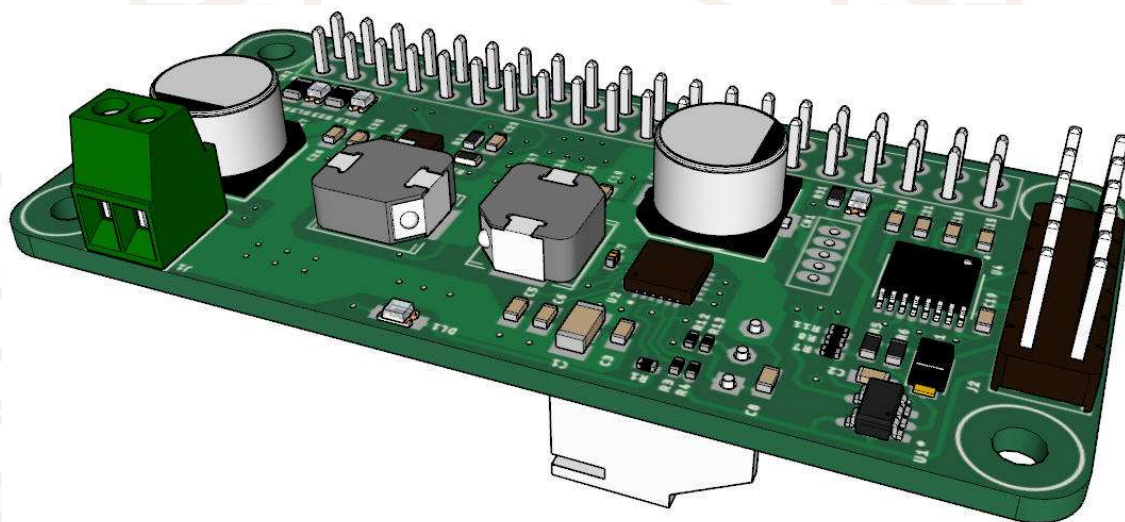


MiniUps v1.0



FUNZIONALITA' ED APPLICAZIONI

MiniUps una soluzione per dotare il Raspberry Pi di funzionalità UPS con la possibilità di configurare il sistema in funzione della batteria LiPo utilizzata come riserva di backup.

Si tratta di una vera e propria centrale di ricarica automatica per batterie LiPo! Applicazioni tipiche sono la realizzazione di sistemi di alimentazione per sistemi elettronici che richiedano un backup energetico per prevenire la perdita di dati come il classico Raspberry Pi (in tutti i suoi allestimenti e versioni) oppure Arduino o più in generale per qualsiasi sistema elettronico che per il quale sia richiesto un funzionamento affidabile h24.

La tensione di alimentazione principale è compresa tra 3.9 V e 14 V, come tensione nominale si consiglia di utilizzare una tensione $5V \pm 5\%$ con capacità in corrente da almeno 3 A fornibile su un apposito morsetto a vite presente sulla scheda. Nel caso in cui l'alimentatore non fosse in grado di erogare tutta la corrente necessaria, la scheda, entro certi limiti, è comunque in grado di compensare la riduzione di tensione diminuendo l'assorbimento di corrente in ingresso. Il processo di carica della batteria può portare a notevole generazione di calore, il sistema si autoregola gestendo la corrente di carica in modo da mantenere la temperatura all'interno del campo operativo dello stadio di carica.

Quando la tensione di alimentazione principale è presente il sistema provvede, in modo automatico e completamente trasparente all'utente, a ricaricare, se necessario, la batteria di riserva ed al contempo se richiesto alimenta anche il carico in uscita; nel caso in cui la tensione di alimentazione principale venga a mancare, anche in modo repentino, il sistema commuta la linea di alimentazione in uscita attingendo energia dalla batteria tampone, il passaggio tra le due condizioni è sostanzialmente istantaneo, in modo tale da proteggere il Raspberry (oppure il carico connesso sull'uscita della scheda) da una brusca interruzione della tensione di alimentazione. L'autonomia durante il funzionamento in modalità di backup dipende essenzialmente dalla capacità utilizzata per la batteria di riserva che, nel caso della batteria fornita di serie con la scheda, è tale da garantire un intervallo di autonomia di parecchi minuti con correnti in uscita dell'ordine dei 3 A, prima che la scheda provveda a rimuovere l'alimentazione sul carico preservando così la batteria da una condizione di scarica profonda.

Il **MiniUps** è quindi una scheda di ricarica rapida intelligente per batterie LiPo singola cella da 3.7 V con corrente di ricarica massima da 3.0 A. Realizzata con un circuito stampato multistrato è dotata di un ingresso di accensione/spengimento attivo basso (predisposto per la connessione di un pulsante normalmente aperto connesso verso lo 0V) ed è in grado di notificare al mondo esterno l'imminente spegnimento mediante un microcontrollore integrato sulla scheda stessa. In aggiunta, a completa, disposizione dell'utente, la scheda può essere interrogata attraverso bus I2C per conoscere i parametri di

funzionamento del sistema (presenza o assenza della tensione principale, tensione della batteria, ecc...), attraverso un diodo led montato sulla scheda si ha anche una rappresentazione visuale dello stato di funzionamento del sistema (agganciato alla rete di alimentazione principale, in modalità UPS).

Oltre alle dotazioni legate alle funzionalità come UPS la scheda integra anche un modulo RTC (DS3231S) e relativa batteria (CR1025) connesso al bus I2C del Raspberry ed un driver RS232 (MAX3232) connesso direttamente alle linee RX-TX del Raspberry.

Il microcontrollore presente sulla scheda sovrintende la gestione della sezione di carica della batteria e la connessione/disconnessione della linea di alimentazione del Raspberry attraverso uno stadio a MOSFET, il microcontrollore comunica e controlla il corretto funzionamento del sistema di ricarica intelligente; la tensione di batteria è poi innalzata al livello della tensione di alimentazione del Raspberry mediante uno stadio switching DC-DC in topologia boost che garantisce in uscita una tensione stabilizzata di 5.1V, indipendentemente dal valore assunto dalla tensione della batteria.

Il compito del microcontrollore consiste nel configurare tutti i parametri di funzionamento della sezione di carica e gestire gli stati di funzionamento della scheda in base alla logica definita attraverso l'impostazione di un jumper passo 2.54 mm presente su un connettore apposito.

I parametri di funzionamento e configurazione della scheda sono gestiti dal microcontrollore attraverso il bus I2C del Raspberry, la scheda può essere interrogata per conoscere informazioni circa la presenza o assenza della tensione di alimentazione principale, la tensione della batteria, ecc... Il microcontrollore permette anche all'utente di memorizzare dei dati su una parte della memoria EEPROM interna offrendo così un'ulteriore possibilità per le applicazioni utente (ad esempio si possono salvare parametri di configurazione che devono essere caricati all'inizio del *boot* del sistema e che non si vuole, per qualche motivo, che risiedano sulla memoria di massa principale del Raspberry).

A livello di risorse GPIO la scheda occupa una sola linea (pin 11 del connettore Raspberry corrispondente al GPIO17, questa linea identificata con RPIOFF ha in serie un resistore di limitazione della corrente da 1k), utilizzata per avvisare il Raspberry che il sistema a breve verrà spento. L'utilizzo di questa linea consente all'utente di salvare i dati delle applicazioni e permettere così uno spegnimento pulito del sistema, quindi senza perdita di dati ed escludendo il rischio di corrompere il contenuto della SD card conseguente ad un'interruzione improvvisa della tensione di alimentazione. Per intercettare lo stato di questa linea è possibile installare un servizio che avverta il sistema di questo evento oppure controllarla in polling direttamente all'interno dell'applicazione utente.

Tutti gli ingressi e le uscite della scheda possono essere utilizzati saldando direttamente sulle piazzole oppure montando dei connettori passo 2.54 mm (la scheda è fornibile con diversi allestimenti, per maggiori dettagli si prega di fare riferimento alla sezione COME ORDINARE presente al termine del presente manuale).



Prima di collegare la scheda ad un carico esterno leggere attentamente tutte le informazioni contenute nel presente manuale di utilizzo.



La scheda, nel normale funzionamento, può generare del calore, si deve garantire sempre un ricircolo di aria sufficiente a permetterne il raffreddamento.

CARATTERISTICHE TECNICHE SCHEDA ELETTRONICA

Tensione di uscita (alimentazione carico)	5.1V
Tensione di ingresso (valore nominale e intervallo permesso)	5V (3.9V ÷ 14V)
Massima corrente in ingresso	3A
Massima corrente in uscita	3A
Massima corrente di carica batteria	3A
Efficienza circuito di ricarica	>90%
Temperatura operativa	0° ÷ +70°C
Protezione contro l'inversione della polarità della tensione in ingresso	presente, protetto ESD
Protezione contro l'inversione della polarità della tensione di batteria	non presente
Protezione sovratemperatura da NTC batteria	presente
Corrente di carica impostata per la batteria predefinita	380mA
Modulo RTC, indirizzo I2C	0x68
Batteria modulo RTC	CR1025 Li/MnO2
Indirizzo I2C scheda UPS	0x08

CARATTERISTICHE TECNICHE BATTERIA LiPo FORNITA A CORREDO DEL MODULO

Capacità nominale	420mAh
Tensione di batteria nominale	3.7V
Tensione batteria di fine carica	4.2V
Corrente di carica massima	420mA (1C)
Corrente di scarica massima	6300 mA (15C)
Autonomia alla corrente di scarica massima	≥3.8' @ 15C
Sensore integrato di temperatura (NTC)	presente
Protezione tensione massima di carica (overcharge)	4.28 ± 0.025V
Protezione sovracorrente di carica / scarica	450mA / 6300mA
Protezione corto circuito	presente
Protezione contro scarica profonda	3.0V±0.1V
Connessione	Connettore JST (S3B-XH-A)

BATTERIA LiPo - SCHEMA CONNETTORE E CAVI BATTERIA – X1

Il connettore X1 della batteria (JST modello S3B-XH-A) si trova sul lato saldature della scheda.

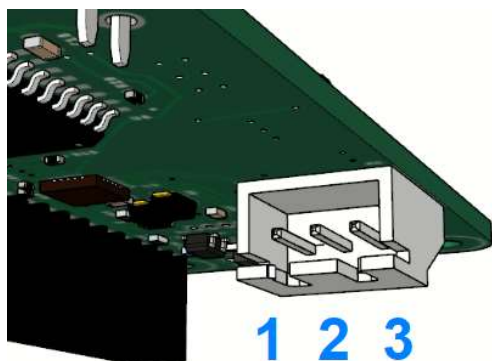


Figura 1: connettore batteria

PIN	SEGNALE	FUNZIONE
1	+VBAT	Positivo batteria
2	NTC	Termistore NTC
3	0V	Negativo batteria

Tabella 1: connettore batteria



Figura 2: assieme batteria inclusa con modulo UPS.

CAVO	SEGNALE	FUNZIONE
ROSSO	+VBAT	Positivo batteria
BIANCO	NTC	Termistore NTC
NERO	0V	Negativo batteria

Tabella 2: colore cavi batteria.

SCHEMA CONNETTORE DI ALIMENTAZIONE – J1

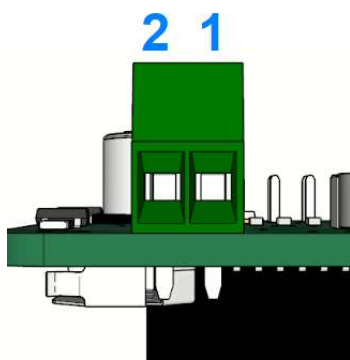


Figura 3: connettore di alimentazione/ricarica batteria.

