1 Main Slave i-esimo

Si riporta nel seguito il contenuto del file main dello Slave, che sarà preso come prototipo per la descrizione di tutti i dispositivi Slave, in quanto tutti uguali a meno delle ovvie differenze in termini di configurazione dei filtri CAN e di contenuto dati di alcuni messaggi atti ad intraprendere azioni (esempio: Il codice-cifre da ricevere e restituire al PIC Master da parte di ogni Slave varia in base allo Slave considerato, ma logica e struttura di funzioni e routine non cambiano).

Codice MPLAB:

```
#include "Def_Library.h"
#include "Config.h"
#include "General_Settings.h"
#include "Function_Protoypes.h"

void main(void)
{
    //Eseguo il reset iniziale del dispositivo
    Initial_Reset();

    //Configuro i registri generici
    general_settings();

    //Configuro il modulo CAN
    CAN_Config_Mode();

    //Configuro il CAN per la modalità normale
    CAN_Normal_Mode();

    while(1)
    {
        if (Sensor==1)
        {
            Lock=0; //La porta può essere chiusa
            LEDGreen=0; //Spengo il LED
            //Aspetto un po' per dare il tempo alla persona di entrare e chiudere la porta
            if (Sensor_Wait==0)
            {
                TMROON=1;
            }
            if (Sensor_Wait==1) //Se il timer si è settato
            {
                Sensor_Wait=0; //Resetto lo stato della variabile
                if (PORTBbits.RB0) //Se RB0 è 1, allora la porta è ancora aperta
                {
                    Sensor=2; //Quindi devo segnalare la cosa al PIC Master
                    SystemBlock(); //Chiamo la funzione di "porta già aperta"
                }
                else //Se RB0 rimane a 0, la porta è chiusa
            }
        }
    }
}
```

```

    {
        //Resetto la password per permettere nuovi accessi
        SavedPassword[0]= -1;
        SavedPassword[1]= -1;
        SavedPassword[2]= -1;
        SavedPassword[3]= -1;

        Sensor=0; //Bene, riporto a zero la variabile
        INTOIF=0; //Provo a resettare il flag
        INTOIE=1; //Riattivo l'interrupt sensore
    }
}

if (Sensor==2)
{
    TMR1ON=1; //Attivo il Timer1
    if (Sensor_Counter==3) //Circa 6 secondi
    {
        Sensor_Counter=0; //Azzero il contatore per ricominciare il conto
        if (PORTBbits.RBO) //Da ora testo il PORTB, fino a quando non torna a zero
        {
            SystemBlock(); //Chiamo la funzione di "porta già aperta"
        }
        else //Se torna a 0
        {
            //Resetto la password per permettere nuovi accessi
            SavedPassword[0]= -1;
            SavedPassword[1]= -1;
            SavedPassword[2]= -1;
            SavedPassword[3]= -1;

            TMR1H=0; //Resetto la parte alta del Timer1
            TMR1L=0; //Resetto la parte bassa del Timer1
            TMR1ON=0; //Stoppo il Timer1

            Sensor=0; //Resetto la variabile di stato del sensore
            INTOIF=0; //Riazzeraro il flag
            INTOIE=1; //Riattivo l'interrupt del sensore
        }
    }
}

if (Msg_0==1)
{
    Slave_Receive_Routine();
}

if (PasswordReceived==1)
{

```

```

        Wait=1;
        AccessRoutine();
    }
    if(CAN_Error_Check==1)
    {
        CAN_Error_Handling();
    }
}
//Fine while
}
//Fine Main

void __interrupt() ISR(void) //Routine di interrupt

{
    di(); //Disattivo tutti gli interrupt
    CAN_ISR();

    if(PEIE==0)
    {
        PEIE=1; //Riattivo gli interrupt periferici perché all'azzeramento di GIE, viene azzerato
    }
}

}//Fine interrupt service routine

```

Come si può notare, la struttura logica del main degli Slave è assolutamente similare a quella precedentemente descritta per il PIC Master, ossia basata una “macchina a stati” governata dagli interrupt (contenuti ancora una volta nell'apposito file c **CAN\_Interrupt.c**). Ancora una volta, nel prosieguo verranno analizzate le funzioni sopra riportate, in modo anche da evidenziarne le divergenze rispetto al PIC Master. Si anticipa tuttavia che non verrà ripreso (o verrà ripreso solo parzialmente) quanto segue:

- `#include "Def_Library.h" / #include "Config.h" / #include "Function_Protoypes.h"`

Questi file hanno la medesima funzione descritta per il PIC Master, cambiano le variabili (che verranno descritte al momento del loro utilizzo), ma i bit di configurazione sono i medesimi.

- `Initial_Reset();`

In quanto essa è identica a quella descritta in precedenza a meno dell'elenco delle variabili.

- `general_settings();`

Verranno unicamente evidenziati i settaggi diversi da quelli del PIC Master (perché molto simili).

- `CAN_Config_Mode();`

Non verrà ridiscussa in quanto l'unico cambiamento riguarda il settaggio del filtro che identifica il dispositivo, ossia:

`RXF0SIDL=0b00100000;`

- `CAN_Normal_Mode();`

Le funzioni di configurazione/richiesta delle modalità di funzionamento del modulo CAN sono assolutamente identiche a quelle descritte per il PIC Master.

Inoltre, tra i PIC Slave ne è presente uno che per il suo ruolo all'interno dell'architettura del sistema presenta lievi differenze rispetto al prototipo dello Slave (perlopiù si tratta di ulteriori semplificazioni), di quel particolare dispositivo si parlerà brevemente al termine del capitolo.

#### 4.2.2 General Settings (Slave)

Come già accennato altrove (nello specifico nella sezione 4.1.4), i settaggi e le configurazioni dei registri relativi ai diversi PIC Slave presentano moltissime analogie con la routine di settaggio del PIC Master, ma anche sostanziali differenze, dovute alle routine specifiche del PIC Slave che quest'ultimo deve eseguire. Occorre precisare che tali impostazioni non differiscono tra un PIC Slave e l'altro, in quanto questi dispositivi devono svolgere più o meno le medesime mansioni (le quali verranno trattate nelle prossime sezioni). L'unica eccezione è data dal PIC Slave 5, che presenta alcune piccole differenze rispetto ai settaggi degli altri Slave, in quanto PIC responsabile della porta principale, la cui gestione è diversa dalle altre porte. Laddove non specificato, i settaggi sono gli stessi per tutti i PIC Slave.

Si riporta in seguito il codice integrale della routine denominata “general\_settings”, la quale è invocata nel main del PIC Slave non appena questo ha effettuato il reset iniziale:

//Configuro i registri generici

```
general_settings();
```

Tale routine è contenuta integralmente nel file header **General\_Settings.h**.

Codice MPLAB (*tra parentesi, in grassetto, si riporta la variante di codice corrispondente relativa al PIC Slave 5*):

```
void general_settings(void) //Funzione dei settaggi
{
    //Inizializzazione PORT

    //PORTB
    TRISB=0b00001001; //Setto tutto le porte, in particolare RB2 dovrà essere
                      //usata come CANTX (viene settata in automatico quando
                      //si inizializzano i registri CAN, mentre RB3 va manualmente
                      //settato come input (1) in modo da essere il CANRX)
    LATB=0x00;

    //PORTA
    LATA=0x00;

    //PORTC
    TRISC=0x00; //Tutte le porte C sono output
    LATC=0x00;

    //PORTD
    TRISD=0b11111000;
    LATD=0x00;

    //Settaggi aggiuntivi
    //I PIN delle porte A sono analogiche di default, il nostro sistema gestisce segnali digitali
    ADCON1=0b00001111; //Setto tutti i pin analogici come digitali
    ADCON0bits.AD0=0; //Spengo il convertitore A/D

    //Inizializzazione Registro di clock interno del sistema (datasheet 36/490)
    //Frequenza di lavoro a 1Mhz
    OSCCON=0b01000110;
```

```

//Disattivo il modulo CCP (datasheet 167/490)
CCP1CON=0b00000000; //Usa solo gli ultimi 4 bit

//Disabilito la priorità degli interrupt
RCONbits.IPEN=0;

//Configuro il Timer0 (datasheet slide 151/490)
T0CON=0b00000110;
//Overflow in circa 33.5 secondi

//Configuro il Timer1 (datasheet slide 155/490)
T1CON=0b10110000;
//Overflow in circa 2 secondi

//Configuro il Timer3 (datasheet slide 163/490) (riga di codice assente)
T3CON=0b10110000; (riga di codice assente)
//Overflow in circa 2 secondi (riga di codice assente)

//Setto i registri di ricezione per i messaggi CAN
RXB1CON=0b00100000;
RXB0CON=0b00100000;

//Impostazione/abilitazione interrupt
INTCON=0b11110000; //Abilito anche il timer0 (bit 5)

//Registro di abilitazione interrupt periferici 1
PIE1=0b00000001; // (datasheet slide 127/490)
//bit 0: Enable interrupt di overflow timer1

//Registro di abilitazione interrupt periferici 2
PIE2=0b00000010; // (datasheet slide 128/490) (PIE2=0b00000000; // (datasheet slide 128/490))
//bit 1: Enable interrupt di overflow timer3

//Registro di abilitazione interrupt periferici (CAN)
PIE3=0b00100001; //(datasheet slide 129/490 o 322/490)
//I bit 7,6,5 settati a 0 disattivano i 3 interrupt CAN di (IRXIE, WAKIE, ERRIE)
//I bit 1 e 0 gestiscono gli interrupt di ricezione rispettivamente sui Buffer CAN 1 e 0
}

```

Come si evince dal confronto del codice sopra riportato con la controparte relativa al PIC Master (riportata nella sezione 4.1.4), vi sono alcune differenze degne di nota, che vengono qui riportate in ordine di apparizione all'interno del programma. Se non diversamente specificato, quanto riportato nei seguenti punti vale anche per il PIC Slave 5.

- Il settaggio del registro **TRISB** presenta un pin di ingresso in più (il pin **RBO**):

**TRISB**=0b00001001;

Questo pin, infatti, è connesso direttamente con il sensore della porta corrispondente, il quale viene utilizzato come fonte di interrupt esterna (*INTO*) qualora il sensore restituisca un valore logico alto, ossia qualora la porta risulti aperta (vedere il *punto 7* per maggiori dettagli).

2. Il settaggio del registro *TRISA* risulta futile, dato che i pin ad esso associati non sono utilizzati.
3. Il settaggio del registro *TRISD* presenta una variazione nella configurazione dei pin di input/output:

*TRISD=0b11111000;*

In particolare, i pin da *RD3* a *RD7* sono settati come ingressi. Questi pin sono connessi al tastierino specifico della porta, tre di questi pin identificano tre specifiche colonne del tastierino stesso, gli altri non sono utilizzati.

4. Il settaggio del Timer0 (registro *TOCON*) differisce nel valore del prescaler impostato, pari a 1:128:

*TOCON=0b00000110;*

Questo comporta un dimezzamento del tempo impiegato dal Timer0 per raggiungere l'overflow, rispetto al Timer0 del PIC Master. Inoltre, il Timer risulta spento al settaggio, perché nei PIC Slave il Timer0 viene impiegato per una routine diversa, relativa al tempo limite per la chiusura della porta in condizioni nominali, e deve quindi cominciare a conteggiare non appena il software lo richiede (per maggiori dettagli si rimanda alle *sezioni 4.2.4 e 4.2.5*).

5. Solo per il PIC Slave 5, la configurazione del Timer3 è assente. Questo perché tale timer non viene usato, mentre negli altri Slave serve per settare il tempo limite per l'inserimento della password (la descrizione accurata delle routine coinvolte è rimandata alle *sezioni 4.2.3 e 4.2.4*).
6. Manca del tutto il settaggio dei registri relativi al modulo EUSART (registri *RCSTA*, *TXSTA*, *BAUDCON*, *SPBRGH* e *SPBRG*), in quanto tale rete di comunicazione sussiste soltanto tra PIC Master e Raspberry/scheda FPGA.
7. Durante il settaggio del registro *INTCON*, viene abilitata anche la generazione di interrupt proveniente da uno stimolo esterno (bit *INTOE*).

*INTCON=0b11110000;*

Questo bit così settato consente di generare un interrupt in presenza di un ingresso alto sul pin denominato *INTO*, al quale è connesso il sensore della porta corrispondente. In altre parole, viene generato un interrupt in presenza di una porta aperta (la cui routine è descritta nella *sezione 4.2.5*).

8. Per la stessa ragione enunciata nel *punto 6*, nel registro *PIE1* il bit di enable della ricezione dati sul modulo EUSART (bit *RCIE*) è settato a 0.

*PIE1=0b00000001;*

9. Come conseguenza del *punto 5*, solo nei settaggi del PIC Slave 5, sul registro *PIE2* non viene abilitato l'interrupt generato dall'overflow del Timer3 (*TMR3IE*).

*PIE2=0b00000000;*

10. Nel registro *PIE3*, il bit *IRXIE* (enabler di interrupt sulla trasmissione/ricezione di messaggi CAN non validi) è settato a 0.

*PIE3=0b00100001;*

Questo perché ogni PIC Slave è in grado di comunicare con il resto del sistema soltanto tramite la rete CAN, e quindi risulterebbe impossibile segnalare questa condizione d'errore al PIC Master o a qualunque altro dispositivo attraverso altre vie. La gestione degli errori CAN verrà trattata nel dettaglio più avanti (per maggiori dettagli, si rimanda alla *sezione 4.2.6*).

#### 4.2.3 Routine di ricezione CAN (Slave)

La routine di ricezione messaggi implementata sui diversi PIC Slave del sistema risulta leggermente differente rispetto alla routine precedentemente descritta per il caso del PIC Master.

Quando un nuovo messaggio viene ricevuto (e supera il filtraggio), un interrupt di ricezione viene lanciato e la seguente porzione di codice viene eseguita:

Codice MPLAB:

```
if (RXBOIF)
{
    //Cosa faccio ora che il buffer0 ha ricevuto un messaggio?

    if (RXBODO==10 && RXBOD1==20 && RXBOD2==30 && RXBOD3==40 && RXBOD4==50)
    {
        Test=1;
        Msg_0=1;
    }

    else if (RXBODO==11 && RXBOD1==11 && RXBOD2==11 && RXBOD3==11 && RXBOD4==11)
    {
        Alive_Ans=1;
        Msg_0=1;
    }

    else //Se il messaggio non è uno di quelli codificati, allora è una password
    {
        //Se la precedente password è quella di default posso salvarne una nuova e permettere l'accesso
        If (SavedPassword[0]==-1 && SavedPassword[1]==-1 && SavedPassword[2]==-1 && SavedPassword[3]==-1 && TMROON==0 && TMR1ON==0 && Sensor==0)
        {
            SavedPassword[0]=RXBODO;
            SavedPassword[1]=RXBOD1;
            SavedPassword[2]=RXBOD2;
            SavedPassword[3]=RXBOD3;
            PasswordReceived=1;
        }

        else //Se non lo è, significa che qualcuno sta accedendo all'area e dunque devo attendere che la password si resetti
        {
            AccessDenied=1; //Informo il PIC della porta principale della cosa
            Msg_0=1;
        }
    }

    //Azzero il Buffer di ricezione
    RXBODO=0;
    RXBOD1=0;
    RXBOD2=0;
    RXBOD3=0;
    RXBOD4=0;
```

```

RXBOD5=0;
RXBOD6=0;
RXBOD7=0;

RXBOFUL=0; //Azzero il flag, in modo da segnalare la possibilità di ricevere un nuovo messaggio
RXBOIF=0; //Azzero il flag
}

```

Come si può notare, quello che fa la routine sopra riportata è leggere il contenuto dei registri del buffer di ricezione ed in base ai valori letti agire di conseguenza. Per evitare di trattenere il programma troppo a lungo all'interno di un interrupt, la routine di ricezione è stata pensata per eseguire a questo punto solo le minime azioni necessarie e a rimandare al main le azioni più dispendiose in termini di tempo. Nel PIC Master, l'assenza di questo genere di eventi ha reso la routine più semplice e compatta.

Come si può notare le casistiche possibili sono fondamentalmente 3:

1. *if (RXBOD0==10 && RXBOD1==20 && RXBOD2==30 && RXBOD3==40 && RXBOD4==50)*

Questa prima eventualità riguarda la funzione di “Initial\_Test()” introdotta in precedenza per il PIC Master e ha come immediata conseguenza il settaggio di due variabili di stato *Test* ed *Msg\_0* che serviranno ad eseguire la porzione corretta di codice nel main (che verrà trattata più avanti in questa sezione);

```

Test=1;
Msg_0=1;

```

2. *else if (RXBOD0==11 && RXBOD1==11 && RXBOD2==11 && RXBOD3==11 && RXBOD4==11)*

La seconda eventualità riguarda la richiesta di conferma di operatività ricevuta dal PIC Master, per la quale ancora una volta è stata prevista una funzione a parte, infatti come per il caso precedente qui il programma si limita a settare due variabili di stato, *Alive\_Ans* e *Msg\_0*;

```

Alive_Ans=1;
Msg_0=1;

```

3. *else //Se il messaggio non è uno di quelli codificati, allora è una password*

L'ultimo caso è relativo alla ricezione di una password d'accesso in conseguenza di una richiesta di accesso da parte di un utente. A seguito di questa ricezione, i possibili casi sono due:

- *If (SavedPassword[0]==-1 && SavedPassword[1]==-1 && SavedPassword[2]==-1 && SavedPassword[3]==-1 && TMROON==0 && TMR1ON==0 && Sensor==0)*

Questo è il solo caso in cui (come verrà spiegato meglio nel seguito) viene consentito l'accesso ad un nuovo utente, infatti, come si evince dal codice, all'interno dell'*if* viene salvata la password d'accesso in un apposito array che verrà poi utilizzato per la routine di “convalida” della password:

```

SavedPassword[0]= RXBOD0;
SavedPassword[1]= RXBOD1;
SavedPassword[2]= RXBOD2;
SavedPassword[3]= RXBOD3;

```

Dopodichè viene settata ad 1 la variabile di stato associata alla prima fase della routine d'accesso vera e propria:

```
>PasswordReceived=1;
```

Questo caso si verifica quando l'array contenente le cifre della password è al suo valore di default (ossia -1,-1,-1,-1) e, nessuna delle variabili che identificano la routine di gestione/check dello stato di chiusura della porta è attiva (ossia la richiesta non è troppo a ridosso di un'apertura passata).

- Altrimenti (*else*), se la password non è al suo valore di default (quindi è in corso un accesso o la verifica di chiusura corretta della porta, si veda il seguito per maggiori dettagli) vengono settate le necessarie variabili per l'esecuzione di un blocco di codice dedicato (che verrà analizzato poco oltre, al termine del paragrafo)

```
AccessDenied=1; //Informo il PIC della porta principale della cosa  
Msg_0=1;
```

Infine:

```
//Azzero il Buffer di ricezione  
RXB0D0=0;  
RXB0D1=0;  
RXB0D2=0;  
RXB0D3=0;  
RXB0D4=0;  
RXB0D5=0;  
RXB0D6=0;  
RXB0D7=0;
```

```
RXB0FUL=0; //Azzero il flag, in modo da segnalare la possibilità di ricevere un nuovo messaggio  
RXBOIF=0; //Azzero il flag
```

Queste ultime righe si occupano di:

- Azzerare i registri di ricezione;
- Resetare il bit *RXB0FUL* (così abilito la possibilità di ricevere un nuovo messaggio su buffer 0);
- Resetare il flag di interrupt;

Riprendendo ora il discorso riguardante quelle situazioni la cui gestione sarebbe potuta risultare dispendiosa in termini di tempo (ovvero i casi 1.2 e 3.2 sopra citati). Come visto in precedenza il verificarsi di una delle tre situazioni porta al settaggio di alcune variabili di stato, una di esse, *Msg\_0*, viene sempre settata e porta nel main all'esecuzione del seguente blocco di codice:

```
if (Msg_0==1)  
{  
    Slave_Receive_Routine();  
}
```

Il quale ha lo scopo di effettuare una chiamata alla funzione “Slave\_Receive\_Routine()”, il cui corpo è contenuto nel file **CAN\_Functions.c**.

Codice MPLAB:

```
void Slave_Receive_Routine(void)  
{  
    if (Test==1)
```

```

{
    //Ho ricevuto il messaggio di test iniziale....
    LEDRed=1; //Accendo i LED per 5 secondi
    LEDGreen=1;
    __delay_ms(5000);
    LEDRed=0;
    LEDGreen=0;
    Test=0;
}
else if (Alive_Ans==1)
{
    IamAlive(); //Rispondo al Master
    Alive_Ans=0;
}
else if (AccessDenied==1)
{
    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRIO=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXBOSIDH=0b00000000;
    TXBOSIDL=0b10100000; //Scrivo allo slave 5

    //Setto la lunghezza del messaggio (5 byte in questo caso)
    TXB0DLC=0b00000001;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=29; //Scrivo nel registro 0 del buffer 0

    //Setto il bit di trasmissione
    TXREQ0=1;

    AccessDenied=0;
}

Msg_0=0;
}

```

Questa funzione smista e gestisce tutti quegli eventi di ricezione che non sono stati precedentemente gestiti in modo immediato all'interno dell'interrupt, ossia:

*if (Test==1)*

Se è stato ricevuto il comando per il test iniziale, allora il PIC dovrà accendere 2 LED per la durata di 5 secondi, dopodiché azzero la variabile di stato.

Altrimenti:

*else if (Alive\_Ans==1)*

Se è stata ricevuta una richiesta di conferma di operatività da parte del PIC Master, verrà chiamata la funzione "IamAlive()", definita nel file **CAN\_Functions.c** e riportata nel seguito.

Codice MPLAB:

```
void IamAlive(void)
{
    //Devo far sapere al master che sono ancora vivo

    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRIO=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXBOSIDH=0b00000000;
    TXBOSIDL=0b11000000; //Scrivo al Master (Slave Tx -> Master RX)

    //Setto la lunghezza del messaggio (5 byte in questo caso)
    TXBODLC=0b00000101;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=11; //Scrivo nel registro 0 del buffer 0
    TXB0D1=11; //Scrivo nel registro 1 del buffer 0
    TXB0D2=11; //Scrivo nel registro 2 del buffer 0
    TXB0D3=11; //Scrivo nel registro 3 del buffer 0
    TXB0D4=11; //Scrivo nel registro 4 del buffer 0

    //Setto il bit di trasmissione
    TXREQ0=1;

    //Lampeggio di segnalazione
    LEDGreen=1;
    __delay_ms(50);
    LEDGreen=0;
}
```

Questa funzione ha il solo scopo di inviare un messaggio al PIC Master via CAN contenente la conferma di operatività (serie di 5 numeri uguali tra loro, diversa per ogni dispositivo Slave del sistema), dopodiché come per il precedente blocco, viene resettata la variabile di stato ad esso associato e viene segnalato all'utente il corretto invio della risposta tramite un lampeggio di un LED verde.

Altrimenti ancora:

```
else if (AccessDenied==1)
```

Se un accesso è stato negato devo informarne il PIC gestore della porta di accesso in modo tale che esso possa segnalarlo all'utente in attesa di conferma. Questo avviene con le stesse modalità esposte poc'anzi per la routine di Alive CAN, con la differenza che questo messaggio consiste in un solo carattere (29) inviato appunto non al PIC Master ma al PIC della porta principale (si anticipa che questo messaggio causerà semplicemente il lampeggio di un LED rosso sul dispositivo in risposta ad una sua corretta ricezione).

Infine, indipendentemente da quale dei due precedenti casi si sia verificato, viene resettata la variabile di stato della funzione "Slave\_Receive\_Routine" (Msg\_0).

#### 4.2.4 Routine di accesso

La funzione “AccessRoutine”, come suggerito dal nome, gestisce la routine d’accesso ad ogni porta del corridoio. In particolare, si tratta di una struttura a macchina a stati composta da sei stati diversi identificati da variabili globali:

1. *Wait*;
2. *KeyPad*;
3. *PassCheck*;
4. *OpenDoor*;
5. *PassClean*;
6. *DoorBlocked*.