PIN	SEGNALE	FUNZIONE	SEZIONE CAVI
1	0V	0V	Cavo 16 AWG
2	+VIN	5V ± 5% (3.9V ÷ 14V)	Cavo 16 AWG

Tabella 3: connettore alimentazione.

Coppia di serraggio massima delle viti del connettore: 0.15 Nm

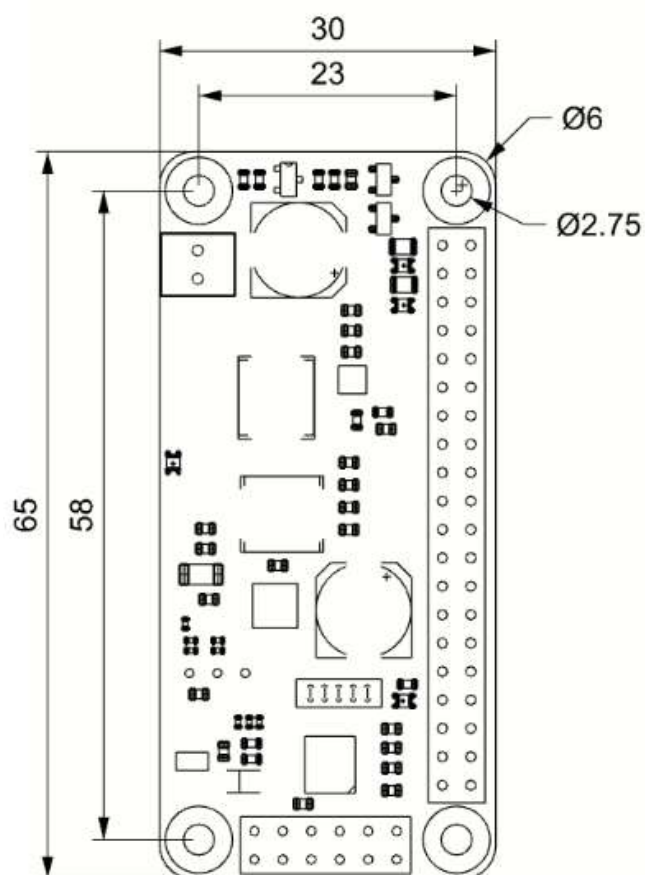


Figura 4: dimensioni meccaniche – vista in pianta da lato componenti.

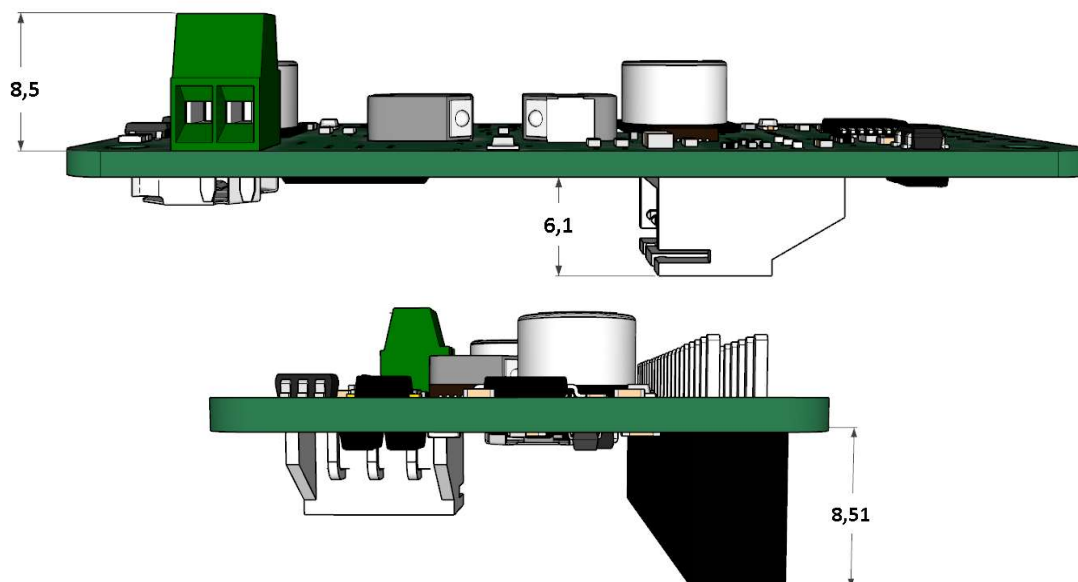


Figura 5: dimensioni meccaniche – ingombro connettori.

LINEE DI INGRESSO ED USCITA

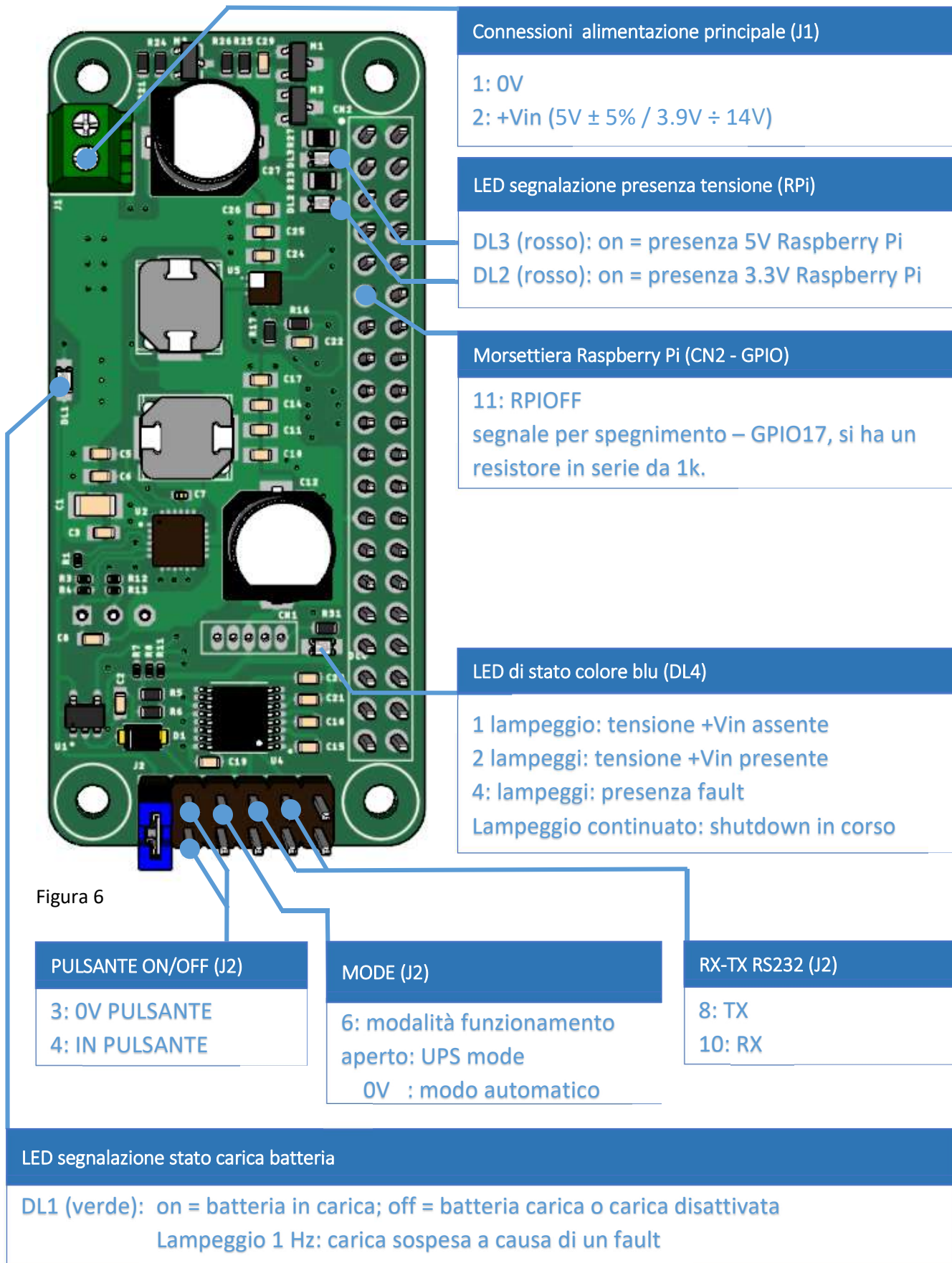


Figura 6

Il sistema può operare con due modalità distinte:

- Modalità automatica (UPS mode);
- Modalità manuale (desktop mode) – impostazione predefinita;

la scelta relativa alla modalità di funzionamento, da effettuarsi sempre a carico spento o non collegato, è definita dall'impostazione del livello di tensione presente sul pin 6 (MODE) del connettore utente J2; se lasciato aperto il sistema utilizza la modalità automatica, invece cortocircuitando il pin 6 con il pin 5 (0V) il sistema utilizza la modalità manuale. J2 è un header maschio da due righe con 6 vie ciascuna passo 2.54 mm è quindi possibile utilizzare un jumper passo 2.54 mm per cortocircuitare tra loro i due pin di configurazione.



La modifica della modalità di funzionamento deve essere effettuata sempre a carico non alimentato (led DL3 spento), si procede quindi con l'impostazione della modalità desiderata e si effettua il reset della scheda rimuovendo il ponticello tra i pin 1 e 2 e reinserendolo dopo alcuni secondi.

In alternativa la modalità si può modificare "on the fly" anche con sistema attivo, ma solo durante la fase di shutdown (durata 60s), al termine dello shutdown il carico viene spento ed il sistema entra automaticamente nella modalità scelta.

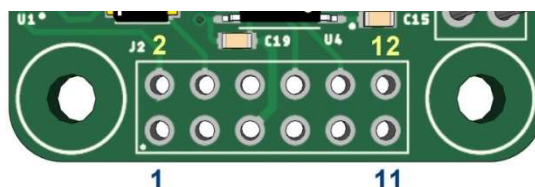


Figura 7: connettore utente J2.

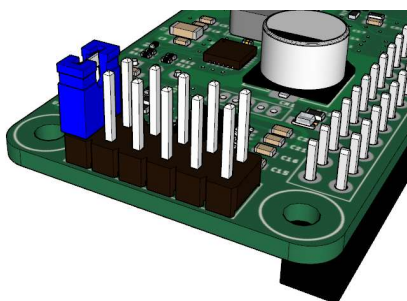


Figura 8: connettore J2 - Jumper JP1 per reset microcontrollore.

PIN	SEGNALE	FUNZIONE
1	VDD	Tensione 5V generale
2	VDDUP	Tensione sezione uP (5V)
3	0V	Connesso allo 0V
4	INBTN	Ingresso pulsante NO
5	0V	Connesso allo 0V
6	MODE	Selezione modo funzionamento
7	0V	Connesso allo 0V
8	TX	TX RPi
9	0V	Connesso allo 0V
10	RX	RX RPi
11	0V	Connesso allo 0V
12	-	Non collegato

Tabella 4: connettore utente J2.

Modalità UPS (accensione e spegnimento automatico in base alla tensione di alimentazione)

Si imposta lasciando aperto il pin 6 del connettore J2:

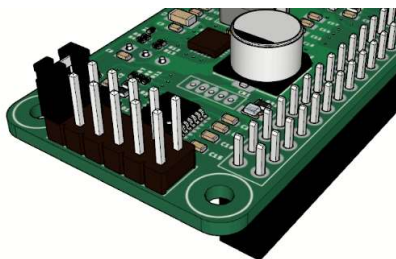


Figura 9: configurazione della modalità automatica (UPS).