Tutti i suddetti stati sono contenuti in un ciclo *while*(1), e vengono eseguiti soltanto quando la loro variabile identificativa ha valore uguale ad uno. Di seguito il codice completo della funzione “AccessRoutine”:

```
void AccessRoutine(void) {
    TMR3ON=1; //Attivo il Timer3

    //Scrivo al PIC della porta principale di sbloccare la serratura
    //Imposto la priorità dei messaggi
    TXB0CONbits.TXPRI0=1;
    TXB0CONbits.TXPRI1=1;

    //Imposto l'identificativo messaggio
    TXBOSIDH=0b00000000;
    TXBOSIDL=0b10100000; //Scrivo allo slave 5

    //Setto la lunghezza del messaggio (1 byte in questo caso)
    TXB0DLC=0b00000001;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=19; //Scrivo nel registro 0 del buffer 0

    //Setto il bit di trasmissione
    TXREQ0=1;

    PasswordReceived=0;

    while (1) {
        if (Alive_Ans==1)
        {
            IamAlive(); //Rispondo al Master
            Alive_Ans=0;
        }
        else if (AccessDenied==1)
        {
            //Imposto la priorità dei messaggi
            TXB0CONbits.TXPRI0=1;
            TXB0CONbits.TXPRI1=1;
```

```

//Imposto l'identificativo messaggio
TXBOSIDH=0b00000000;
TXBOSIDL=0b10100000; //Scrivo allo slave 5

//Setto la lunghezza del messaggio (1 byte in questo caso)
TXBODLC=0b00000001;

//Imposto il contenuto dei registri da inviare
//TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
TXBODO=29; //Scrivo nel registro 0 del buffer 0

//Setto il bit di trasmissione
TXREQ0=1;

AccessDenied=0;
}
Msg_0=0;
}
if(Counter<30)
{
if(Wait==1) //Attendi che l'utente sia pronto a digitare la password
{
ROWA=0;
ROWB=0;
ROWC=0;
ROWD=1;
LEDGreen=1;

if(COL3)
{
Wait=0;
KeyPad=1; //Passa alla funzione tastierino
__delay_ms(delayButton);
LEDGreen=0;
TMR3H=0; //Resetto la parte alta del Timer3
TMR3L=0; //Resetto la parte bassa del Timer3
TMR3ON=0; //Disattivo il Timer3

Counter=0; //Azzero il contatore
}
}
else
{
TMR3H=0; //Resetto la parte alta del Timer3
TMR3L=0; //Resetto la parte bassa del Timer3
TMR3ON=0; //Disattivo il Timer3
Counter=0; //Azzero il contatore
}
}
}
}

```

```

//Resetto la password per permettere nuovi accessi
SavedPassword[0]= -1;
SavedPassword[1]= -1;
SavedPassword[2]= -1;
SavedPassword[3]= -1;

return;
}

if(KeyPad==1)                                //Quando è stata completata la ricezione della password, un
{
    ROWA=1;                                     //Interrupt setta KeyPad=1
    ROWB=0;                                     //Attivo la prima riga
    ROWC=0;
    ROWD=0;

if(COL1 && ArrayCount<4)      {
    DigitPassword[ArrayCount] = 1;           //NUMERO 1
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL2 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 2;           //NUMERO 2
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL3 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 3;           //NUMERO 3
    __delay_ms(delayButton);
    ArrayCount++;
}

ROWA=0;
ROWB=1;                                       //Attivo la seconda riga
ROWC=0;
ROWD=0;

if(COL1&& ArrayCount<4)
{
    DigitPassword[ArrayCount] = 4;           //NUMERO 4
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL2&& ArrayCount<4)
{
    DigitPassword[ArrayCount] = 5;           //NUMERO 5
}

```

```

    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL3 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 6;           //NUMERO 6
    __delay_ms(delayButton);
    ArrayCount++;
}

ROWA=0;
ROWB=0;
ROWC=1;                                //Attivo la terza riga
ROWD=0;

if(COL1 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 7;           //NUMERO 7
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL2 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 8;           //NUMERO 8
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL3 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 9;           //NUMERO 9
    __delay_ms(delayButton);
    ArrayCount++;
}

ROWA=0;
ROWB=0;
ROWC=0;
ROWD=1;                                //Attivo la quarta riga
if(COL1 && ArrayCount>0)                //Cancella ultimo carattere inserito
{
    ArrayCount--;
    DigitPassword[ArrayCount] = 0;
    __delay_ms(delayButton);
}
if(COL2 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 0;           //NUMERO 0
    __delay_ms(delayButton);
}

```

```

        ArrayCount++;
    }
if(COL3) //Verifica Password inserita
{
    KeyPad=0;
    PassCheck=1; //Attiva lo stato di verifica della Password
    __delay_ms(delayButton);
}
}

if(PassCheck==1) //Verifica coincidenza della password
{
    PassCheck=0;
    if (DigitPassword[0]==SavedPassword[0])
    { if (DigitPassword[1]==SavedPassword[1])
        { if (DigitPassword[2]==SavedPassword[2])
            { if (DigitPassword[3]==SavedPassword[3])
                {
                    LEDGreen=1;
                    __delay_ms(delayLed);
                    LEDGreen=0;
                    OpenDoor=1; //Passa alla funzione OpenDoor
                }
                else //Se l'ultimo carattere non è giusto
                {
                    LEDRed=1;
                    __delay_ms(delayLed);
                    LEDRed=0;
                    PassClean= 1; //Passa allo stato PassClean
                    TryCount++;
                }
            }
            else //Se il terzo carattere non è giusto
            {
                LEDRed=1;
                __delay_ms(delayLed);
                LEDRed=0;
                PassClean= 1; //Passa allo stato PassClean
                TryCount++;
            }
        }
        else //Se il secondo carattere non è giusto
        {
            LEDRed=1;
            __delay_ms(delayLed);
            LEDRed=0;
            PassClean= 1; //Passa allo stato PassClean
            TryCount++;
        }
    }
}

```

```

        }
    }
    else           //Se il terzo carattere non è giusto
    {
        LEDRed=1;
        __delay_ms(delayLed);
        LEDRed=0;
        PassClean= 1;      //Passa allo stato PassClean
        TryCount++;
    }
}

if (OpenDoor==1)
{
    //Disattivo tutte le righe
    ROWA=0;
    ROWB=0;
    ROWC=0;
    ROWD=0;

    DigitPassword[0]=-1;
    DigitPassword[1]=-1;
    DigitPassword[2]=-1;
    DigitPassword[3]=-1;

    OpenDoor=0;
    ArrayCount=0;
    Lock=1;           //Abilito apertura porta
    return;
}

if (PassClean==1)
{
    PassClean = 0;
    ArrayCount=0;

    //Disattivo tutte le righe
    ROWA=0;
    ROWB=0;
    ROWC=0;
    ROWD=0;

    DigitPassword[0]=0;
    DigitPassword[1]=0;
    DigitPassword[2]=0;
    DigitPassword[3]=0;

    if (TryCount==3)      //se i tentativi sbagliati sono 3
    {
        TryCount=0;
    }
}

```

```

DoorBlocked=1; //passa allo stato porta bloccata
}
else
{
    Wait=1; //torna allo stato tastierino
}
}

if(DoorBlocked==1) //stato porta bloccata
{
    DoorBlocked=0;

    //Disattivo tutte le righe
    ROWA=0;
    ROWB=0;
    ROWC=0;
    ROWD=0;

    LEDRed=1; //led rosso lampeggiante
    __delay_ms(100);
    LEDRed=0;
    __delay_ms(100);
    LEDRed=1;
    __delay_ms(100);
    LEDRed=0;
    __delay_ms(100);
    LEDRed=1;
    __delay_ms(100);
    LEDRed=0;

    //Resetto la password per permettere nuovi accessi
    SavedPassword[0]=-1;
    SavedPassword[1]=-1;
    SavedPassword[2]=-1;
    SavedPassword[3]=-1;

    return;
    //Troppi errori fatti, bisogna ricominciare
}

}
}

//fine della funzione AccessRoutine()

```

Quando la funzione “AccessRoutine” viene chiamata, per prima cosa viene attivato il Timer3. Subito dopo il generico PIC Slave invia il messaggio 19 allo slave 5 (porta principale), comunicandogli di sbloccare la serratura per lasciar entrare l’utente. A questo punto la variabile *PasswordReceived* può essere resettata. Il codice necessario a questa operazione è riportato e commentato sotto:

```

void AccessRoutine(void) {
    TMR3ON = 1; //Attivo il Timer3
}

```

```

//Scrivo al PIC della porta principale di sbloccare la serratura
//Imposto la priorità dei messaggi
TXB0CONbits.TXPRI0=1;
TXB0CONbits.TXPRI1=1;

//Imposto l'identificativo messaggio
TXBOSIDH=0b00000000;
TXBOSIDL=0b10100000; //Scrivo allo slave 5

//Setto la lunghezza del messaggio (5 byte in questo caso)
TXB0DLC=0b00000101;

//Imposto il contenuto dei registri da inviare
//TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
TXB0DO=19; //Scrivo nel registro 0 del buffer 0

//Setto il bit di trasmissione
TXREQ0=1;

PasswordReceived=0;

```

A questo punto viene aperto il ciclo `while(1)` che ad ogni ciclo verifica se sia arrivato un messaggio di tipo `Msg_0` e discrimina tra il caso in cui il messaggio ricevuto sia una richiesta di alive e quello in cui sia stato negato un accesso al corridoio principale:

```

while (1) {
    if (Msg_0 == 1)
    {
        if (Alive_Ans==1)
        {
            IamAlive(); //Rispondo al Master
            Alive_Ans==0
        }
        else if (AccessDenied==1)
        {
            //Imposto la priorità dei messaggi
            TXB0CONbits.TXPRI0=1;
            TXB0CONbits.TXPRI1=1;

            //Imposto l'identificativo messaggio
            TXBOSIDH=0b00000000;
            TXBOSIDL=0b10100000; //Scrivo allo slave 5

            //Setto la lunghezza del messaggio (1 byte in questo caso)
            TXB0DLC=0b00000001;

            //Imposto il contenuto dei registri da inviare

```

```

//TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
TXB0D0=29; //Scrivo nel registro 0 del buffer 0

//Setto il bit di trasmissione
TXREQ0=1;

AccessDenied=0;
}
//Altre azioni
Msg_0 = 0;
}

```

Dopodiché viene controllata la variabile *Counter* (implementata nella sezione main dello slave): se questo non ha raggiunto il limite massimo di attesa configurato (un minuto), viene controllata la variabile *wait*; quest'ultima è sempre uguale ad uno appena viene eseguita la funzione, per cui verrà attivata l'ultima riga del tastierino ed un LED verde, in modo che l'utente possa facilmente identificare la porta selezionata in precedenza (nel caso mancassero delle chiare indicazioni) e avviare la procedura di verifica della password premendo il tasto "#" (Enter). A questo punto il LED verrà spento, la variabile *Wait* ed il contatore saranno azzerate, il Timer3 resettato e disattivato, la password ricevuta resettata e la variabile *KeyPad* verrà settata ad uno, permettendo l'esecuzione dello stato successivo.

```

if(Counter<30)
{
    if(Wait==1)                                //Attendi che l'utente sia pronto a digitare la password
    {
        ROWA=0;
        ROWB=0;
        ROWC=0;
        ROWD=1;
        LEDGreen=1;

        if(COL3)
        {
            Wait=0;
            KeyPad=1;                          //Passa alla funzione tastierino
            __delay_ms(delayButton);
            LEDGreen=0;
            TMR3H=0;      //Resetto la parte alta del Timer3
            TMR3L=0;      //Resetto la parte bassa del Timer3
            TMR3ON=0;     //Disattivo il Timer3

            Counter=0;    //Azzero il contatore
        }
    }
    else
    {
        TMR3H=0;      //Resetto la parte alta del Timer3
        TMR3L=0;      //Resetto la parte bassa del Timer3
        TMR3ON=0;     //Disattivo il Timer3
    }
}

```

```

Counter=0; //Azzero il contatore
//Resetto la password per permettere nuovi accessi
SavedPassword[0]= -1;
SavedPassword[1]= -1;
SavedPassword[2]= -1;
SavedPassword[3]= -1;
return;
}

```

Di seguito (Fig. 4.2.4-1) una descrizione schematica del codice appena descritto:

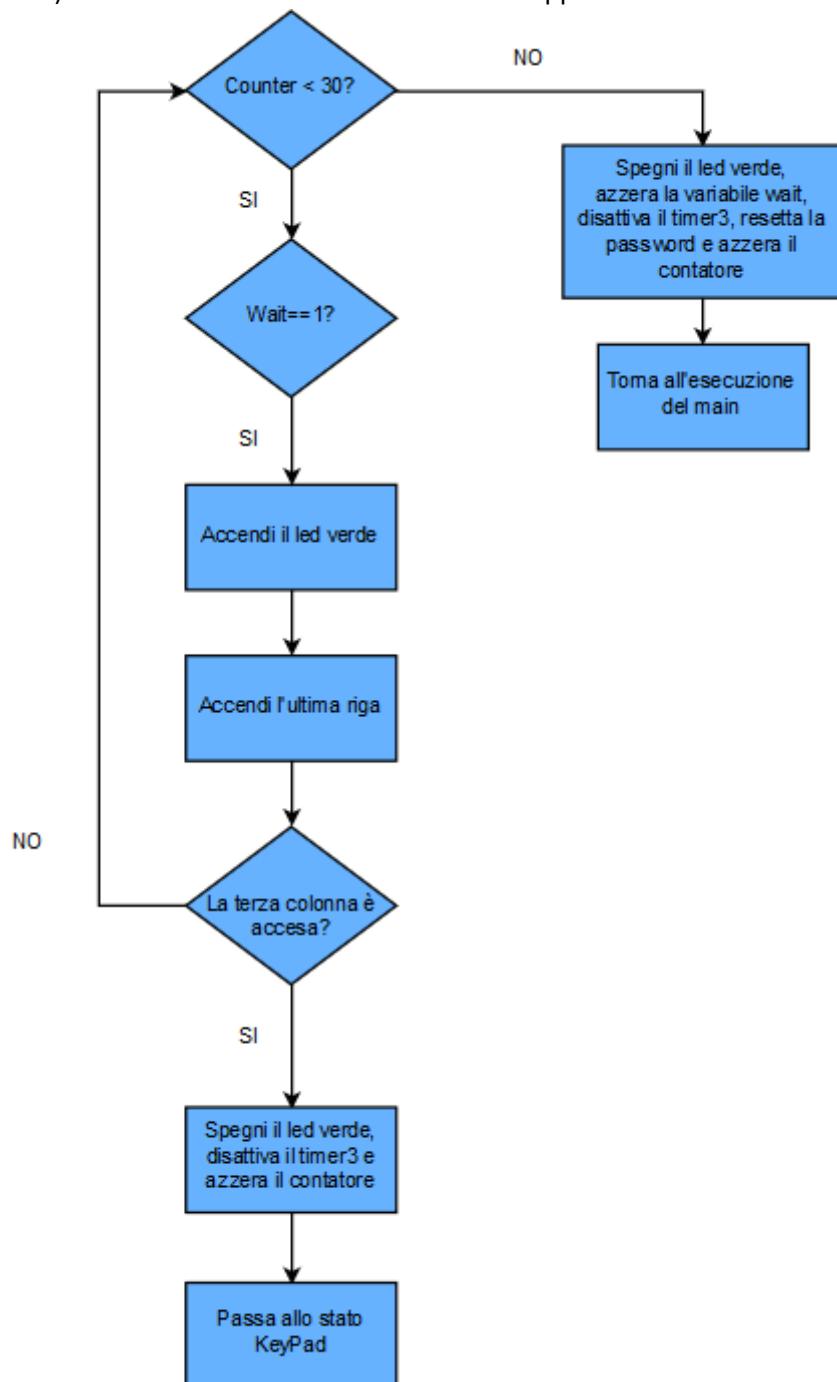


Fig. 4.2.4-1 Pseudocodice dello stato di attesa

Durante lo stato definito da *KeyPad* si esegue una routine tipica dei tastierini a matrice: viene attivata una riga alla volta (con una velocità tale da non essere percepita dall'utente) e, mentre ogni riga è attiva, viene eseguita la verifica di tutte le colonne; se una di queste è attiva (cioè l'utente ha fatto pressione su un tasto appartenente a quella colonna) viene memorizzato in un array il carattere corrispondente a quello premuto (e.g. se non è ancora stato digitato nessun tasto e l'utente preme sul carattere 1, la posizione 0 dell'array *DigitPassword* verrà occupata dal carattere "1"; se sono già stati digitati 2 caratteri e l'utente preme sul carattere 9, la posizione 2 dell'array *DigitPassword* sarà occupata dal carattere "9"; etc.). È stato quindi implementato un contatore *ArrayCount*, che è incrementato ogni volta che viene fatta pressione su di un tasto.

La pressione di un tasto numerico è abilitata soltanto se il contatore *ArrayCount* è inferiore a 4, cioè sono state inserite meno di 4 cifre, onde evitare sovrascritture dei caratteri digitati in altre variabili.

Se l'utente sbaglia a digitare un carattere, può fare pressione sul tasto "\*" in basso a sinistra (Cancella): questo permette di decrementare il contatore *ArrayCount* e di eliminare dall'array *DigitPassword* l'ultimo carattere digitato.

Quando l'utente è certo di aver inserito la password esatta dovrà cliccare sul tasto "#" in basso a destra (Enter): questo azzera la variabile *KeyPad* e setta ad uno la variabile *PassCheck*.

Di seguito la descrizione in codice dello stato *KeyPad* ed in Fig. 4.2.4-2 l'algoritmo schematizzato:

```

if(KeyPad==1)                                //Quando è stata completata la ricezione della password, un
{
    ROWA=1;                                  //Interrupt setta KeyPad=1
    ROWB=0;                                  //Attivo la prima riga
    ROWC=0;
    ROWD=0;

    if(COL1 && ArrayCount<4)    {
        DigitPassword[ArrayCount] = 1;      //NUMERO 1
        __delay_ms(delayButton);
        ArrayCount++;
    }
    if(COL2 && ArrayCount<4)
    {
        DigitPassword[ArrayCount] = 2;      //NUMERO 2
        __delay_ms(delayButton);
        ArrayCount++;
    }
    if(COL3 && ArrayCount<4)
    {
        DigitPassword[ArrayCount] = 3;      //NUMERO 3
        __delay_ms(delayButton);
        ArrayCount++;
    }

    ROWA=0;                                 //Attivo la seconda riga
    ROWB=1;
    ROWC=0;
    ROWD=0;

    if(COL1&& ArrayCount<4)

```

```

{
    DigitPassword[ArrayCount] = 4;           //NUMERO 4
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL2&& ArrayCount<4)
{
    DigitPassword[ArrayCount] = 5;           //NUMERO 5
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL3 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 6;           //NUMERO 6
    __delay_ms(delayButton);
    ArrayCount++;
}

ROWA=0;
ROWB=0;
ROWC=1;                                //Attivo la terza riga
ROWD=0;

if(COL1 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 7;           //NUMERO 7
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL2 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 8;           //NUMERO 8
    __delay_ms(delayButton);
    ArrayCount++;
}
if(COL3 && ArrayCount<4)
{
    DigitPassword[ArrayCount] = 9;           //NUMERO 9
    __delay_ms(delayButton);
    ArrayCount++;
}

ROWA=0;
ROWB=0;
ROWC=0;
ROWD=1;                                //Attivo la quarta riga
if(COL1 && ArrayCount>0)                //Cancella ultimo carattere inserito
{

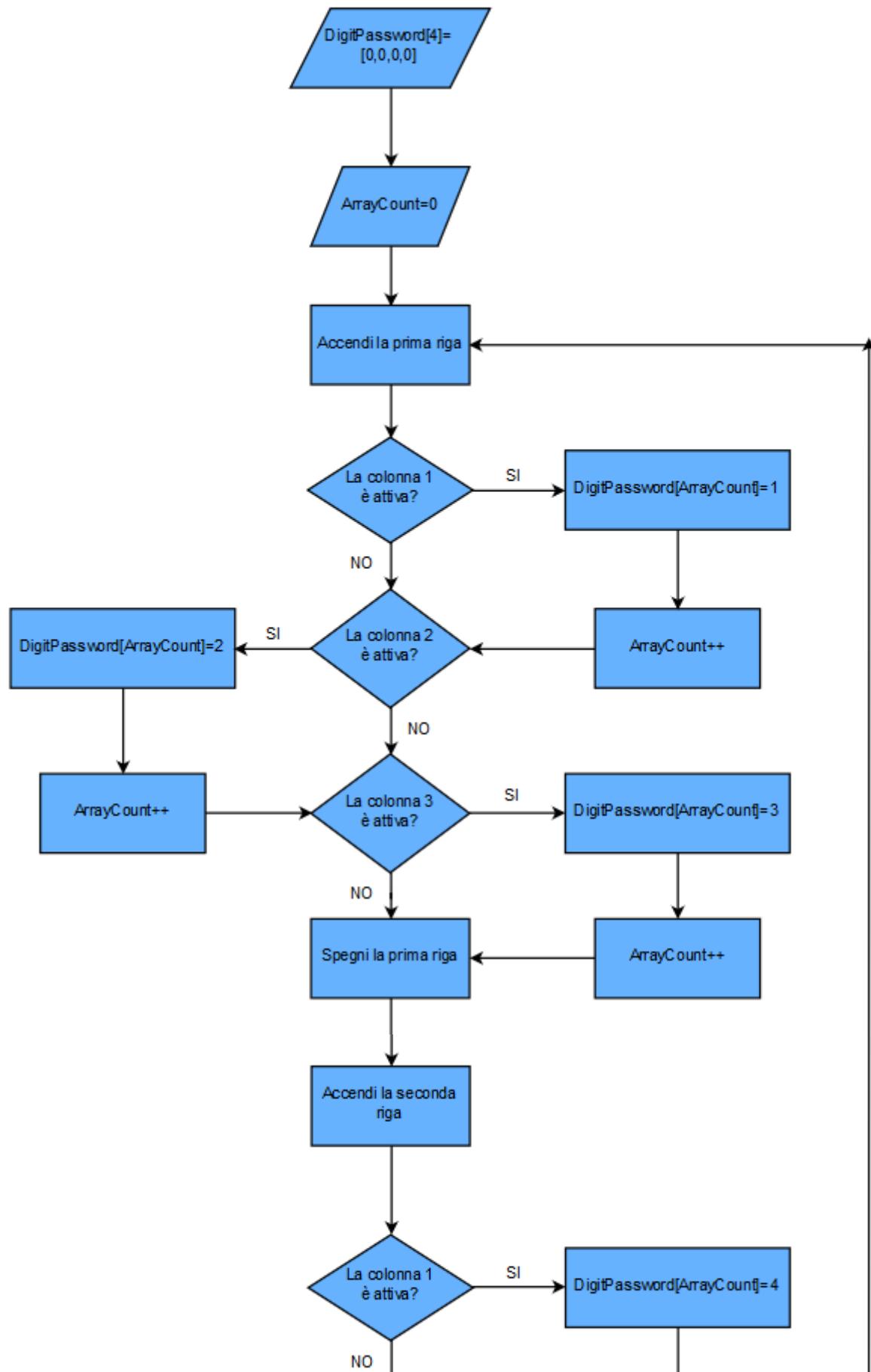
```

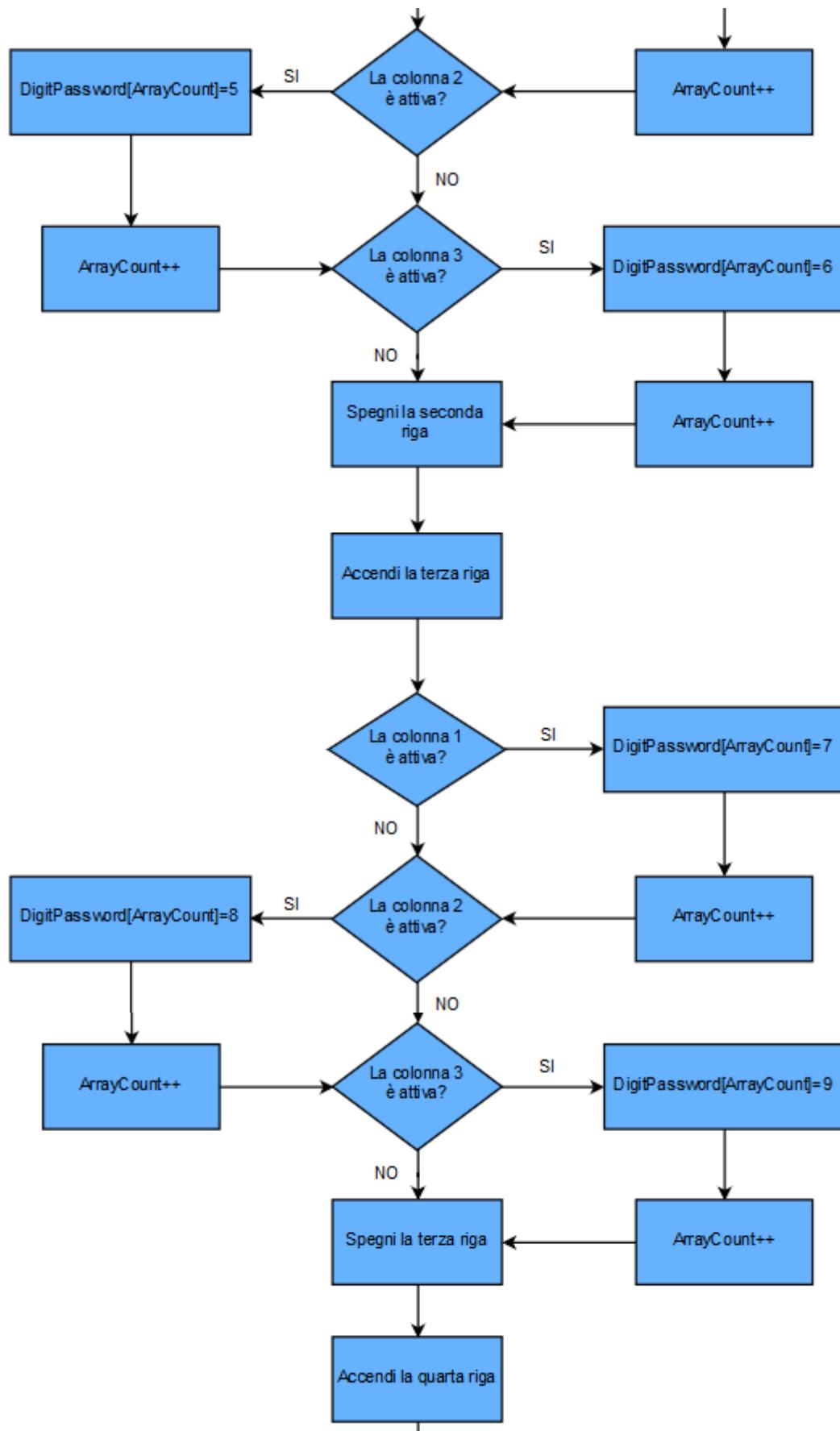
```

        ArrayCount--;
        DigitPassword[ArrayCount] = 0;
        __delay_ms(delayButton);
    }
    if(COL2 && ArrayCount<4)
    {
        DigitPassword[ArrayCount] = 0;           //NUMERO 0
        __delay_ms(delayButton);
        ArrayCount++;
    }
    if(COL3)                                //Verifica Password inserita
    {
        KeyPad=0;
        PassCheck=1;                      //Attiva lo stato di verifica della Password
        __delay_ms(delayButton);
    }
}

```

Il delay di 300 millisecondi alla pressione di ogni tasto è necessario per evitare che lo stesso carattere venga memorizzato più volte. Se però l'utente esercita una pressione prolungata per un tempo maggiore, lo stesso carattere verrà salvato in posizioni successive dell'array *DigitPassword*.