Permette di gestire in modo automatico la linea di alimentazione del carico proprio come accade con un UPS tradizionale. In presenza della tensione di alimentazione principale, la scheda attiva automaticamente il processo di ricarica della batteria e quando la tensione di quest'ultima raggiunge i 3.9 V provvede ad alimentare il carico. Qualora la tensione principale venga a mancare il sistema entra nella modalità di backup, se la tensione di batteria diventa minore o uguale a 3.6V si ha l'attivazione della linea di shutdown RPIOFF (attiva alta), trascorsi ulteriori 60 s il sistema procede con la disconnessione del carico. Il carico verrà nuovamente alimentato solo al ritorno dell'alimentazione principale e con una tensione della batteria di backup maggiore o uguale a 3.9V.

È possibile interrompere questa modalità di funzionamento cortocircuitando il pin 3 con il pin 4 di J2, in questo modo si avvia la sequenza di spegnimento ed è possibile in questo intervallo di tempo mantenere aperto il pin 6 del connettore J2 (si ritorna automaticamente alla modalità automatica) oppure cortocircuitarlo con il pin 7 (0V) per attivare, trascorsi i 60s dello shutdown, la modalità di funzionamento manuale.

Modalità manuale (accensione e spegnimento attraverso pulsante esterno)

Si configura inserendo un jumper passo 2.54 mm nella terza posizione del connettore J2 (il pad 5 ed il pad 6 sono cortocircuitati).

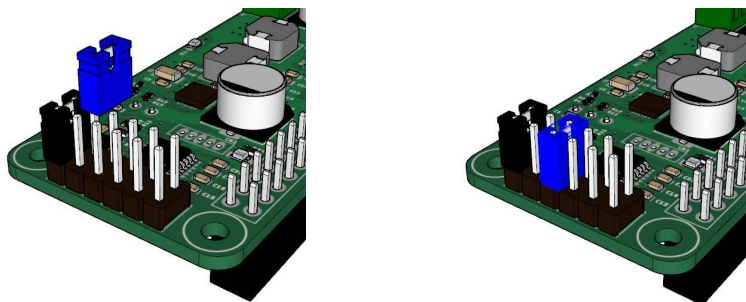


Figura 10: configurazione della modalità manuale.

Questa modalità permette di accendere e spegnere il Raspberry Pi, oppure una generica scheda embedded o più in generale un carico generico, in modo controllato dall'utente, attraverso un pulsante esterno, normalmente aperto oppure un contatto pulito, tra il pin 3 ed il pin 4 del connettore J2.

Il microcontrollore presente sulla scheda sovrintende le operazioni di accensione e spegnimento e controlla costantemente il valore della tensione di batteria. Il diodo led DL3 di colore rosso si accende segnalando la presenza della tensione di alimentazione sui pin 2 e pin 4 del connettore CN2. Nel caso in cui il carico sia costituito da un Raspberry dovrà accendersi anche il diodo led DL2 indicando la presenza della tensione 3.3V generata dal Raspberry stesso.

L'accensione del carico si effettua connettendo tra loro il pin 4 ed il pin 3 (0V) del connettore J2 per un intervallo di tempo di almeno 3 s (la tensione di batteria deve essere comunque maggiore o uguale a 3.9V).

Lo spegnimento si effettua in modo analogo connettendo tra loro il pin 4 ed il pin 3 (0V) del connettore J2 per un intervallo di tempo di almeno 5 s. Non appena la scheda entra nella fase di spegnimento viene attivata la linea RPIOFF che può essere utilizzata, ad esempio nel caso in cui il sistema alimenti una scheda Raspberry Pi, per avviare il processo di shutdown evitando così una possibile perdita di dati legata ad una interruzione improvvisa dell'alimentazione, trascorsi 60s la scheda interrompe l'alimentazione al carico rimanendo in stand-by.

Se la tensione di batteria dovesse diventare inferiore a 3.6V, indipendentemente dallo stato del pulsante di accensione/spegnimento, la scheda avvia automaticamente la procedura di shutdown attivando la linea RPIOFF, in seguito attende 60s ed al termine provvede a sezionare l'alimentazione del carico rimuovendo la tensione di alimentazione dai pin 2 e 4 del connettore CN2 (relativo al connettore GPIO del Raspberry), infine il segnale RPIOFF è riportato al livello basso ed il sistema ritorna nello stato di attesa fino a quando la tensione di batteria non risulta maggiore di 3.9V, da questo momento in poi è possibile riaccendere il carico utilizzando il pulsante.

Caratteristiche comuni per entrambe le modalità di funzionamento

In entrambe le modalità di funzionamento la batteria LiPo è sempre al sicuro, il sistema disattiva automaticamente il carico quando la tensione risulta inferiore a 3.6 V avviando la procedura di spegnimento proprio come se questa fosse stata richiesta manualmente dalla linea di ingresso di accensione/spegnimento, in questo modo il carico, se necessario, può effettuare uno shutdown in sicurezza, senza perdita di dati.

MESSA IN SERVIZIO – PASSO 1: modulo RTC e batteria CR1025 – installazione e configurazione

Il modulo è dotato di RTC integrato e relativa batteria tampone (CR1025), la batteria si installa sul lato saldature del circuito stampato nell'apposito alloggiamento.



È importante inserire la batteria in modo tale che il polo negativo sia a contatto con il circuito stampato ed il polo positivo a contatto con l'alloggiamento metallico, inserire la batteria a rovescio può danneggiare irrimediabilmente il modulo RTC.

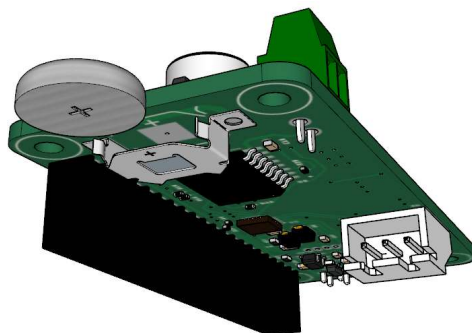


Figura 11: inserimento della batteria tampone per il modulo RTC (vista dal Lato Saldature).

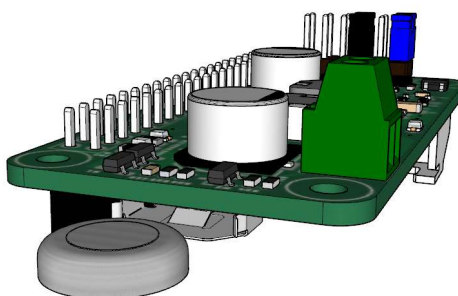


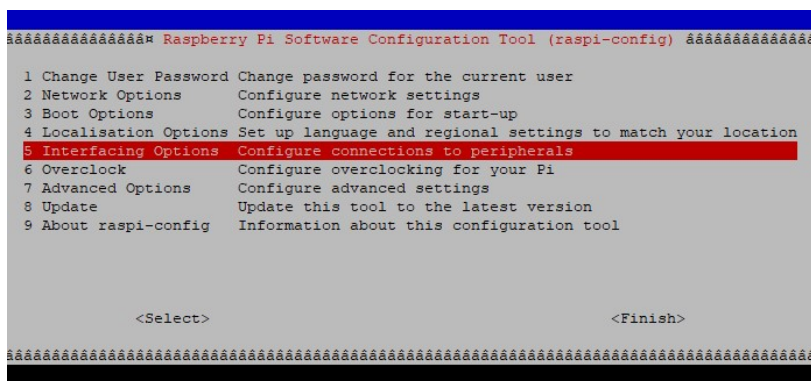
Figura 12: inserimento della batteria tampone per il modulo RTC (vista dal Lato Componenti).

Il passo successivo prevede la configurazione di alcune caratteristiche del Raspberry e l'installazione di tool specifici che potranno essere utilizzati per la lettura del modulo RTC.

1. A Raspberry spento, se non si dispone di una connessione di rete, collegare un monitor HDMI ed una tastiera su una porta USB libera. Se sul Raspberry è attivo il servizio SSH ed è disponibile una connessione di rete ci si potrà connettere al Raspberry da un qualsiasi PC dotato di terminale PuTTY¹ mediante protocollo SSH, in questo caso non è quindi necessario collegare direttamente al Raspberry monitor e tastiera. Si inserisce nell'apposita sede una microSDHC con il sistema operativo desiderato precaricato.
2. Verificare sulla scheda UPS che la modalità di funzionamento sia quella manuale (configurazione predefinita) vedere fig. 10, in seguito inserire la scheda UPS nel connettore GPIO del Raspberry Pi.
3. Collegare la batteria LiPo alla scheda attraverso l'apposito connettore X1 presente sul lato saldature della scheda stessa, riferimento figura 1.
4. Inserire la scheda nel connettore GPIO del Raspberry.
5. Inserire il jumper di reset del microcontrollore nel connettore J2 come riportato nella figura 8, il diodo led DL4 potrebbe iniziare a lampeggiare.
6. Si applica al connettore J1 la tensione di alimentazione principale, si attende che la batteria abbia completato il ciclo di carica (il led DL1 da verde fisso si spegne), attraverso un pulsante normalmente aperto connesso su J2 tra il pin 4 ed il pin 3 (0V) tenendolo premuto per almeno 3s, si attiva la linea di accensione, il Raspberry Pi deve accendersi. I led di stato DL2 e DL3 (entrambi di colore rosso) devono essere accesi e fissi, indicando la presenza della tensione a 5V e 3.3V sul connettore GPIO del Raspberry.
7. Completato il *boot*, si effettua il login con credenziali root oppure come utente normale; in quest'ultimo caso si dovrà ricordare di premettere ai vari comandi riportati di seguito il comando `sudo` per permetterne l'esecuzione con privilegi di amministratore.
8. Si verifica se è stato attivato nel sistema in uso l'interfaccia I2C attraverso l'utility di configurazione del Raspberry *raspi-config* scrivendo nel terminale il seguente comando seguito dalla pressione del tasto *Invio*:

```
sudo raspi-config
```

Utilizzando i tasti freccia ci si posiziona sulla riga *Interfacing Options* e si preme il tasto invio:



```

##### Raspberry Pi Software Configuration Tool (raspi-config) #####
1 Change User Password Change password for the current user
2 Network Options       Configure network settings
3 Boot Options          Configure options for start-up
4 Localisation Options  Set up language and regional settings to match your location
5 Interfacing Options   Configure connections to peripherals
6 Overclock             Configure overclocking for your Pi
7 Advanced Options      Configure advanced settings
8 Update                Update this tool to the latest version
9 About raspi-config    Information about this configuration tool

<Select>                                     <Finish>
#####
```

¹ <https://www.chiark.greenend.org.uk/~sgtatham/putty/>

Le immagini si riferiscono ad un sistema Raspberry Pi 4 model B rev 1.1. con sistema operativo *buster* e kernel Linux 4.19.75-v71+², per sistemi operativi differenti le opzioni potrebbero trovarsi in altri punti del menù di configurazione, ma saranno comunque sempre presenti e facilmente identificabili.