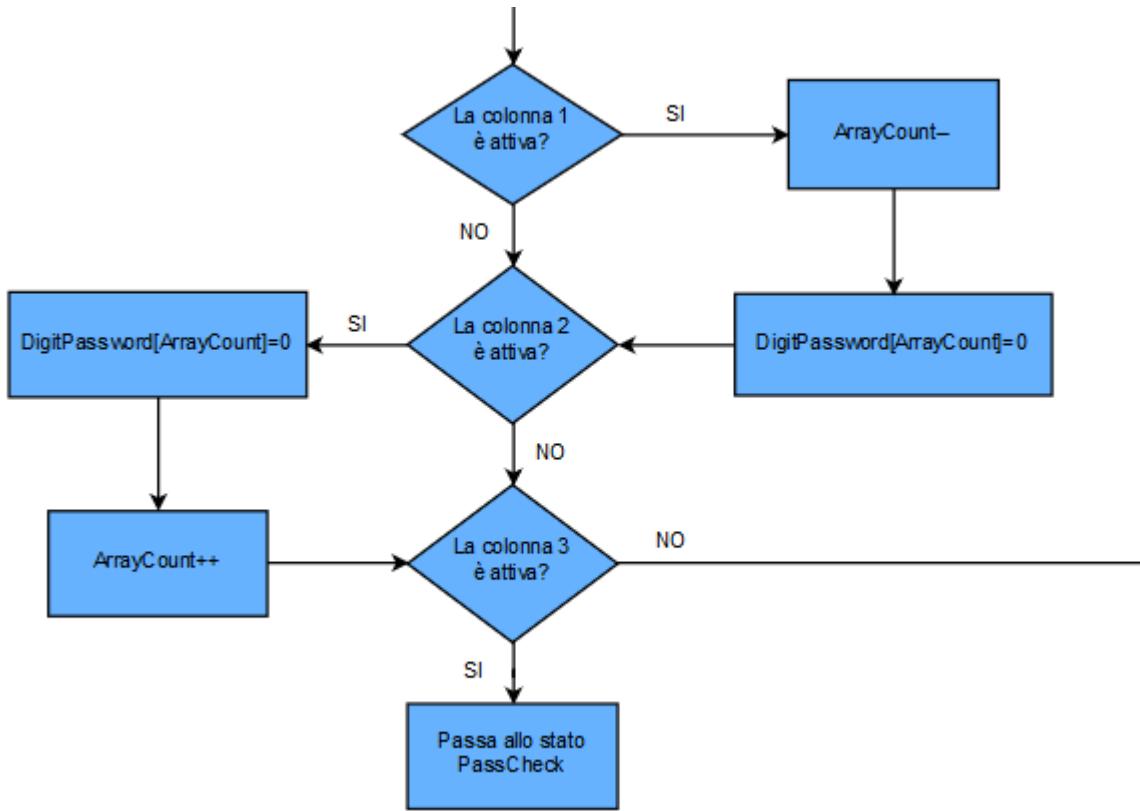


Fig. 4.2.4-2 Pseudocodice dello stato KeyPad

Lo stato *PassCheck* prevede una semplice verifica di coincidenza (matching) tra la password digitata dall’utente e quella ricevuta dal PIC Master via CAN. Se tutti i caratteri degli array nella stessa posizione coincidono, allora si accende un LED verde per due secondi e lo stato *OpenDoor* viene settato ad uno; se invece almeno uno è differente, si accende un LED rosso per due secondi, il contatore *TryCount* che conta i tentativi errati di inserire la password viene incrementato e lo stato *PassClean* viene settato ad uno.

Come prima, vediamo in seguito il codice implementato ed in Fig. 4.2.4-3 l’algoritmo:

```

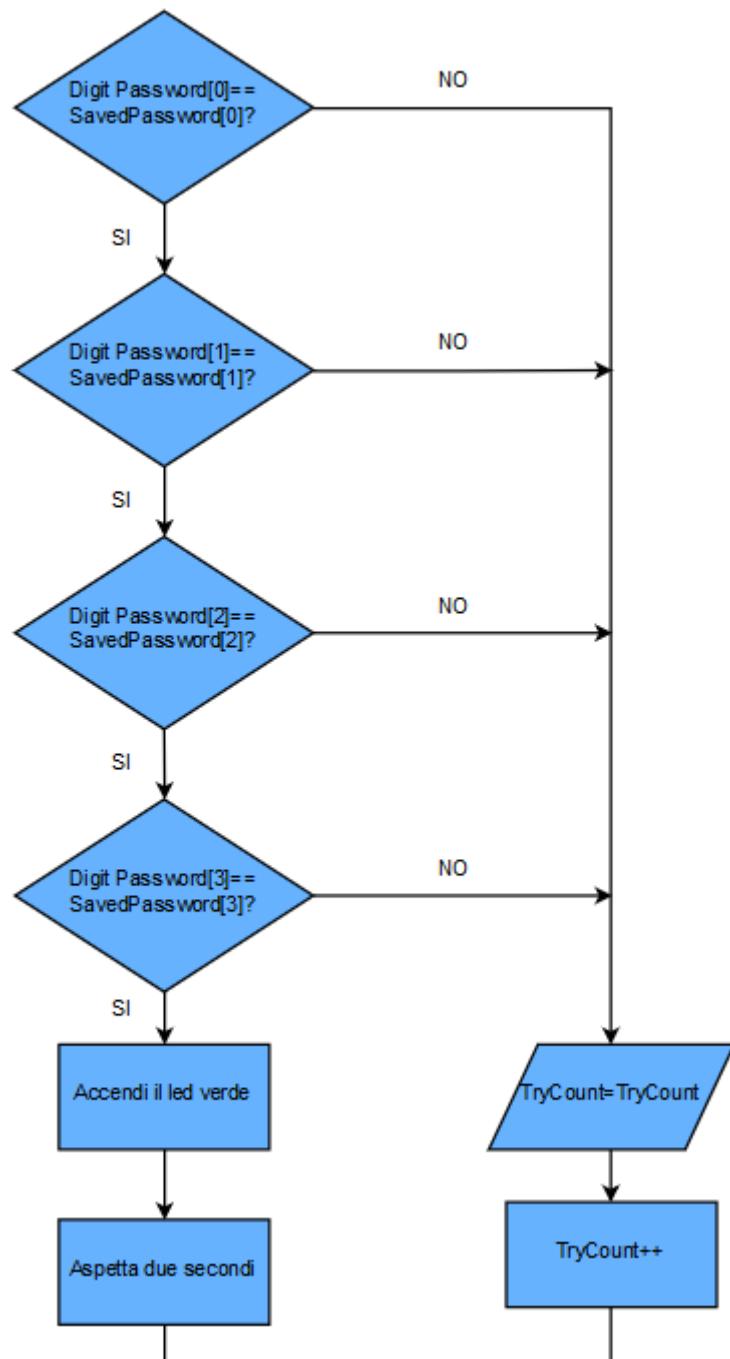
if(PassCheck==1) //Verifica coincidenza della password
{
    PassCheck=0;
    if (DigitPassword[0]==SavedPassword[0])
    { if (DigitPassword[1]==SavedPassword[1])
        { if (DigitPassword[2]==SavedPassword[2])
            { if (DigitPassword[3]==SavedPassword[3])
                {
                    LEDGreen=1;
                    __delay_ms(delayLed);
                    LEDGreen=0;
                    OpenDoor=1; //Passa alla funzione OpenDoor
                }
            else //Se l’ultimo carattere non è giusto
            {
                LEDRed=1;
                __delay_ms(delayLed);
                LEDRed=0;
            }
        }
    }
}

```

```

    PassClean= 1;      //Passa allo stato PassClean
    TryCount++;
}
}
else           //Se il terzo carattere non è giusto
{
    LEDRed=1;
    __delay_ms(delayLed);
    LEDRed=0;
    PassClean= 1;      //Passa allo stato PassClean
    TryCount++;
}
}
else           //Se il secondo carattere non è giusto
{
    LEDRed=1;
    __delay_ms(delayLed);
    LEDRed=0;
    PassClean= 1;      //Passa allo stato PassClean
    TryCount++;
}
}
else           //Se il terzo carattere non è giusto
{
    LEDRed=1;
    __delay_ms(delayLed);
    LEDRed=0;
    PassClean= 1;      //Passa allo stato PassClean
    TryCount++;
}
}

```



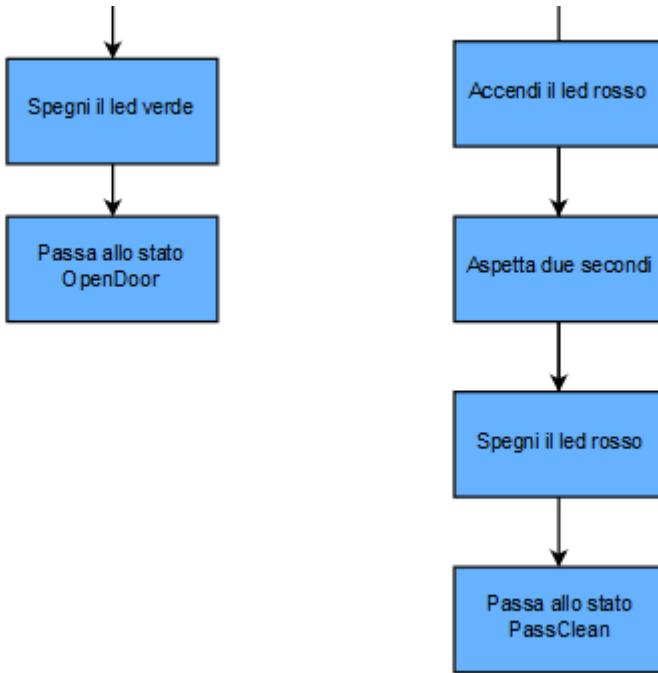


Fig. 4.2.4-3 Pseudocodice dello stato PassCheck

Lo stato *OpenDoor* abilita l'apertura della serratura, attivando il pin del PIC corrispondente all'alimentazione della serratura. Poi resetta la password digitata, azzerà i contatori di posizione nell'array e di numero di tentativi errati e forza il programma ad uscire dalla funzione "AccessRoutine" e tornare al main:

```

if (OpenDoor==1)
{
//Disattivo tutte le righe
    ROWA=0;
    ROWB=0;
    ROWC=0;
    ROWD=0;

    DigitPassword[0]=-1;
    DigitPassword[1]=-1;
    DigitPassword[2]=-1;
    DigitPassword[3]=-1;

    OpenDoor=0;
    ArrayCount=0;
    TryCount=0; //Resetto gli eventuali tentativi errati
    Lock=1;           //Abilito apertura porta
    return;
}

```

Lo stato *PassClean* invece, oltre a pulire tutti i campi dell'array *DigitPassword* e ad azzerare il contatore *ArrayCount*, verifica lo stato attuale del contatore *TryCount*:

- Se questo è pari a 3 lo azzerà e attiva lo stato *DoorBlocked*, settandolo ad uno;
- Se questo è inferiore a 3, quindi l'utente ha a disposizione almeno un altro tentativo, setta ad uno lo stato *KeyPad*.