Utilizzando i tasti freccia si ricerca l'opzione I2C e si preme *Invio*.

```

##### Raspberry Pi Software Configuration Tool (raspi-config) #####
P1 Camera      Enable/Disable connection to the Raspberry Pi Camera
P2 SSH         Enable/Disable remote command line access to your Pi using SSH
P3 VNC         Enable/Disable graphical remote access to your Pi using RealVNC
P4 SPI         Enable/Disable automatic loading of SPI kernel module
P5 I2C         Enable/Disable automatic loading of I2C kernel module
P6 Serial      Enable/Disable shell and kernel messages on the serial connection
P7 1-Wire      Enable/Disable one-wire interface
P8 Remote GPIO Enable/Disable remote access to GPIO pins

<Select>                                <Back>
#####
```

Sempre mediante i tasti freccia si seleziona *Yes* e si preme il tasto *Invio*:

```

#####
Would you like the ARM I2C interface to be enabled?

<Yes>                                <No>
#####
```

Si preme nuovamente il tasto *Invio* per confermare la selezione.

```

#####
The ARM I2C interface is enabled

<Ok>
#####
```

² Comandi Linux utili per individuare informazioni sul sistema operativo sono i seguenti:

```
uname -a
cat /etc/os-release
```

Il sistema rappresenta la finestra iniziale, utilizzando il tasto *Tab* (tabulazione) ci si sposta sul pulsante *Finish* e si preme *Invio* per uscire dal programma di configurazione.

```

##### Raspberry Pi Software Configuration Tool (raspi-config) #####
1 Change User Password Change password for the current user
2 Network Options       Configure network settings
3 Boot Options          Configure options for start-up
4 Localisation Options  Set up language and regional settings to match your location
5 Interfacing Options   Configure connections to peripherals
6 Overclock             Configure overclocking for your Pi
7 Advanced Options      Configure advanced settings
8 Update                Update this tool to the latest version
9 About raspi-config    Information about this configuration tool

<Select>                                <Finish>
#####
```

9. Si procede con l'installazione del package "*i2c-tools*"³:

```
sudo apt-get update
sudo apt-get install i2c-tools
```

questo pacchetto permette di utilizzare una serie di strumenti utili per la gestione e la diagnostica del bus I2C del Raspberry.

10. Lo scopo del modulo RTC è di fornire data ed ora esatti anche in assenza di connessione verso un server NTP, è quindi importante operare in modo che la lettura della data e ora venga effettuata al boot del sistema, questo è possibile andando a modificare il file **/boot/config.txt**.

Partenza automatica con sistemi operativi basati su *systemd*

Nel caso delle versioni più recenti di sistema operativo (da Jessie in avanti – 25/04/2015, per versioni antecedenti vedere le note riportate al termine di questo paragrafo), utilizzando l'editor *vi*, si scriverà il seguente comando:

```
sudo vi /boot/config.txt
```

scorrendo fino alla fine del file si aggiunge la seguente riga:

```
dtoverlay=i2c-rtc,ds3231
```

A questo punto è possibile salvare le modifiche apportate al file ed uscire da *vi* premendo il tasto *:* e scrivendo *wq* seguito dalla pressione del tasto *Invio*.

Al prossimo riavvio della macchina l'ora sarà aggiornata automaticamente dal sistema leggendo dal modulo RTC.

Partenza automatica con sistemi operativi antecedenti a *systemd*

Nel caso di versioni del sistema operativo *wheezy* o antecedenti, quindi non basate su *systemd*, la procedura da utilizzare per automatizzare la lettura del modulo RTC alla partenza del sistema risulta differente. Nel caso di necessità è possibile contattare il produttore per maggiori informazioni.

³ https://i2c.wiki.kernel.org/index.php/I2C_Tools

11. Si disabilita il modulo `fake hwclock`⁴ (utilizzato dal kernel per simulare il comportamento di un RTC reale nel caso in cui il sistema non sia dotato di una connessione di rete che permetta di avere data ed ora sempre aggiornate) per evitare possibili interferenze con il modulo `hwclock`:

```
sudo apt-get -y remove fake-hwclock
```

Si disattiva anche l'avvio automatico del servizio al boot del sistema, nel caso di sistemi operativi più datati basati ancora su SysV si utilizza il seguente comando:

```
sudo update-rc.d -f fake-hwclock remove
```

Per i sistemi più recenti basati invece su *systemd* (disponibile a partire dalla versione *Jessie*) si utilizza invece il seguente comando:

```
sudo systemctl disable fake-hwclock
```

12. Utilizzando il proprio editor preferito (in questo esempio sarà utilizzato `vi`) aprire il file `/lib/udev/hwclock-set` quindi:

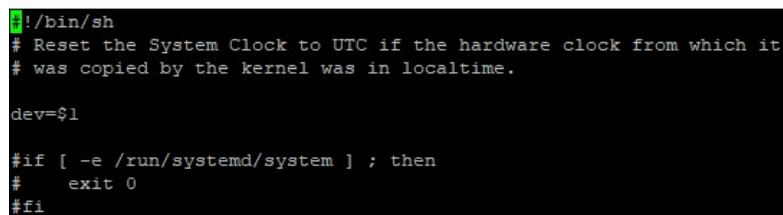
```
sudo vi /lib/udev/hwclock-set
```

scorrendo con i tasti freccia verso il basso si cercano nel file le seguenti righe:

```
if [ -e /run/systemd/system ] ; then
exit 0
fi
```

commentandole inserendo il carattere `#` ad ogni inizio riga, ottenendo quindi:

```
#if [ -e /run/systemd/system ] ; then
#exit 0
#fi
```

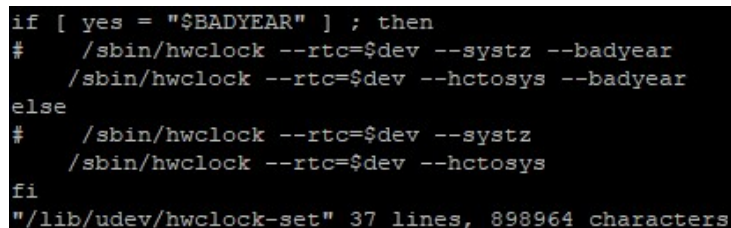


```
#!/bin/sh
# Reset the System Clock to UTC if the hardware clock from which it
# was copied by the kernel was in localtime.

dev=$1

#if [ -e /run/systemd/system ] ; then
#   exit 0
#fi
```

Scorrendo ancora verso la fine del file si commentano le righe come riportato nella figura seguente (per maggiori informazioni vedere il manuale relativo allo strumento *hwclock*⁵):



```
if [ yes = "$BADYEAR" ] ; then
#   /sbin/hwclock --rtc=$dev --systz --badyear
#   /sbin/hwclock --rtc=$dev --hctosys --badyear
else
#   /sbin/hwclock --rtc=$dev --systz
#   /sbin/hwclock --rtc=$dev --hctosys
fi
"/lib/udev/hwclock-set" 37 lines, 898964 characters
```

Si procede salvando in modo permanente le modifiche apportate al file ed uscendo da `vi` premendo il tasto `:` e scrivendo `wq` seguito dalla pressione del tasto *Invi*o.

⁴ <https://packages.debian.org/it/sid/fake-hwclock>

⁵ <https://man7.org/linux/man-pages/man8/hwclock.8.html>

13. Si riavvia il Raspberry con il seguente comando:

```
sudo reboot
```

seguito dalla pressione del tasto *Invio*.

14. Si effettua nuovamente il login e si verifica se il bus I2C risulta effettivamente abilitato dal kernel utilizzando il comando:

```
ls /dev/i2c*
```

```
root@raspberrypi:~# ls /dev/i2c*  
/dev/i2c-1  
root@raspberrypi:~#
```

Il bus I2C risulta correttamente configurato.

15. Per verificare la comunicazione con il modulo RTC si scrive nella finestra del terminale il seguente comando seguito dalla pressione del tasto *Invio*:

```
sudo i2cdetect -y 1
```

il comando attiva la scansione del bus I2C del Raspberry (identificato con il numero 1) procedendo immediatamente con le operazioni senza attendere ulteriore conferma da parte dell'utente (opzione *-y*), al termine della scansione si dovrà verificare nella risposta la sequenza di caratteri UU in corrispondenza dell'indirizzo 0x68, questo significa che il modulo RTC è stato installato correttamente (l'indirizzo 0x08 è invece relativo al microcontrollore presente sulla scheda UPS).

```
root@raspberrypi:~# sudo i2cdetect -y 1  
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  
00:  --  --  --  --  --  --  --  08  --  --  --  --  --  --  
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  
60:  --  --  --  --  --  --  --  UU  --  --  --  --  --  --  
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  
root@raspberrypi:~#
```

16. Il modulo RTC risulta quindi correttamente installato e raggiungibile attraverso il bus I2C del Raspberry, a questo punto si deve verificare l'effettiva capacità di memorizzazione della data ed ora corrente. Con il Raspberry collegato alla rete si verifica utilizzando il comando `date` che le impostazioni di data ed ora risultino effettivamente corrette:

```
root@raspberrypi:~# date  
Wed 28 Apr 10:23:17 CEST 2021  
root@raspberrypi:~#
```

Nel caso in cui non si disponga di una connessione di rete e la data non risulti corretta è possibile impostarla manualmente con il seguente comando (la data specificata dovrà ovviamente essere impostata in base all'effettiva data corrente ed al fuso orario utilizzato UTC oppure CEST):

```
sudo date -s "Wed Apr 28 10:27:00 CEST 2021"
```

Controllata la data si procede con lo scrivere nel modulo RTC la data ed ora corrente utilizzando il comando:

```
sudo hwclock -w
```

in seguito si rilegge la data memorizzata nel modulo RTC con il comando:

```
sudo hwclock -r
```

La data e l'ora devono corrispondere alla data e ora restituita attraverso l'utilizzo del comando `date`, potrebbero esserci alcuni secondi di differenza associati alla tempistica di esecuzione dei comandi.

Qualora ci fossero dei messaggi di errore conseguenti all'esecuzione dei comandi sopra riportati è probabile che ci siano dei problemi hardware con il modulo RTC, in questo caso si suggerisce di utilizzare una scheda differente e ripetere le operazioni sopra elencate.

```
root@raspberrypi:~# date
Wed 28 Apr 10:27:14 CEST 2021
root@raspberrypi:~# sudo hwclock -w
root@raspberrypi:~# sudo hwclock -r
2021-04-28 10:27:23.958208+02:00
root@raspberrypi:~# date
Wed 28 Apr 10:29:54 CEST 2021
root@raspberrypi:~# sudo hwclock -r
2021-04-28 10:29:56.346928+02:00
root@raspberrypi:~#
```

MESSA IN SERVIZIO – PASSO 2: verifica funzionalità connessione I2C verso il microcontrollore

Come già anticipato la scheda è equipaggiata con un microcontrollore, montato sul alto saldature della scheda stessa che, oltre alla gestione del funzionamento generale del sistema, espone all'utente una serie di funzionalità che possono essere utilizzate per controllare lo stato del sistema stesso oltre che altre caratteristiche illustrate nel seguito dell'esposizione.

Per controllare la raggiungibilità e quindi il corretto funzionamento della comunicazione sul bus I2C, si può seguire quanto riportato al passo precedente nel punto 15, l'indirizzo del microcontrollore corrisponde a 0x08.

ORGANIZZAZIONE REGISTRI PER INTERFACCIAMENTO SU BUS I2C

La scheda si interfaccia su bus I2C esponendo all'utente una serie di registri, l'accesso può essere effettuato dal bus I2C del Raspberry oppure da qualsiasi altro sistema che sia in grado di comunicare su bus I2C, nel proseguo dell'esposizione saranno presentati esempi di comunicazione specifici per l'utilizzo con sistemi Raspberry, ma i concetti esposti sono estendibili a qualsiasi altro sistema con supporto I2C.

Come anticipato, oltre ad una serie di registri utilizzati per la gestione delle funzionalità più specifiche della scheda (presenza o meno e valore della tensione di alimentazione principale, tensione della batteria LiPo, fase della carica della batteria LiPo, ecc...), sono previsti ulteriori 16 registri, ciascuno avente dimensione di 1 byte, mappati direttamente sulla memoria EEPROM interna al microcontrollore, per utilizzo diretto da parte dell'utente, con accesso completo in lettura e scrittura. Poiché questi registri sono effettivamente contenuti in una memoria EEPROM, la memorizzazione dei dati in essi contenuti sarà di tipo persistente, quindi indipendente dal fatto che il Raspberry risulti acceso oppure spento.

Possibili esempi di utilizzo dell'area EEPROM possono essere legati alla memorizzazione di un codice seriale legato al Raspberry attualmente in uso evitando così che la scheda possa essere spostata su un altro dispositivo; per memorizzare informazioni di configurazione che per qualche motivo non si vogliono scrivere sulla microSDHC del Raspberry e che sono utilizzate per configurare alla prima accensione delle

applicazioni presenti sul Raspberry stesso, oppure, più in generale, per qualsiasi altra tipologia di applicazione in cui sia necessario memorizzare dei dati.

0x55	BTCHGSTAT
0x54	VINSTATE
0x53	FAULT_PREV
0x52	FAULT_CURR
0x51	FAULT_RST
0x50	FAULT_FLAG
0x4F	
...	
0x43	RISERVATA
0x42	IBCHG
0x41	VIN
0x40	
0x3F	RISERVATA
0x3E	BATTV
0x3D	
...	
0x35	RISERVATA
0x34	IBCHGMAX
0x33	
...	
0x30	
0x2F	RISERVATA
0x2E	Revisione firmware — — — — —
0x2D	Versione firmware
0x2C	
...	
0x12	RISERVATA
0x11	Numero accensioni lo-byte — — — —
0x10	Numero accensioni hi-byte
0x0F	
...	
0x00	EEPROM UTENTE

Figura 13: organizzazione dei registri in memoria.

Si riportano di seguito i registri esposti dal microcontrollore verso il mondo esterno con il relativo indirizzo, funzione ed eventuali parametri se richiesti.

Indirizzo	Nome	Accesso	Funzione
0x00 ÷ 0x0F	EE_USER	R/W	Area utente EEPROM interna
0x10	POWERON_HI	R	Numero reset microcontrollore scheda HI-BYTE
0x11	POWERON_LO	R	Numero reset microcontrollore scheda LO-BYTE
0x2E	FW_REV	R	Revisione del firmware
0x2D	FW_VER	R	Versione del firmware
0x34	IBCHGMAX	R	Corrente massima impostata per la carica della batteria LiPo
0x3E	BATTV	R	Tensione della batteria LiPo
0x41	VIN	R	Tensione di ingresso (presenza/assenza e valore)
0x42	IBCHG	R	Corrente di carica della batteria LiPo
0x50	FAULT_FLAG	R	Flag segnalazione FAULT
0x51	FAULT_RST	R	Azzeramento del flag di FAULT
0x52	FAULT_CURR	R	Legge lo stato corrente del registro di FAULT
0x53	FAULT_PREV	R	Legge lo stato del registro di FAULT precedente all'ultimo evento di FAULT.
0x54	VINSTATE	R	Stato tensione di ingresso (presente/assente)
0x55	BTCHGSTAT	R	Fase carica batteria LiPo

Tabella 5: descrizione registri.

Legenda modalità di accesso:

R = lettura

W = scrittura

R/W = lettura e scrittura

La scrittura in un'area non permessa oppure di sola lettura non eseguirà l'operazione richiesta ed il contenuto della cella indirizzata non sarà quindi alterato; per la lettura su aree riservate potrebbero essere restituiti dei valori che non riflettono il reale contenuto della cella indirizzata.

REGISTRI EEPROM (0x00 ÷ 0x0F)

Per i registri EEPROM il valore scritto oppure letto è quello specificato senza necessità di ulteriori elaborazioni o interpretazioni del dato da scrivere o ricevuto dopo una lettura.

REGISTRO POWER ON (0x10 ÷ 0x11)

Contiene il numero di volte che il microcontrollore si è riavviato (numero di volte che il microcontrollore è stato riavviato a causa o di un reset software oppure per un'interruzione della tensione di alimentazione della sezione del microcontrollore) ed è espresso su due byte, il valore di questo contatore si ottiene con la seguente formula:

$$Power\ on = 256 * HIBYTE + LOBYTE$$

in cui:

HIBYTE: valore restituito dalla lettura del registro 0x10 dopo conversione in decimale

LOBYTE: valore restituito dalla lettura del registro 0x11 dopo conversione in decimale

REGISTRO VERSIONE FIRMWARE FW_VER (0x2D)

Contiene la versione corrente del firmware installato sul microcontrollore a bordo della scheda.

REGISTRO REVISIONE FIRMWARE FW_REV (0x2E)

Contiene la revisione della versione corrente del firmware installato sul microcontrollore a bordo della scheda.

REGISTRO IBCHGMAX (0x34)

Contiene le impostazioni predefinite della scheda per la corrente di carica massima relativa alla batteria LiPo, il valore di corrente, espresso in mA, si ricava sommando tra di loro le corrispondenze tra i vari bit del valore restituito dalla lettura del registro ed il peso relativo espresso in mA.

Bit	Contributo di corrente (mA)
6 (MSB)	4096
5	2048
4	1024
3	512
2	256
1	128
0 (LSB)	64

Tabella decodifica bit impostazione corrente carica massima batteria LiPo

Il valore impostato per la batteria fornita a corredo del sistema è uguale a 0x06 (espresso in esadecimale) oppure 0000 0110 esprimendolo in base 2 (numero binario), applicando la regola descritta in precedenza la corrente finale sarà uguale a:

$$I_{BATTCHGMAX} = 0 * 4096 + 0 * 2048 + 0 * 1024 + 0 * 512 + 1 * 256 + 1 * 128 + 0 * 64 = 384\ mA$$

REGISTRO BATTV (0x3E)

Contiene il valore della tensione relativo alla batteria LiPo. Il valore di tensione, espresso in mV, si ricava sommando un offset fisso di 2304 mV con la somma delle corrispondenze tra i vari bit del valore restituito dalla lettura del registro ed il peso relativo espresso in mV, l'aggiornamento di questo registro avviene 1 volta al secondo.

Bit	Contributo di tensione (mV)
6 (MSB)	1280
5	640
4	320
3	160
2	80
1	40
0 (LSB)	20

Tabella decodifica bit per la tensione della batteria LiPo

Se la lettura del registro fornisce in risposta ad esempio 0x5E, ovvero 0101 1110 che decodificato fornisce:

$$V_{BATT} = 2304 + 1 * 1280 + 0 * 640 + 1 * 320 + 1 * 160 + 1 * 80 + 1 * 40 + 0 * 20 = 4184 \text{ mV}$$

ovvero una tensione rilevata di 4.184 V, la tensione misurata mediante multimetro direttamente sui pin del connettore della batteria risulta uguale a 4.173 V.

L'errore assoluto massimo tra il valore letto attraverso il registro e la misura reale è sempre minore o uguale a 20 mV.

Permette di sapere se in ingresso è presente o meno la tensione di alimentazione principale ed in caso affermativo ne restituisce il valore.

La struttura dei bit di questo registro è riportata di seguito:

Bit	Significato
7	Stato tensione di alimentazione principale 0: assente 1: presente
[6 ÷ 0]	Tensione espressa come somma di contributi (vedere la tabella seguente)

Tabella significato bit registro VIN

I bit da 0 (LSB) a 6 (MSB) consentono di determinare il valore della tensione di ingresso (tensione applicata al morsetto J1). Il valore, espresso in mV, si ottiene considerando un offset fisso di 2600 mV al quale si sommano i contributi dei vari bit pesati in mV secondo la tabella seguente; questo registro è aggiornato automaticamente 1 volta al secondo.

Bit	Contributo di tensione (mV)
6 (MSB)	6400
5	3200
4	1600
3	800
2	400
1	200
0 (LSB)	100

Tabella decodifica bit per la tensione di ingresso

A titolo di esempio, supponiamo che la lettura del registro fornisca in risposta, il valore 0x97, ovvero 1001 0111. In base al significato dei singoli bit si deduce che la tensione in ingresso è presente (bit 7 di valore 1); il valore della tensione sarà dato dalla decodifica, secondo la tabella sopra riportata, del numero binario 001 0111 (ottenuto escludendo il bit 7):

$$V_{VIN} = 2600 + 0 * 6400 + 0 * 3200 + 1 * 1600 + 0 * 800 + 1 * 400 + 1 * 200 + 1 * 100 = 4900 \text{ mV}$$

ovvero 4.9 V, per questo caso specifico, la tensione misurata direttamente sui pin del connettore della batteria utilizzando un multimetro digitale, si avevano 4.862 V.

L'errore assoluto massimo tra il valore letto attraverso il registro e la misura reale è sempre minore o uguale a 100 mV.

Se si desidera conoscere solo lo stato della tensione di alimentazione principale, quindi se presente oppure non presente, è possibile utilizzare il registro VINSTA (indirizzo 0x54).

REGISTRO IBCHG (0x42)

Se la batteria si trova nella fase di carica restituisce il valore della corrente di carica, nel caso in cui la tensione di batteria risulti inferiore a 2V la lettura di questo registro fornirà sempre 0.

Il valore della corrente, espresso in mA, si ricava sommando tra loro le corrispondenze tra i vari bit del valore restituito dalla lettura del registro ed il peso relativo espresso in mA. L'aggiornamento di questo registro è effettuato periodicamente 1 volta al secondo.

Bit	Contributo di corrente (mA)
7 (MSB)	0
6	3200
5	1600
4	800
3	400
2	200
1	100
0 (LSB)	50

Tabella decodifica bit per la corrente attraverso la batteria nella fase di carica

Se la lettura del registro fornisce in risposta, ad esempio, un valore di 0x02 => 0000 0010, significa che la batteria viene caricata con una corrente il cui valore si calcola con la seguente espressione:

$$I_{BCHG} = 0 * 3200 + 0 * 1600 + 0 * 800 + 0 * 400 + 1 * 200 + 0 * 100 + 0 * 50 = 200 \text{ mA}$$

quindi una corrente di carica pari a circa 200 mA.

REGISTRO FAULT_FLAG (0x50)

Permette di sapere se si è verificato un evento di fault.

La struttura dei bit di questo registro è riportata di seguito:

Bit	Significato
[7 ÷ 0]	0: assenza nessun FAULT, funzionamento regolare 1: si è verificato un FAULT

Tabella registro FAULT_FLAG

In presenza di un fault si ha anche il lampeggio del diodo led DL1 ad una frequenza di circa 1 Hz.

Quando si rileva la presenza di un fault è possibile determinare quale fault ha provocato tale segnalazione.

Per determinare la sorgente del fault si procede come di seguito specificato:

- Si effettua la lettura del registro FAULT_PREV (indirizzo 0x53) e la si memorizza.
- Si effettua la lettura del registro FAULT_CURR (indirizzo 0x52) e la si memorizza.
- Si può azzerare la segnalazione del fault leggendo il registro FAULT_RST (indirizzo 0x51)

Dal confronto dei due registri FAULT_PREV e FAULT_CURR è possibile determinare quale è la causa della segnalazione di fault. Può capitare che alla prima messa in servizio il flag sia attivato, in questo caso è sufficiente provare a riavviare il microcontrollore (JP1) oppure azzerare il flag con la procedura sopra riportata.

REGISTRO FAULT_RST (0x51)

Permette di azzerare le segnalazioni contenute nel registro di FAULT. Per azzerare il flag di segnalazione presenza FAULT (registro FAULT_FLAG 0x50) è sufficiente effettuare una lettura del contenuto di questo registro.

Prima di azzerare la segnalazione di fault è opportuno andare a leggere il contenuto dei registri FAULT_PREV e FAULT_CURR per determinare quale è stata la causa della segnalazione.

REGISTRO FAULT_CURR (0x52)

Permette di leggere il contenuto corrente del registro di FAULT.

Bit	Significato
7	Non utilizzato
6	Non utilizzato
[5-4]	Stato fault associato alla carica della batteria 00: normale 01: fault in ingresso 10: fault per sovratemperatura (spegnimento) 11: fault carica oltre il tempo limite predefinito
3	Stato fault tensione batteria 0: normale 1: tensione batteria maggiore di 4,37 V
[2÷0]	Stato fault associato alla lettura termistore NTC batteria 000: normale 001: temperatura troppo bassa 010: temperatura troppo alta

Tabella significato bit registro FAULT_CURR.