In seguito, il codice implementato ed in Fig. 4.2.4-4 l'algoritmo dello stato *PassClean*:

```
if (PassClean==1)
{
    PassClean = 0;
    ArrayCount=0;

//Disattivo tutte le righe
    ROWA=0;
    ROWB=0;
    ROWC=0;
    ROWD=0;

    DigitPassword[0]=-1;
    DigitPassword[1]=-1;
    DigitPassword[2]=-1;
    DigitPassword[3]=-1;

if (TryCount==3)           //se i tentativi sbagliati sono 3
{
    TryCount=0;
    DoorBlocked=1; //passa allo stato porta bloccata
}
else
{
    Wait=1; //torna allo stato di attesa
}
}
```

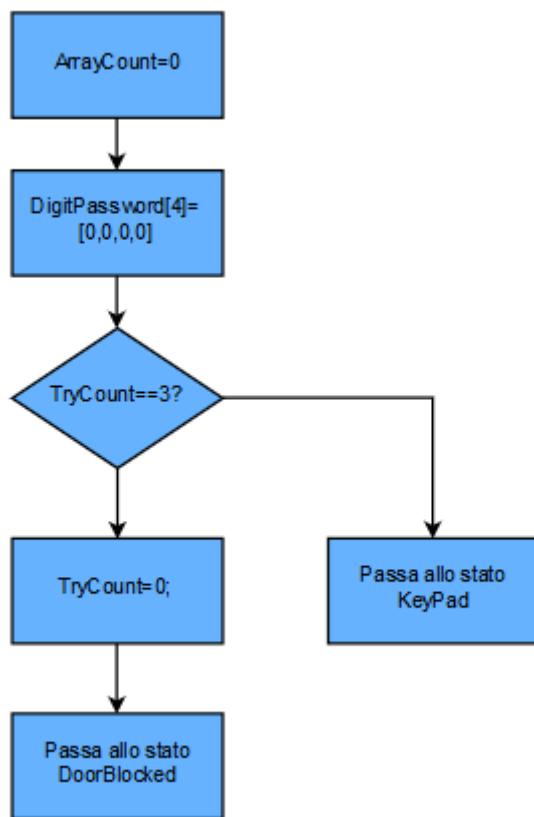


Fig. 4.2.4-4 Pseudocodice dello stato Passclean

Infine, lo stato *DoorBlocked* fa lampeggiare un LED rosso e, dopo aver resettato la password ricevuta, forza il programma ad uscire dalla funzione “AccessRoutine()”:

```

if (DoorBlocked==1)          //stato porta bloccata
{
    DoorBlocked=0;

//Disattivo tutte le righe
    ROWA=0;
    ROWB=0;
    ROWC=0;
    ROWD=0;

    LEDRed=1;                  //led rosso lampeggiante
    __delay_ms(100);
    LEDRed=0;
    __delay_ms(100);
    LEDRed=1;
    __delay_ms(100);
    LEDRed=0;
    __delay_ms(100);
    LEDRed=1;
    __delay_ms(100);
}

```

```
LEDRed=0;  
  
//Resetto la password per permettere nuovi accessi  
SavedPassword[0]=-1;  
SavedPassword[1]=-1;  
SavedPassword[2]=-1;  
SavedPassword[3]=-1;  
return;  
//Troppi errori fatti, bisogna ricominciare  
  
}  
}  
}  
//fine della funzione AccessRoutine()
```

#### 4.2.5 Routine di gestione stato porta

All'apertura di una porta, il sensore manda il segnare al rispettivo PIC che, vedendo che il pin **RBO** si trova a livello alto, attiva la routine di interrupt relativa all'eventualità di porta aperta. Tale routine setta ad 1 la variabile *Sensor*, azzera il flag **INTOIF** e disattiva l'interrupt del sensore:

```
if (INTOIF)
{
    Sensor=1; //Setto al primo livello lo stato della variabile sensore
    INTOIF=0; //Azzero il flag
    INTOIE=0; //Disattivo l'interrupt del sensore
}
```

Nel main file dello slave è quindi descritta la routine da eseguire nel caso una porta venga aperta in qualsiasi momento:

```
if (Sensor==1)
{
    Lock=0; //La porta può essere chiusa
    LEDGreen=0; //Spengo il LED
    //Aspetto un po' per dare il tempo alla persona di entrare e chiudere la porta
    if (Sensor_Wait==0)
    {
        TMROON=1;
    }
    if (Sensor_Wait==1) //Se il timer si è settato
    {
        Sensor_Wait=0; //Resetto lo stato della variabile
        if (PORTBbits.RBO) //Se RBO è 1, allora la porta è ancora aperta
        {
            Sensor=2; //Quindi devo segnalare la cosa al PIC Master
            SystemBlock(); //Chiamo la funzione di "porta già aperta"
        }
        else //Se RBO rimane a 0, la porta è chiusa
        {
            //Resetto la password per permettere nuovi accessi
            SavedPassword[0]=-1;
            SavedPassword[1]=-1;
            SavedPassword[2]=-1;
            SavedPassword[3]=-1;

            Sensor=0; //Bene, riporto a zero la variabile
            INTOIF=0; //Provo a resettare il flag
            INTOIE=1; //Riattivo l'interrupt sensore
        }
    }
}
if (Sensor==2)
{
    TMR1ON=1; //Attivo il Timer1
```

```

if (Sensor_Counter==3) //Circa 6 secondi
{
    Sensor_Counter=0; //Azzero il contatore per ricominciare il conto
    if (PORTBbits.RB0) //Da ora testo il PORTB, fino a quando non torna a zero
    {
        SystemBlock(); //Chiamo la funzione di "porta già aperta"
    }
    else //Se torna a 0
    {
        //Resetto la password per permettere nuovi accessi
        SavedPassword[0]=-1;
        SavedPassword[1]=-1;
        SavedPassword[2]=-1;
        SavedPassword[3]=-1;

        TMR1H=0; //Resetto la parte alta del Timer1
        TMR1L=0; //Resetto la parte bassa del Timer1
        TMR1ON=0; //Stoppo il Timer1

        Sensor=0; //Resetto la variabile di stato del sensore
        INTOIF=0; //Riazzero il flag
        INTOIE=1; //Riattivo l'interrupt del sensore
    }
}
}

```

Se la variabile *Sensor* è uguale ad uno per prima cosa viene settato a zero il livello del pin collegato alla serratura, permettendo alla porta di bloccarsi appena verrà chiusa. Dopodiché viene attivato il Timer0 che all'overflow impostato setta ad uno la variabile *Sensor\_Wait*:

```

if (TMROIF)
{
    TMROH=0; //Disattivo la parte alta del Timer0
    TMROL=0; //Disattivo la parte bassa del Timer0
    TMROON=0; //Disattivo il Timer0
    Sensor_Wait=1; //Setto ad 1 la variabile
    TMROIF=0;
}

```

A questo punto si verifica se la porta è ancora aperta controllando il livello del pin *RBO*:

```

if (Sensor_Wait==1) //Se il timer si è settato
{
    Sensor_Wait=0; //Resetto lo stato della variabile
    if (PORTBbits.RB0) //Se RBO è 1, allora la porta è ancora aperta
    {
        Sensor=2; //Quindi devo segnalare la cosa al PIC Master
        SystemBlock(); //Chiamo la funzione di "porta già aperta"
    }
    else //Se RBO rimane a 0, la porta è chiusa
    {
        Sensor=0; //Bene, riporto a zero la variabile
    }
}

```

```

INTOIF=0; //Provo a resettare il flag
INTOIE=1; //Riattivo l'interrupt sensore
}
}

```

- a) Se il livello di *RBO* è alto, la porta è ancora aperta: il programma setta a 2 la variabile *Sensor* e chiama la funzione *SystemBlock()*;
- b) Se il livello di *RBO* è basso, la porta si è richiusa: il programma riporta a 0 la variabile *Sensor* e riattiva l'interrupt del sensore.

La funzione *SystemBlock()* scrive sul CAN un messaggio (80,80,80,80,80), destinato al buffer 0 del PIC Master:

```

void SystemBlock(void)
{
    //Devo informare il Master della presenza di una porta aperta

    //Imposto la priorità dei messaggi
    TXB0CON=0b00000011;

    //Imposto l'identificativo messaggio
    TXB0SIDH=0b00000000;
    TXB0SIDL=0b11000000; //Scrivo al Master (Slave Tx -> Master RX)

    //Setto la lunghezza del messaggio (3 byte in questo caso)
    TXB0DLC=0b00000101;

    //Imposto il contenuto dei registri da inviare
    //TXBnDm=0b00000000 : (n=numero buffer; m=numero registro del buffer)
    TXB0D0=80; //Scrivo nel registro 0 del buffer 0
    TXB0D1=80; //Scrivo nel registro 1 del buffer 0
    TXB0D2=80; //Scrivo nel registro 2 del buffer 0
    TXB0D3=80; //Scrivo nel registro 3 del buffer 0
    TXB0D4=80; //Scrivo nel registro 4 del buffer 0

    //Setto il bit di trasmissione
    TXREQ0=1;
}

```

A questo punto, come descritto nel main file del PIC Master, quando quest'ultimo riceve il messaggio (80,80,80,80,80) al buffer 0, viene settata ad 1 la variabile *A\_Door\_is\_Open*:

```

if (Msg_0==1)
{
    //Cosa faccio ora che il buffer0 ha ricevuto un messaggio?
    if (RXB0D0==80 & RXB0D1==80 & RXB0D2==80 & RXB0D3==80 & RXB0D4==80)
    {
        //C'è una porta aperta!
        A_Door_is_Open=1;
    }

    Msg_0=0;
}

```

}

Nel main sono quindi definite le istruzioni da seguire quando *A\_Door\_is\_Open* è settato ad uno:

```
if (A_Door_is_Open==1)
{
    //Devo comunicare a Raspeberry/FPGA di bloccare gli accessi
    while (TRMT!=1);    //Aspetto di poter trasmettere un nuovo messaggio
    TXREG=0xDD;          //Trasmetto il codice di porta aperta
    A_Door_is_Open=0;      //Azzero la variabile
}
```

Il segnale di porta aperta viene quindi inviato al dispositivo al PIC Master, il quale a sua volta invierà un corrispondente carattere al dispositivo master di sistema, che non permetterà alcun nuovo accesso finché la porta non verrà chiusa.

#### 4.2.6 CAN Errors Handling (Slave)

La routine di gestione degli errori CAN implementata sui dispositivi Slave è una versione semplificata di quella descritta in precedenza per il PIC Master, ma è assolutamente simile per modalità di attivazione e logica attuativa, in particolare:

- I dispositivi Slave non gestiscono gli errori CAN attraverso il flag/interrupt `IRX`, ossia non gestiscono gli errori di trasmissione e/o ricezione di messaggi. Questa scelta progettuale è stata dettata sostanzialmente dal fatto che i dispositivi PIC comunicano con il resto del sistema esclusivamente attraverso la rete CAN e sono programmati per interagire unicamente (eccezione fatta per i controlli sull'accesso) con il PIC Master, dunque al verificarsi di un tale errore, il dispositivo non avrebbe modo di comunicare l'accadimento al resto del sistema in maniera robusta e sicura;
- I dispositivi Slave non gestiscono gli errori CAN derivanti dall'Overflow dei registri di ricezione per motivi similari al punto precedente;

La gestione degli errori operata dai dispositivi Slave (attraverso il flag `ERRIF`) si riduce dunque a quanto segue. Quando il flag `ERRIF` viene settato, a seguito del verificarsi di una condizione di errore CAN, e viene quindi generato un interrupt, il seguente blocco di codice viene eseguito:

```
if(ERRIF)
{
    //Cosa faccio se si verifica un errore sul CAN?
    CAN_Error_Check=1;
    ERRIF=0; //Azzero il flag
    ERRIE=0; //Devo disattivare temporaneamente questi interrupt per poter agire
}
```

Il quale, in modo assolutamente simile a quanto avviene nel PIC Master, setta la variabile di stato associata alla routine di gestione degli errori, azzera il flag e disabilita momentaneamente questo tipo di interrupt.

Questo, nel main porta ad eseguire il successivo blocco:

```
if(CAN_Error_Check==1)
{
    CAN_Error_Handling();
}
```

Il quale contiene la chiamata alla funzione vera e propria di gestione degli errori, contenuta nel file **CAN\_Functions.c**:

```
void CAN_Error_Handling(void)
{
    if(RXBP==1 || TXBP==1 || TXBO==1)
    {
        //Gli errori di ricezione e/o trasmissione sono tanti o troppi, il dispositivo necessita un reset
        RESET();
    }

    RXB0OVFL=0;
    RXB1OVFL=0;
    ERRIE=1;
}
```

Come detto in precedenza questa routine non gestisce tutti gli errori ma solo il caso più grave, ossia il caso in cui (seguendo la classificazione usata per il PIC Master) si verifichi un errore/condizione di livello 3, la quale sancisce l'impossibilità del dispositivo di operare correttamente e per cui è previsto un tentativo di ripristino

tramite reset del dispositivo (dato che i flag relativi agli Overflow dati sono associati allo stesso registro, in caso di interrupt essi vengono sempre azzerati, ma nessuna azione specifica viene presa).

#### 4.2.7 PIC porta principale

Come già detto, tutti i PIC Slave associati alle porte sono stati programmati allo stesso identico modo (eccezione fatta per le maschere ed i filtri ovviamente differenti per ogni PIC). L'unico PIC con un codice leggermente diverso è quello associato alla porta principale di accesso al corridoio: per questa porta non è necessario avere una routine di accesso come per le altre, infatti il riconoscimento dell'utente viene gestito dal componente a monte del sistema (Raspberry in modalità di funzionamento normale ed FPGA in modalità ridotta). Quando un utente si è identificato ed ha richiesto l'accesso ad una specifica porta, il PIC Slave che riceve la password dell'utente scrive il messaggio 19 se l'accesso è stato concesso o 29 se è stato negato (come spiegato nella *sezione 4.2.4* di questo documento) sul CAN con l'identificativo del PIC Slave della porta principale, il quale leggere il messaggio ed in base al suo contenuto svolge uno dei seguenti blocchi di codice:

```
else if (RXBODO==19)
{
    Lock=1; //La porta può essere aperta
    first_req=1;
    //Se una precedente richiesta è stata fatta prima di 30 secondi
    if (TMROON==1) //Se il timer era già attivo, lo resetto e lo spegno
    {
        TMROH=0;
        TMROL=0;
        TMROON=0;

        Sensor=0; //Bene, riporto a zero la variabile
        INTOIF=0; //Provo a resettare il flag
        INTOIE=1; //Riattivo l'interrupt sensore
    }
}

else if (RXBODO==29)
{
    AccessDenied=1;
    Msg_0=1;
}
```

Per le differenze riguardanti i settaggi si rimanda alla *sezione 4.2.2* di questo documento.

## 5 Conclusioni e considerazioni di progetto

Questo capitolo si occuperà di trattare tutte quelle condizioni/configurazioni/situazioni che non sono state precedentemente esplicitamente considerate e/o per cui non è stata prevista un'adeguata strategia di gestione. In altri termini verranno nel seguito esposti alcuni tra i possibili sviluppi futuri di questo progetto che per motivi di tempo e/o denaro non è stato possibile portare a compimento.

- **Gestione dell'apertura delle porte dall'interno degli ambienti considerati:**

Durante tutto l'arco progettuale è stata assunta come ipotesi la presenza, nella realizzazione finale, di porte dotate di apertura meccanica manuale sul lato interno (porte con sistema antipanico), per questo motivo non sono state gestite né a livello software, né tantomeno a livello hardware tutte quelle situazioni di conflitto/criticità per la sicurezza la cui origine è data dall'uscita verso l'esterno di un soggetto, precedentemente autorizzato in ingresso. Infatti, non sono state riportate ad esempio strategie per la gestione dell'apertura delle porte dall'interno nel momento in cui queste siano in attesa di ricevere una password dall'utente tramite tastierino (durante la routine d'accesso), o del caso in cui una porta venga immediatamente aperta dopo che un nodo ha ricevuto e salvato una password di accesso MA non è ancora entrato nella routine di accesso vera e propria. Per entrambe queste casistiche le soluzioni sono fortemente dipendenti dall'ambito applicativo in cui questo progetto potrebbe andare a trovare la propria realizzazione, infatti al variare ad esempio dell'importanza della sicurezza degli ambienti intesa come controllo dei flussi del personale uscente e/o entrante tali soluzioni potrebbero limitarsi all'aggiunta di qualche routine software (Timer per il check periodo dello stato delle porte...) oppure potrebbero richiedere sistemi di sicurezza paralleli (telecamere di sicurezza, sistemi di allarme, etc...). Per questioni di tempo la valutazione di tutte queste casistiche non è stata possibile ed è pertanto stata annoverata tra gli sviluppi possibili.

- **Creazione di un database più capiente e flessibile:**

Il database utilizzato per il progetto risulta limitato al salvataggio di 16 utenti, il quale può essere portato a 48 utilizzando la parte di memoria che fino ad ora non è stata impiegata. Qualora sia necessario considerare un pool di utenza più ampio, sarebbe necessario l'impiego di supporti di memoria esterni.

- **Taratura dei Timer:**

Come esposto in precedenza, i Timer implementati nella rete PIC sono stati settati per facilitare la prototipazione e lo sviluppo del modello, tuttavia le loro tempistiche dovrebbero essere riviste sulla base dell'ambito applicativo reale in cui il sistema si troverà a funzionare.

- **Sostituzione e/o parallelizzazione del sistema dei tastierini con componentistica più avanzata:**

Al momento il sistema gestisce unicamente password numeriche, costituite da cifre comprese tra 0 e 9. Si potrebbe pensare di rimuovere questa limitazione tramite upgrade software, modificando il programma sui dispositivi di controllo (FPGA, Raspberry, PIC), e hardware sostituendo i tastierini a membrana con tastiere vere e proprie. In alternativa, tramite l'implementazione di sensori di rilevazione di impronte digitali per la convalida degli accessi, il problema potrebbe essere aggirato, tuttavia questo comporterebbe una revisione sostanziale della gestione accessi.

- **Upgrade software del Master primario di sistema (Raspberry):**

Sulla base delle specifiche dell'applicazione reale potrebbe essere necessario/doveroso rivedere e ottimizzare la gestione dei processi, ad esempio modularizzando ulteriormente le funzioni del programma. Sarebbe inoltre possibile migliorare l'interfaccia utente aggiungendo più opzioni per la gestione del database e/o raffinandone gli aspetti grafico-estetici al fine di renderla più "user-friendly".

- **Implementazione futura di ridondanza hardware a livello dei nodi della rete CAN:**

A livello di rete di nodi CAN i nostri dispositivi Slave comunicano tra loro e con il PIC Master unicamente tramite la suddetta rete e non dispongono ad ora di un sistema di Fault-Tolerance contro guasti fatali per il nodo (PIC bruciato, Sensore non operativo, etc...). Inoltre, non esiste un canale di riserva per la comunicazione qualora la rete CAN risulti malfunzionante.

- **Interfacciamento con altri sistemi:**

Non è stata, ad ora, prevista alcuna modalità di interfacciamento dell'impianto sopra descritto con altri sistemi esterni preesistenti o meno (ad esempio: sistemi d'allarme, antincendio, etc...).

- **Upgrade della gestione/visualizzazione delle situazioni di guasto:**

Ad esempio, migliorando/sviluppando le routine di diagnostica dei malfunzionamenti:

1. Display/Risoluzione più efficace degli errori CAN;
2. Display in tempo reale dello status dello Switch;
3. Introduzione di una routine per la visualizzazione più immediata dello stato di funzionamento della rete di comunicazione RS-232.

- **Implementazione del salvataggio del database utenti sul dispositivo Master di sistema ausiliario:**

Allo stato attuale ad ogni accensione il database viene creato sul Raspberry in memoria non volatile e viene poi inviato all'FPGA, la quale però salva i dati sulla sola RAM, per cui in caso di spegnimento perde il database. Non è stato gestito in questo senso il caso in cui il dispositivo primario (Raspberry in questo caso) sia malfunzionante sin dalla prima accensione.

## Appendice A: File supplementari per il PIC Master

Sono riportati di seguito i file relativi al PIC Master, in particolare il file **Def\_Library.h** (Fig. A-1, A-2, A-3), il file **Function\_Protoypes.h** (Fig. A-4) e il file **Variables\_Declaration.c** (Fig. A-5).

```
1 #define _XTAL_FREQ 1000000
2 #include <xc.h>
3 #include <stdio.h>
4
5 //Definisco nomi comodi per identificare alcuni bit
6 #define CCP1IE PIE1bits.CCP1IE
7 #define CCP1IF PIR1bits.CCP1IF
8 #define IRXIF PIR3bits.IRXIF
9 #define IRXIE PIE3bits.IRXIE
10 #define WAKIF PIR3bits.WAKIF
11 #define WAKIE PIE3bits.WAKIE
12 #define ERRIF PIR3bits.ERRIF
13 #define ERRIE PIE3bits.ERRIE
14 #define TXB2IF PIR3bits.TXB2IF
15 #define TXB2IE PIE3bits.TXB2IE
16 #define TXB1IF PIR3bits.TXB1IF
17 #define TXB1IE PIE3bits.TXB1IE
18 #define TXBOIF PIR3bits.TXB0IF
19 #define TXBOIE PIE3bits.TXB0IE
20 #define RXB0FUL RXB0CONbits.RXFUL
21 #define RXB1FUL RXB1CONbits.RXFUL
22 #define RXBOIF PIR3bits.RXBOIF
23 #define RXBOIE PIE3bits.RXBOIE
24 #define RXB1IF PIR3bits.RXB1IF
25 #define RXB1IE PIE3bits.RXB1IE
26 #define PEIE INTCONbits.PEIE
27 #define INTOIF INTCONbits.INT0IF
28 #define TMROON T0CONbits.TMROON
29 #define TMROIE INTCONbits.TMROIE
30 #define TMROIF INTCONbits.TMROIF
31 #define TMR1ON T1CONbits.TMR1ON
32 #define TMR1IF PIR1bits.TMR1IF
33 #define TMR1IE PIE1bits.TMR1IE
34 #define TMR3IE PIE2bits.TMR3IE
35 #define TMR3IF PIR2bits.TMR3IF
36 #define TMR3ON T3CONbits.TMR3ON
37 #define TXREQ0 TXB0CONbits.TXREQ
38 #define TXREQ1 TXB1CONbits.TXREQ
39 #define TXREQ2 TXB2CONbits.TXREQ
40 #define LEDVerde LATBbits.LB5 // DA TOGLIERE
41 #define LEDRedCAN LATBbits.LB4
42 #define LEDRedFPGA LATCbits.LC3
43 #define LEDGreenPI LATDbits.LD0
44 #define LEDBlueSWITCH LATDbits.LD1
```

Fig. A-1 Def\_Library Master pt 1

```

45 #define LEDWhite LATBbits.LB1
46 #define Lock LATDbits.LD2
47 #define RXBOOVFL COMSTATbits.RXBOOVFL
48 #define RXBLOVFL COMSTATbits.RXBLOVFL
49 #define TXBO COMSTATbits.TXBO
50 #define TXBP COMSTATbits.TXBPF
51 #define RXBP COMSTATbits.RXBPF
52 #define TXWARN COMSTATbits.TXWARN
53 #define RXWARN COMSTATbits.RXWARN
54 #define EWARN COMSTATbits.EWARN
55 #define S0 PORTDbits.RD7 //Segnale di controllo per lo switch
56 #define S1 PORTDbits.RD6 //Segnale di controllo per lo switch
57 #define ALIVE 0xFF //Rappresenta il carattere di alive
58 #define ERROR 0xEE //Rappresenta il carattere di errore (PER ORA)
59 #define TRMT TXSTAbits.TRMT
60 #define CREN RCSTAbits.CREN
61 #define TXEN TXSTAbits.TXEN
62
63
64 //Definisco variabili globali utili per gestione degli stati del sistema
65 extern char A_Door_is_Open; //Variabile associata alla rilevazione di porta aperta da parte del sensore magnetico
66 extern char Alive_Check; //Variabile associata all'avvio della routine di Alive
67 extern char Check_Slave; //Variabile di stato associata alla routine di Alive
68 extern char Check_Req; //Variabile utile per effettuare il check sicuro all'interno del ciclo di Alive
69 extern char Broken_Door; //Variabile di stato per identificare la condizione di porta rottta
70 extern char CAN_Invalid_Msg; //Variabile usata per gestire la situazione di messaggio CAN non valido
71 extern char Counter_Inv_Msg; //Contatore per il numero di Messaggi CAN errati
72 extern char CAN_Error_Check; //Variabile che abilita il Check degli errori CAN
73
74 //Contatori per implementazione handling errori CAN
75 extern char RXBOVLF_Counter;
76
77 //Variabili di stato associate allo stato dei singoli PIC-slave (Fintanto che la variabile è 0 lo slave è funzionante)
78 extern char Slave1_status; //Slave 1
79 extern char Slave2_status;
80 extern char Slave3_status;
81 extern char Slave4_status;
82 extern char Slave5_status;
83
84 //Variabili associate all'avvenuta (o meno) trasmissione della condizione di malfunzionamento degli slave
85 extern char S1_st_Transmit; //Slave 1
86 extern char S2_st_Transmit;
87 extern char S3_st_Transmit;