REGISTRO FAULT_PREV (0x53)

Permette di leggere quale era il contenuto del registro di FAULT prima che si manifestasse un evento di fault.

Bit	Significato
7	Non utilizzato
6	Non utilizzato
[5-4]	Stato fault associato alla carica della batteria 00: normale 01: fault in ingresso 10: fault per sovratemperatura (spegnimento) 11: fault carica oltre il tempo limite predefinito
3	Stato fault tensione batteria 0: normale 1: tensione batteria maggiore di 4,37 V
[2÷0]	Stato fault associato alla lettura termistore NTC batteria 000: normale 001: temperatura troppo bassa 010: temperatura troppo alta

Tabella significato bit registro FAULT_PREV.

REGISTRO VINSTATE (0x54)

Informa se in ingresso è presente o meno la tensione di alimentazione principale.

Bit	Significato
[7 ÷ 6]	Non utilizzati
0	Stato tensione di alimentazione principale 0: assenza tensione alimentazione 1: presenza tensione di alimentazione

Tabella bit registro VINSTATE.

Nel caso in cui interessi conoscere anche il valore è possibile utilizzare il registro VIN (indirizzo 0x41).

REGISTRO BTCHGSTAT (0x55)

Permette di conoscere in quale fase di carica si trova la batteria LiPo, di seguito si riportano i possibili valori ottenuti dalla lettura di questo registro.

Valore	Significato
0	Batteria non in carica
1	Fase di precarica
2	Fase di carica rapida
3	Carica terminata

Tabella bit registro BTCHGSTAT.

Per conoscere il valore della corrente di carica si può leggere il contenuto del registro IBCHG (indirizzo 0x42).

LETTURA REGISTRO DA BUS I2C

Per la lettura di un registro attraverso la shell bash del Raspberry PI si può utilizzare il seguente comando (può essere eseguito anche senza privilegi di amministratore) seguito dalla pressione del tasto *Invio*:

```
i2cget -y 1 0x08 <indirizzo_registro_hex> b
```

ottenendo in risposta il valore, espresso in esadecimale, del valore letto all'indirizzo specificato.

Se si dovesse ottenere in risposta `Error: Read failed` questo può essere sintomo di un problema hardware nelle connessioni oppure nel caso in cui la lettura venga richiesta in corrispondenza di un reset del microcontrollore.

SCRITTURA REGISTRO DA BUS I2C

Per la scrittura di un registro attraverso la shell bash del Raspberry PI si utilizza il seguente comando (può essere eseguito anche senza privilegi di amministratore) seguito dalla pressione del tasto *Invio*:

```
i2cset -y 1 0x08 <indirizzo_registro_hex> <valore_da_scrivere_hex> b
```


ESEMPIO: lettura e scrittura su registro area utente memoria EEPROM

Si riporta un esempio di lettura, scrittura e successiva lettura per verifica della prima locazione di memoria dell'area utente sulla memoria EEPROM. Tutti i comandi sono seguiti dalla pressione del tasto *Invio*.

Si inizia con la lettura del contenuto della memoria all'indirizzo 0x00:

```
pi@raspberrypi:~ $ i2cget -y 1 0x08 0x00 b
0x00
```

Si prova a scrivere, nella stessa locazione, il valore 128 ovvero 0x80 in esadecimale:

```
pi@raspberrypi:~ $ i2cset -y 1 0x08 0x00 0x80 b
```

si rilegge il contenuto della locazione appena scritta:

```
pi@raspberrypi:~ $ i2cget -y 1 0x08 0x00 b
0x80
```

Se si ottiene in risposta 0x80 questo conferma che il dato era stato effettivamente memorizzato nella locazione 0 all'interno dell'area EEPROM utente; se invece si ottiene un valore differente o un messaggio di errore è opportuno ricontrollare le connessioni e se i comandi sono stati inseriti con la sintassi corretta.

```
pi@raspberrypi:~ $ i2cget -y 1 0x08 0x00 b
0x00
pi@raspberrypi:~ $ i2cset -y 1 0x08 0x00 0x80 b
pi@raspberrypi:~ $ i2cget -y 1 0x08 0x00 b
0x80
pi@raspberrypi:~ $
```

ESEMPIO: lettura e scrittura registro POWER ON numero accensioni microcontrollore

In questo caso si devono leggere due registri, il 0x10 per la parte alta ed il registro 0x11 per la parte bassa.

```
pi@raspberrypi:~ $ i2cget -y 1 0x08 0x10 b
0x00
pi@raspberrypi:~ $ i2cget -y 1 0x08 0x11 b
0x01
pi@raspberrypi:~ $
```

In questo caso il microcontrollore è stato riavviato $256 \cdot 0 + 1 = 1$ volta.

Verificato il funzionamento del sistema attraverso il terminale del Raspberry ci addentriamo più nella programmazione lato utente per vedere come poter implementare la lettura e la scrittura dei registri del microcontrollore direttamente da Python.

Prima di procedere è opportuno installare alcuni tool utilizzando il seguente comando fornito con privilegi di amministratore:

```
sudo apt-get install -y python-smbus python3-smbus python-dev python3-dev
```

L'opzione `-y` evita all'utente di dover rispondere Y ad ogni pacchetto che deve essere installato. Questi pacchetti permettono la gestione del bus I2C del Raspberry da script scritti per essere eseguiti con interprete Python 2.x e Python 3.x.

La libreria utilizzata per la gestione del bus I2C da script Python è la *smbus*, di seguito si riporta un esempio di script che illustra come utilizzare i metodi di lettura e scrittura sui registri del microcontrollore (che, si ricorda, dal punto di vista del Raspberry si comporta come *slave device*), il codice presentato non fa altro che incrementare di una unità, ogni 5s, il valore contenuto in un registro del microcontrollore, per fermare l'esecuzione dello script è sufficiente premere la combinazione di tasti CTRL+C.

Per eseguire lo script si scrive nella *shell* il seguente comando seguito dalla pressione del tasto *Invio*:

```
python esempio-i2c.py
```

Codice script esempio-i2c.py

```
#!/usr/bin/python
# -----
# Prerequisiti
# sudo apt-get install python-smbus python-dev
#
# Determinare numero bus I2C
# ls -l /dev/i2c-1
# crw-rw---- 1 root i2c 89, 1 Feb 14 2019 /dev/i2c-1
# -----

import os
import sys
import smbus    # Per gestione bus I2C
import time

UP_DEVICE_ADDRESS = 0x08    # Indirizzo microcontrollore
UP_REG_ADDRESS = 0x00       # Registro microcontrollore

# -----
#                               MAIN
# -----

if __name__ == "__main__":

    # Pulisce lo schermo
    os.system("clear")

    # BUS I2C-1 (vedere risposta al comando ls -l /dev/i2c-1)
    rpi_bus = smbus.SMBus(1)

    cycle_counter = 0
    while True:

        try:
            cycle_counter+=1;
```

```

print("-----")
sys.stdout.flush()
print("Ciclo lettura-incremento-scrittura-lettura numero: " \
      +str(cycle_counter))
sys.stdout.flush()
print("Premere CTR+C per uscire.")
sys.stdout.flush()
print("-----")
sys.stdout.flush()

# Esegue la lettura del registro specificato
byte_val = rpi_bus.read_byte_data(UP_DEVICE_ADDRESS, UP_REG_ADDRESS)
print(" Lettura registro  : " + str(UP_REG_ADDRESS) + \
      " - Valore letto: " + str(byte_val))
sys.stdout.flush()

time.sleep(1)

# Incremento di 1 il valore del dato letto
byte_val += 1
print("Incremento il valore letto di 1")
sys.stdout.flush()

# Esegue una scrittura singola sul registro specificato
print(" Scrittura Registro: " + str(UP_REG_ADDRESS) + \
      " - Valore scritto: " + str(byte_val))
sys.stdout.flush()
rpi_bus.write_byte_data(UP_DEVICE_ADDRESS, UP_REG_ADDRESS, byte_val)

time.sleep(1)

# Esegue una nuova lettura dello stesso registro
byte_val = rpi_bus.read_byte_data(UP_DEVICE_ADDRESS, UP_REG_ADDRESS)
print(" Lettura registro  : " + str(UP_REG_ADDRESS) + \
      " - Valore letto: " + str(byte_val))
sys.stdout.flush()

# Pausa prima del prossimo ciclo
time.sleep(5)

except (KeyboardInterrupt, SystemExit), ex:
    # Gestisce interruzione CTRL+C
    print("CTRL-C detected!")
    sys.stdout.flush()
    sys.exit(0)
except Exception as ex:
    # Gestione qualsiasi altra eccezione
    template = "ERROR: An exception of type {0} occurred." + \
              "Arguments:\n{1!r}"
    msg = template.format(type(ex).__name__, ex.args)
    print(msg)
    sys.stdout.flush()
    # Codice di uscita
    print("(main) Exiting from main program")
    sys.stdout.flush()
    sys.exit(0)

```

Per trasferire il file da un sistema Windows sul Raspberry attraverso una connessione di rete si può utilizzare ad esempio l'applicazione WinSCP⁶ che permette il trasferimento di file con protocollo SFTP (SSH File Transfer Protocol) attraverso un'interfaccia grafica anche con supporto *drag&drop*, inoltre, una volta che i file sono stati trasferiti, ne permette anche la modifica con un semplice click. Come editor Windows si

⁶ <https://winscp.net/eng/download.php>

suggerisce Notepad++⁷ in quanto salva i file in formato direttamente compatibile con sistemi operativi Linux come quello utilizzato sul Raspberry.

L'esempio riportato comprende anche altri aspetti generali sulla programmazione applicati più nello specifico al linguaggio Python come la gestione delle eccezioni, questo proprio solo per rendere più completo il codice ed eventualmente fornire qualche spunto in tal senso, di seguito invece ci si concentrerà prevalentemente su quelli che sono gli aspetti più di nostro interesse ovvero la comunicazione sul bus I2C.

Dal punto di vista della programmazione Python la prima cosa da fare consiste nel comunicare all'interprete Python che nel proseguo dello script dovranno essere utilizzati metodi specifici (relativi al bus I2C) contenuti in una certa libreria (*smbus*), a tale scopo si ha la seguente riga di codice:

```
import smbus
```

Per utilizzare questi metodi (che sono in sostanza delle funzioni) si deve istanziare un oggetto di tipo bus il cui codice si trova all'interno della libreria *smbus*.

Un'istanza, da un punto di vista pratico, è un oggetto fisico (quindi rappresentato da del codice che viene effettivamente caricato in memoria ed inizializzato) ricavato da un modello che è contenuto in una specifica libreria (in questo caso *smbus*) e che viene inizializzato proprio per essere utilizzato in seguito, in questo caso l'istanza del bus sarà creata con la seguente riga di codice:

```
rpi_bus = smbus.SMBus(1)
```

da notare come il nome assegnato *rpi_bus* sia completamente arbitrario, può quindi essere un qualsiasi nome assegnato in modo significativo atto a ricordare ciò che rappresenta realmente in quanto punterà e permetterà l'accesso all'oggetto vero e proprio presente nella memoria del sistema fino a quando il codice dello script sarà in esecuzione; ad ogni modo, qualunque esso sia, sarà comunque il riferimento con cui andremo ad identificare, all'interno del nostro script, il bus I2C fisico del Raspberry.

Come illustrato in precedenza, nel caso della gestione manuale del bus I2C attraverso la shell bash del Raspberry, si deve indicare su quale bus I2C si vuole lavorare, questo perché, in certi sistemi, è possibile che siano presenti più bus I2C, nel caso in esame il bus è unico ed è indirizzato con 1.

Per determinare il riferimento del bus si può utilizzare il seguente comando da shell:

```
ls -l /dev/i2c-*
```

```
root@raspberrypi:/home/Esempi/ds3231# ls -l /dev/i2c-*
crw-rw---- 1 root i2c 89, 1 Feb 14 2019 /dev/i2c-1
root@raspberrypi:/home/Esempi/ds3231#
```

in questo caso, trovando solo un unico riferimento, *i2c-1*, si utilizza appunto 1 come identificatore del bus.

Sul bus I2C si possono avere più dispositivi (detti *slave*) che potranno essere indirizzati dal Raspberry (che in questo caso ha il ruolo di *master* ovvero di coordinatore delle operazioni che saranno effettuate sul bus I2C), è quindi utile definire delle costanti relative all'indirizzo di ogni slave che, nel nostro caso, è costituito dal microcontrollore presente sulla scheda con indirizzo 0x08, quindi:

```
UP_DEVICE_ADDRESS = 0x08
```

⁷ <https://notepad-plus-plus.org/downloads/>

Per eseguire una lettura su un singolo registro si utilizza il metodo `read_byte_data` sull'istanza del bus che è stata identificata con il nome `rpi_bus`, in particolare:

```
byte_val = rpi_bus.read_byte_data(UP_DEVICE_ADDRESS, UP_REG_ADDRESS)
```

in cui:

`UP_DEVICE_ADDRESS` : indirizzo dello *slave device* (in questo caso il microcontrollore)

`UP_REG_ADDRESS` : indirizzo del registro che si desidera leggere (nell'esempio 0x00 in EEPROM)

`byte_val` : variabile che conterrà il valore restituito dalla lettura del registro `UP_REG_ADDRESS`

in modo del tutto analogo, per scrivere su un registro dello *slave device*, occorrerà fornire l'indirizzo dello slave sul bus I2C (quindi il microcontrollore), l'indirizzo del registro in cui si vuole scrivere (nell'esempio si scrive sull'area EEPROM all'indirizzo 0x00) ed ovviamente il dato da scrivere.

La sintassi da utilizzare è la seguente:

```
rpi_bus.write_byte_data(UP_DEVICE_ADDRESS, UP_REG_ADDRESS, byte_val)
```

in cui:

`UP_DEVICE_ADDRESS` : indirizzo dello *slave device* (in questo caso il microcontrollore)

`UP_REG_ADDRESS` : indirizzo del registro dello *slave device* in cui si vuole scrivere (nell'esempio 0x00 in EEPROM)

`byte_val` : variabile che conterrà il valore da scrivere nel registro `UP_REG_ADDRESS`

Questo, in sintesi, è quanto serve per poter gestire direttamente da uno script Python la lettura e la scrittura su bus I2C, ulteriori informazioni sono reperibili consultando l'help del pacchetto `smbus` e la relativa documentazione Python⁸.

⁸ <http://wiki.erazor-zone.de/wiki:linux:python:smbus:doc>

La linea RPIOFF è gestita dalla scheda per segnalare quando è previsto uno spegnimento imminente del sistema a causa di un black-out e della condizione di batteria quasi scarica, questa linea è agganciata direttamente al pin GPIO17 del connettore GPIO del Raspberry Pi ed è l'unico pin dedicato utilizzato dalla scheda.

Il tempo previsto tra la segnalazione dell'evento di spegnimento (linea RPIOFF) e l'effettivo spegnimento del Raspberry è impostato a 60 secondi.

L'autonomia della batteria tampone, quando questa risulta completamente carica ed in completa assenza della tensione di alimentazione primaria, risulta maggiore di 4' considerando anche una condizione operativa gravosa da parte del Raspberry che preveda un assorbimento in corrente dell'ordine dei 3 A.

La batteria fornita a supporto con la scheda è comunque in grado di erogare una corrente di 15C, corrispondente a circa 6300 mA, per un tempo minimo di circa 3.8 minuti (batteria nuova, completamente carica) il che garantisce ai normali livelli di utilizzo una operatività più che sufficiente per uno spegnimento controllato nei tempi indicati.

Per gestire in sicurezza lo shutdown è fondamentale intercettare il momento in cui la linea RPIOFF viene forzata al livello alto dal modulo UPS, questo si può fare con due approcci differenti:

- Controllare in polling (modo ciclico) dalla propria applicazione lo stato della linea RPIOFF e nel caso in cui si rilevi lo stato attivo provvedere allo shutdown del sistema;
- Installare un servizio che, indipendentemente dalle applicazioni che sono in esecuzione, lavorando in background avvisi tutto il sistema che deve essere eseguito uno shutdown;

entrambi i metodi hanno lo scopo di effettuare lo shutdown del Raspberry, ma si hanno importanti differenze. Nel primo (polling) è responsabilità dell'utente gestire in modo ciclico la linea RPIOFF, nel secondo è il sistema che si preoccupa di gestire la condizione di shutdown in modo indipendente dai programmi utente in esecuzione (servizio con esecuzione in background).

Benché si consiglia caldamente di utilizzare il secondo approccio, quello che prevede quindi un servizio apposito per il controllo e la gestione della linea RPIOFF, si illustrano di seguito entrambe le tecniche per una esposizione che sia, da un punto di vista didattico, la più completa possibile.

Questa soluzione è la più semplice possibile e prevede che nel programma utente sia presente un ciclo che controlli costantemente lo stato della linea RPIOFF con una risoluzione temporale di almeno 1 secondo, di meno sovraccaricherebbe l'esecuzione del programma e quindi del sistema, un tempo maggiore sovraccarica di meno sistema, ma deve comunque essere impostato in modo tale da lasciare, rispetto ai 60 secondi, un certo margine di sicurezza per permettere al sistema di spegnersi completamente in conseguenza del comando di shutdown.

Per eseguire lo script si scrive nella *shell* il seguente comando seguito dalla pressione del tasto *Invio*:

```
python ex-polling-GPIO17.py
```

Codice script ex-polling-GPIO17.py

```
#!/usr/bin/python

# -----
# Programma di esempio controllo stato GPIO17
# -----

import os
import sys
import time

import RPi.GPIO as GPIO

SHUTDOWN_ENABLED = False    # Flag esecuzione shutdown

# Pulisce lo schermo
os.system("clear")

# Disattiva warnings GPIO
GPIO.setwarnings(False)

# Imposta modalita' utilizzo GPIO
GPIO.setmode(GPIO.BCM)

# Definisce la linee GPIO17 come ingresso
GPIO.setup(17, GPIO.IN) # GPIO17

print("Start GPIO test")
print("hit CTRL+C to exit")
print("-----")
sys.stdout.flush()

make_shutdown = False

try:
    timectr=0
    while (not make_shutdown):

        print("Time: " + str(timectr) + "s")
        sys.stdout.flush()
        time.sleep(1)
        timectr = timectr + 1

        if (GPIO.input(17) == 1):
            print("RPIOFF: line activated, 60 seconds to shutdown!!")
            sys.stdout.flush()
            make_shutdown = True
```

```

        else:
            print("RPIOFF: line is not active - normal behavior")
            sys.stdout.flush()

    print("Shutdown mode activated!")
    sys.stdout.flush()
    # Comando di shutdown
    if (SHUTDOWN_ENABLED):
        subprocess.call(["shutdown", "-h", "now"])

except KeyboardInterrupt:
    print "Program termination detected!"
except SystemExit:
    print "System exit detected!"
except:
    print("Error detected!")
finally:
    print("Program exit.")
    sys.stdout.flush()
    sys.exit(0)

```

il codice riportato è sufficientemente semplice, unica nota è l'utilizzo del modulo `subprocess` per invocare il comando di `shutdown` proprio come se lo si fosse scritto direttamente nella shell.

Il flag `SHUTDOWN_ENABLED` è stato inserito solo per comodità, per un utilizzo reale deve essere impostato su `True` oppure semplicemente eliminato; lasciandolo su `False` si può simulare il comportamento dello script, senza eseguire realmente lo shutdown del sistema, semplicemente forzando al livello logico alto il pin GPIO17 (portando quindi sul pin GPIO17 una tensione non superiore a 3.3V).

Il programma utente dovrebbe quindi incorporare la parte di codice che controlla lo stato della linea RPIOFF e lavorare in modo da essere non bloccante per garantire che non vi siano punti del codice che impediscano la lettura dello stato di questa linea oltre un tempo ritenuto sicuro per effettuare lo spegnimento in sicurezza; una possibile implementazione potrebbe essere quella di inserire il codice sopra riportato all'interno di un thread separato rispetto al codice principale dell'applicazione.

Si ricorda ancora e si insiste volutamente su questo aspetto, che questa soluzione protegge da un'interruzione improvvisa della tensione di alimentazione del Raspberry se e solo se il programma che integra questo controllo risulta in esecuzione, in tutti gli altri casi il sistema non risulta protetto ed è per questo che è sempre preferibile adottare la soluzione presentata nel prossimo paragrafo.

Il controllo attraverso servizio dedicato, eseguito automaticamente alla partenza, come già anticipato, è il modo più sicuro per gestire lo spegnimento sicuro del sistema in quanto non è necessario aggiungere tale gestione ad un eventuale programma utente, ma il servizio, lavorando in background (quindi in totale trasparenza all'utente) provvede in autonomia a controllare ed a spegnere in modo sicuro il sistema in caso di necessità.

Per definire un servizio, nei sistemi operativi più recenti basati su *systemd*, sono necessari due file distinti. Il primo tipo di file è utilizzato per descrivere la risorsa che in questo caso è un servizio (questi file in *systemd* sono chiamati *unit* ed hanno estensione *.service*). L'altro file è quello che implementa il codice vero e proprio del servizio eseguito dal sistema. Infine, per rendere il tutto operativo, si deve effettuare l'installazione del servizio che, da un punto di vista pratico, si traduce nell'eseguire alcuni comandi nella shell per fare sì che questo possa essere avviato automaticamente al *boot* del Raspberry.

Anche in questo caso il codice relativo al servizio sarà presentato come script Python, ma potrebbe essere scritto in un qualsiasi altro linguaggio di programmazione; il file *.service* utilizzato invece per descrivere il servizio sarà un semplice file di testo strutturato in modo particolare, suddiviso in sezioni ben definite, ciascuna con un suo particolare scopo e semantica.

Si inizia con il presentare il codice dello script Python che andrà ad implementare il controllo e la gestione vera e propria del segnale RPIOFF.