```

Fig. A-2 Def\_Library Master pt 2

```

88 extern char S4_st_Transmit;
89 extern char S5_st_Transmit;
90
91 //Variabili modulo EUSART e Switch
92 extern unsigned char FSM_Status; //Variabile che definisce lo stato
93 extern unsigned char incoming_data; //Variabile che memorizza il dato in arrivo sul registro RCREG
94 extern unsigned char read_RCREG; //Flag che controlla la routine di gestione dei dati in arrivo
95 extern unsigned char counter; //Variabile che definisce il valore del contatore relativo al timer3
96 extern unsigned char switches; /*Variabile che tiene conto del numero consecutivo di switch avvenuti senza l'invio
97 | di un ALIVE*/
98 extern unsigned char error_flag; //Flag che attiva la routine di errore
99 extern unsigned char data_number; //Contatore relativo al numero di byte di dati ricevuti durante un pacchetto dati
100 extern unsigned char SwitchStatusDisplay; // Variabile di stato della funzione di display stato connessioni switch
101 extern unsigned char Timer3OverflowFlag; //Variabile di stato associata alla funzione di conteggio Overflow Timer3
102 extern unsigned char Raspberry; //Variabile di stato del Raspberry
103 extern unsigned char FPGA; //Variabile di stato dell'FFGA
104 extern unsigned char Double_Switch; //Variabile di stato associata al doppio switch

```

Fig. A-3 Def\_Library Master pt 3

```

1 //CAN Functions
2 void Initial_Test (void);
3 void Password_Test(void);
4 void Are_you_there(void);
5 void CAN_Error_Handling(void);
6
7 //CAN Modes
8 void CAN_Normal_Mode(void);
9 void CAN_Loopback_Mode(void);
10 void CAN_Config_Mode(void);
11 void CAN_Sleep_Mode(void);
12
13 //CAN Interrupts
14 void CAN_ISR(void);
15
16
17 //General Functions
18 void Initial_Reset(void);
19 void Broken_Door_Signal(void);
20
21 [-] //EUSART Functions
22     //Aggiornamento dello stato
23     void EUSART_FSM_refresh (void);
24     //Routine di ALIVE del modulo EUSART
25     void EUSART_ImAlive (void);
26     //Gestione dei junk
27     void Junk_manager (unsigned char junk);
28     //Gestione dei dati
29     void Data_manager (void);
30     //Gestione dello switch
31     void Switch_manager (void);
32     //Controllo sul carattere di ALIVE
33     void EUSART_Alive_check (void);
34     //Macchina a stati per gestione pacchetto dati
35     void EUSART_FSM (void);
36     //Display della situazione EUSART
37     void EUSART_Status_Display (void);

```

*Fig. A-4 Function\_Protoypes Master*

```

1 //Definisco variabili globali utili per gestione degli stati del sistema
2 char A_Door_is_Open=0; //Variabile associata alla rilevazione di porta aperta da parte del sensore magnetico
3 char Alive_Check=0; //Variabile associata all'avvio della routine di Alive
4 char Check_Slave=0; //Variabile di stato associata alla routine di Alive
5 char Check_Req=0; //Variabile utile per effettuare il check sicuro all'interno del ciclo di Alive
6 char Broken_Door=0; //Variabile di stato per identificare la condizione di porta rotta
7 char CAN_Invalid_Msg=0; //Variabile usata per gestire la situazione di messaggio CAN non valido
8 char Counter_Inv_Msg=0; //Contatore per il numero di Messaggi CAN errati
9 char CAN_Error_Check=0; //Variabile che abilita il Check degli errori CAN
10
11 //Contatori per implementazione handling errori CAN
12 char RXBOVLF_Counter=0;
13
14 //Variabili di stato associate allo stato dei singoli PIC-slave (Fintanto che la variabile è 0 lo slave è funzionante)
15 char Slave1_status=0; //Slave 1
16 char Slave2_status=0;
17 char Slave3_status=0;
18 char Slave4_status=0;
19 char Slave5_status=0;
20
21 //Variabili associate all'avvenuta (o meno) trasmissione della condizione di malfunzionamento degli slave
22 char S1_st_Transmit=0; //Slave 1
23 char S2_st_Transmit=0;
24 char S3_st_Transmit=0;
25 char S4_st_Transmit=0;
26 char S5_st_Transmit=0;
27
28 //Variabili modulo EUSART e Switch
29 unsigned char FSM_Status=0; //Variabile che definisce lo stato
30 unsigned char incoming_data=0; //Variabile che memorizza il dato in arrivo sul registro RCREG
31 unsigned char read_RCREG=0; //Flag che controlla la routine di gestione dei dati in arrivo
32 unsigned char counter=0; //Variabile che definisce il valore del contatore relativo al timer3
33 unsigned char switches=0; /*Variabile che tiene conto del numero consecutivo di switch avvenuti senza l'invio
34 | di un ALIVE*/
35 unsigned char error_flag=0; //Flag che attiva la routine di errore
36 unsigned char data_number=0; //Contatore relativo al numero di byte di dati ricevuti durante un pacchetto dati
37 unsigned char SwitchStatusDisplay=0; // Variabile di stato della funzione di display stato connessioni switch
38 unsigned char Timer3OverflowFlag=0; //Variabile di stato associata alla funzione di conteggio Overflow Timer3
39 unsigned char Raspberry=0; //Variabile di stato del Raspberry
40 unsigned char FPGA=0; //Variabile di stato dell'FPGA
41 unsigned char Double_Switch=0; //Variabile di stato associata al doppio switch

```

*Fig. A-5 Variables\_Declaration Master*

## Appendice B: File supplementari per i PIC Slave

Sono riportati di seguito i file relativi ai PIC Slave, in particolare il file **Def\_Library.h** (Fig. B-1, B-2), il file **Function\_Protoypes.h** (Fig. B-3) e il file **Variables\_Declaration.c** (Fig. B-4).

```
1 #define _XTAL_FREQ 1000000
2
3 #include <xc.h>
4 #include <stdio.h>
5
6 //Definisco nomi comodi per identificare alcuni bit
7 #define CCP1IE PIE1bits.CCP1IE
8 #define CCP1IF PIR1bits.CCP1IF
9 #define IRXIF PIR3bits.IRXIF
10 #define IRXIE PIE3bits.IRXIE
11 #define WAKIF PIR3bits.WAKIF
12 #define WAKIE PIE3bits.WAKIE
13 #define ERRIF PIR3bits.ERRIF
14 #define ERRIE PIE3bits.ERRIE
15 #define TXB2IF PIR3bits.TXB2IF
16 #define TXB2IE PIE3bits.TXB2IE
17 #define TXB1IF PIR3bits.TXB1IF
18 #define TXB1IE PIE3bits.TXB1IE
19 #define TXBOIF PIR3bits.TXBOIF
20 #define RXB0FUL RXB0CONbits.RXFUL
21 #define RXB1FUL RXB1CONbits.RXFUL
22 #define RXBOIF PIR3bits.RXBOIF
23 #define RXBOIE PIE3bits.RXBOIE
24 #define RXB1IF PIR3bits.RXB1IF
25 #define RXB1IE PIE3bits.RXB1IE
26 #define PEIE INTCONbits.PEIE
27 #define INTOIE INTCONbits.INTOIE
28 #define INTOIF INTCONbits.INTOIF
29 #define TMROON T0CONbits.TMROON
30 #define TMROIE INTCONbits.TMROIE
31 #define TMROIF INTCONbits.TMROIF
32 #define TMR1ON T1CONbits.TMR1ON
33 #define TMR1IF PIR1bits.TMR1IF
34 #define TMR1IE PIE1bits.TMR1IE
35 #define TMR3ON T3CONbits.TMR3ON
36 #define TMR3IF PIR2bits.TMR3IF
37 #define TXREQ0 TXB0CONbits.TXREQ
38 #define TXREQ1 TXB1CONbits.TXREQ
39 #define TXREQ2 TXB2CONbits.TXREQ
40 #define RXB0OVFL COMSTATbits.RXB0OVFL
41 #define RXB1OVFL COMSTATbits.RXB1OVFL
42 #define TXBO COMSTATbits.TXBO
43 #define TXBP COMSTATbits.TXBP
44 #define RXBP COMSTATbits.RXBP
```

Fig. B-1 Def\_Library Slave pt 1

```

45 #define TXWARN COMSTATbits.TXWARN
46 #define RXWARN COMSTATbits.RXWARN
47 #define EWARN COMSTATbits.EWARN
48
49 //Definizioni inutili al codice, utili solo alla leggibilità e alle connessioni
50 //Output
51 #define LEDRed LATBbits.LB4
52 #define LEDGreen LATBbits.LB5
53 #define LEDWhite LATBbits.LB1
54 #define Lock LATDbits.LD2
55 #define ROWA LATCbits.LC4
56 #define ROWB LATCbits.LC5
57 #define ROWC LATCbits.LC6
58 #define ROWD LATCbits.LC7
59 //Input
60 #define COL1 PORTDbits.RD7
61 #define COL2 PORTDbits.RD6
62 #define COL3 PORTDbits.RD5
63
64
65 //Definisco variabili globali utili per gestione degli stati del sistema
66 extern char Alive_Ans;           //Variabile associata alla necessità di mandare un messaggio di Alive
67 extern char Test;                //Variabile associata al test iniziale
68 extern char Msg_0;               //Variabile associata alla routine di ricezione
69 extern char Porta_Aperta;        //Variabile associata alla rilevazione di porta aperta da parte del sensore magnetico
70 extern char Sensor_Wait;         //Variabile di stato che sostituisce il delay di 30 secondi quando Sensor=1
71 extern char Sensor_Counter;      //Variabile usata per il contatore che sostituisce il delay di 5 secondi quando Sensor=2
72 extern char CAN_Error_Check;     //Variabile che abilita il Check degli errori CAN
73 extern char Sensor;              //Variabile si stato funzione sensore
74
75 //Variabili funzione tastierino
76 extern const int delayButton;
77 extern const int delayLed;
78 extern int TryCount;
79 extern int ArrayCount;
80 extern char PassClean;
81 extern char PassCheck;
82 extern char OpenDoor;
83 extern signed int SavedPassword[4];
84 extern signed int DigitPassword[4];
85 extern char PasswordReceived;
86 extern char KeyPad;
87 extern char Wait;
88 extern char DoorBlocked;
89 extern int Counter;
90 extern char AccessDenied;

```

Fig. B-2 Def\_Library Slave pt 2

```

1 //CAN Functions
2 void IamAlive(void);
3 void SystemBlock(void);
4 void Slave_Receive_Routine(void);
5 void CAN_Error_Handling(void);
6
7 //CAN Modes
8 void CAN_Normal_Mode(void);
9 void CAN_Loopback_Mode(void);
10 void CAN_Config_Mode(void);
11 void CAN_Sleep_Mode(void);
12
13 //CAN Interrupts
14 void CAN_ISR(void);
15
16
17 //General Functions
18 void Initial_Reset(void);
19 void AccessRoutine(void);

```

Fig. B-3 Function\_Protoypes Slave

```

1 //Definisco variabili globali utili per gestione degli stati del sistema
2 char Alive_Ans=0;           //Variabile associata alla necessità di mandare un messaggio di Alive
3 char Test=0;                //Variabile associata al test iniziale
4 char Porta_Aperta=0;        //Variabile associata alla rilevazione di porta aperta da parte del sensore magnetico
5 char Msg_0=0;                //Variabile associata alla ricezione messaggi sul buffer di ricezione CAN RXBO
6 char Sensor_Wait=0;         //Variabile di stato che sostituisce il delay di 30 secondi quando Sensor=1
7 char Sensor_Counter=0;      //Variabile usata per il contatore che sostituisce il delay di 5 secondi quando Sensor=2
8 char CAN_Error_Check=0;     //Variabile che abilita il Check degli errori CAN
9 char Sensor=0;              //Variabile si stato funzione sensore
10
11 //Variabili funzione tastierino
12 const int delayButton=300;
13 const int delayLed=200;
14 int TryCount=0;
15 int ArrayCount=0;
16 char PassClean=0;
17 char PassCheck=0;
18 char OpenDoor=0;
19 signed int SavedPassword[4]={-1,-1,-1,-1};
20 signed int DigitPassword[4]={-1,-1,-1,-1};
21 char PasswordReceived;
22 char KeyPad=0;
23 char Wait=0;
24 char DoorBlocked=0;
25 int Counter=0;
26 char AccessDenied=0;

```

*Fig. B-4 Variables\_Declaration Slave*