Codice script rpimups_shsrv.py

```
#!/usr/bin/python

# -----
# Servizio gestione shutdown da linea RPIOFF
# -----

import RPi.GPIO as GPIO
import time
import os
import sys
from datetime import datetime
import subprocess

# Determina la data corrente
now = datetime.now()
date_time = now.strftime("%m/%d/%Y, %H:%M:%S")

# Definisce file Log in modalita append, e flush immediato
LogFilename = "/home/pi/rpimups_shsrv.log"
logFile = open(LogFilename, 'a', 0)
logFile.write('rpimups_shsrv started on: ' + date_time + '\n')

# Disattiva warnings GPIO
GPIO.setwarnings(False)

# Imposta modalita' utilizzo GPIO
GPIO.setmode(GPIO.BCM)

# Definisce le linee GPIO
GPIO.setup(17, GPIO.IN) # GPIO17 input (RPIOFF)

flag = False
try:
    while (True):
        # GPIO17: gestione ingresso RPIOFF per shutdown
```

```

        if (GPIO.input(17) == 1):
            # Attende 5 secondi dopo che la linea
            # RPIOFF e' stata attivata
            counter = 5
            time.sleep(1)
            GPIO.cleanup()
            # Scrive sul Log
            now = datetime.now()
            date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
            logFile.write('rpimups_shsrv fire shutdown on: ' + date_time + '\n')
            logFile.close()
            time.sleep(0.5)
            # Comando di shutdown
            subprocess.call(["shutdown", "-h", "now"])
            break
except Exception as ex:
    print('Exception detect!')
    sys.stdout.flush()
    template = "An exception of type {0} occurred. Arguments:\n{1!r}"
    message = template.format(type(ex).__name__, ex.args)
    print(message)
    sys.stdout.flush()
finally:
    sys.exit(0)

```

Il codice dello script che implementa il servizio è anch'esso piuttosto semplice; per tracciare l'avvio e l'esecuzione dello shutdown è prevista la scrittura di un file di log **rpimups_shsrv.log** nella cartella **/home/pi**, in seguito si definisce la linea GPIO17 come ingresso. Dopodiché si ha un ciclo infinito che si occupa di controllare costantemente lo stato della linea GPIO17, se questo risulta alto, dopo un intervallo di tempo di 5 secondi lo script registra l'evento sul file di log e chiude il file, in seguito viene eseguito il comando di **shutdown -h now** utilizzando **subprocess**, a questo punto lo script termina.

Come anticipato un servizio deve essere corredato di uno unit file che ne descriva alcune caratteristiche, di seguito si riporta il contenuto del file **rpimups_shsrv.service** per il servizio in esame:

Codice unit file rpimups_shsrv.service

```

[Unit]
Description=Shutdown manager
StartLimitIntervalSec=0

[Service]
Restart=always
RestartSec=1
ExecStart=/usr/bin/python2 /home/pi/rpimups_shsrv.py

[Install]
WantedBy=multi-user.target

```

Nella sezione **[Unit]** il parametro **Description** contiene una semplice descrizione del servizio a cui l'unità si riferisce. Il parametro **StartLimitIntervalSec** fa sì che **systemd** possa fare ripartire sempre il servizio, più in dettaglio, come impostazione predefinita, **systemd** fa ripartire un servizio se quest'ultimo non ha già eseguito più di 5 tentativi in un intervallo di tempo di 10 secondi, se si eccede nel numero di tentativi il servizio non viene più fatto partire. Impostando **StartLimitIntervalSec=0** si evita questo comportamento imponendo a **systemd** di fare ripartire sempre il servizio.

La sezione **[Service]** con la voce **Restart=always** specifica che il servizio deve essere fatto sempre ripartire in presenza di una sua uscita (qualunque sia il motivo), con **RestartSec=1** che prima di riavviarlo deve attendere 1 secondo, questo per non sovraccaricare il sistema (di default il riavvio avviene dopo circa 100 ms); in seguito si ha la riga che contiene il riferimento al codice vero e proprio del servizio.

Avendo uno script python questo dovrà essere eseguito attraverso l'interprete Python ed il codice dello script sarà presente al percorso **/home/pi**. Questa sezione della unit non è interpretata da *systemd* in runtime, ma è utilizzata dai comandi di installazione (enable) e rimozione (disable) del servizio.

L'ultima sezione [Install], con WantedBy specifica per quale tipologia di stato di attività della macchina il servizio deve essere eseguito.

Nei vecchi sistemi basati su SysV lo stato era identificato con il termine *runlevel*, nel caso di *systemd* lo stato si definisce *target*, a questo livello di esposizione è sufficiente sapere che con la classificazione *multi-user.target* si abilita l'esecuzione del servizio con supporto multi-utente, interfaccia testuale e supporto di rete ed equivale da un punto di vista pratico al vecchio runlevel 3.

Completata l'analisi del codice che implementa le operazioni che saranno svolte dal servizio e della *unit* che lo descrive si deve procedere con l'installazione del servizio nel sistema.

Di seguito la procedura con i comandi da utilizzare (tutti seguiti dalla pressione del tasto *Invio*).

Ci si posiziona nella cartella **/home/pi** :

```
cd /home/pi
```

si scarica il repository con *git*

```
git clone http://github.com/Italsensor/RPI-MINI-UPS-R0
```

nel caso in cui non sia installato questo tool è possibile fare riferimento alle note riportate al fondo del manuale.

Ci si sposta nel seguente percorso:

```
cd /home/pi/RPI-MINI-UPS-R0/Examples/Shutdown/Service
```

si impostano i permessi di esecuzione con il seguente comando:

```
chmod 700 rpimups_shsrv.py ; chmod 700 rpimups_shsrv.service
```

l'utilizzo del carattere *;* permette di eseguire i comandi in successione, scrivendoli sulla stessa riga.

Per controllare se i permessi sono stati impostati correttamente si utilizza il seguente comando:

```
stat -c '%n | %A | %a' /home/pi/rpimups*
```

```
root@raspberrypi:/home/pi# stat -c '%n | %A | %a' /home/pi/rpimups*
/home/pi/rpimups_shsrv.py | -rwx----- | 700
/home/pi/rpimups_shsrv.service | -rwx----- | 700
root@raspberrypi:/home/pi#
```

Si copia il file di unit del servizio nella cartella **/etc/systemd/system**, il comando deve essere eseguito con privilegi di *root*:

```
sudo cp rpimups_shsrv.service /etc/systemd/system/rpimups_shsrv.service
```

Si copia il codice che sarà eseguito dal servizio nella cartella **/home/pi**:

```
sudo cp rpimups_shsrv.py /home/pi
```

Prima di avviare il servizio è bene prestare attenzione a quanto segue.



Si ricorda che, una volta avviato il servizio, se per qualche motivo la linea RPIOFF dovesse risultare attiva, trascorso l'intervallo di tempo predefinito sarebbe eseguito lo shutdown del sistema con conseguente spegnimento del Raspberry. Per tale motivo si consiglia quindi, prima di eseguire l'avvio del servizio, di controllare che il modulo UPS abbia l'alimentazione principale applicata per evitare un possibile shutdown non previsto.

Fatti i debiti controlli si può quindi procedere con l'avviare il servizio utilizzando il seguente comando:

```
sudo systemctl start rpimups_shsrv.service
```

Per completezza di informazione, nel caso delle *unit* che descrivono dei servizi, specificare l'estensione `.service` non è strettamente necessario in quanto, per impostazione predefinita, *systemctl* considera che l'estensione sia proprio `.service`.

Per controllare lo stato del servizio si utilizza il comando:

```
sudo systemctl status rpimups_shsrv
```

Si rende lo script avviabile al *boot* del sistema:

```
sudo systemctl enable rpimups_shsrv
```

questo comando provvede a creare un link simbolico dalla copia del file del servizio contenuto in `/lib/systemd/system` oppure in `/etc/systemd/system/` nella cartella che contiene i riferimenti dei file che *systemd* deve lanciare al boot del sistema (tipicamente in `/etc/systemd/system/nome_target.target.wants`).

Un appunto merita essere fatto circa questa differenziazione tra le cartelle che contengono i file `.service`: la cartella `/lib/systemd/system`⁹ è utilizzata dai file che sono forniti come pacchetti scaricabili dai vari repository, mentre la cartella `/etc/systemd/system/` contiene i file gestiti direttamente dall'amministratore di sistema (quindi in questo caso da noi) ed è per questo motivo che il file è stato copiato in quest'ultima specifica cartella.

In modo analogo, per rimuovere l'esecuzione del servizio al *boot* (quindi eliminare automaticamente il link simbolico creato in precedenza con il comando `enable`), si utilizza:

```
sudo systemctl disable rpimups_shsrv
```

Per completezza si riportano di seguito altri comandi utilizzabili per la gestione dei servizi.

Per fermare il servizio : `sudo systemctl stop rpimups_shsrv`

per fare ripartire il servizio: `sudo systemctl restart rpimups_shsrv`

Per maggiori dettagli sul funzionamento di *systemd*¹⁰ e sull'utilizzo del relativo daemon di gestione *systemctl*¹¹ si rimanda al manuale on line.

Per un riassunto dei passi principali sopra riportati relativi alla creazione di un servizio può essere utile consultare la guida ufficiale presente sul sito dell'organizzazione Raspberry¹².

⁹ Questo percorso è valido per distribuzioni Debian e derivate come nel caso del Raspberry, in altre distribuzioni derivate ad esempio da RedHat come CentOS il percorso è `/usr/lib/systemd/system`.

¹⁰ <https://man7.org/linux/man-pages/man1/systemd.1.html>

¹¹ <https://man7.org/linux/man-pages/man1/systemctl.1.html>

¹² <https://www.raspberrypi.org/documentation/linux/usage/systemd.md>

La scheda è anche equipaggiata con driver seriale RS232 (MAX3232) per permettere un interfacciamento diretto tra le linee TX e RX del Raspberry presenti sul connettore GPIO (rispettivamente pin 8 e pin 10) ed il mondo esterno.

Il connettore a vaschetta 9 poli presente sui PC più datati oppure sugli adattatori USB/RS232 ha la seguente connessione:

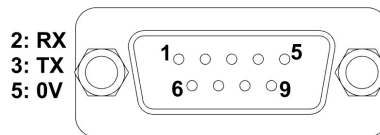


Figura 14: connettore maschio 9 poli porta RS232 (vista lato pin).

Questa disposizione corrisponde anche al connettore femmina, visto però dal lato saldature.

In questo caso il PC svolge il ruolo di *DTE* (Data Terminal Equipment), il Raspberry sarà il *DCE* (Data Communication Equipment).

Per quanto concerne la connessione in esame sono di interesse solo le linee RX-TX e lo 0V in quanto la connessione che si andrà a realizzare è la classica 115200,8,N,1 senza handshake solo come esempio per illustrare come effettuare la connessione e la comunicazione.

Facendo riferimento allo schema del connettore J2, riportato di seguito per comodità, la connessione fisica al Raspberry sarà effettuata utilizzando il pin 8 (TX), il pin 10 (RX) ed indifferentemente il pin 7 oppure 9 per la connessione dello 0V.

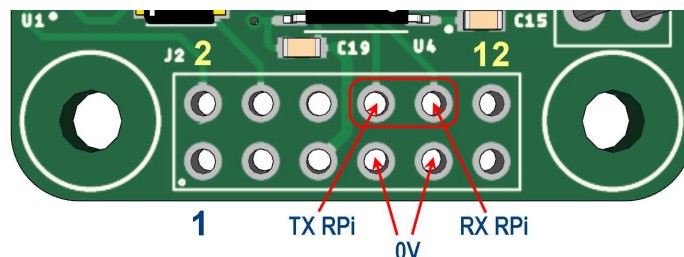


Figura 15: segnali porta seriale RS232 su connettore J2.

Poiché il connettore lato *DTE* (lato PC) è solitamente di tipo maschio, come anche il connettore che si trova sui numerosi modelli di convertitore USB/RS232 in commercio, la connessione dovrà essere effettuata intestando lato Raspberry un connettore a vaschetta di tipo femmina.

La connessione, affinché possa essere in grado di scambiare dati tra il Raspberry ed il sistema esterno (in questo caso si utilizza un PC con un adattatore USB), dovrà essere di tipo incrociato (cross) ovvero il segnale TX sul pin 8 del connettore J2 dovrà essere collegato con il segnale RX del connettore femmina 9 poli, quindi sul pin 2 ; analogamente il segnale RX sul pin 10 del connettore J2 dovrà essere collegato con il segnale TX del connettore femmina 9 poli, quindi sul pin 3.

Per la comunicazione dal PC si può utilizzare un qualsiasi terminale seriale, in questo caso sarà utilizzato **RealTerm**. Una volta in esecuzione si seleziona il pannello **Port** e si configurano i dati di connessione come riportato nella figura seguente.

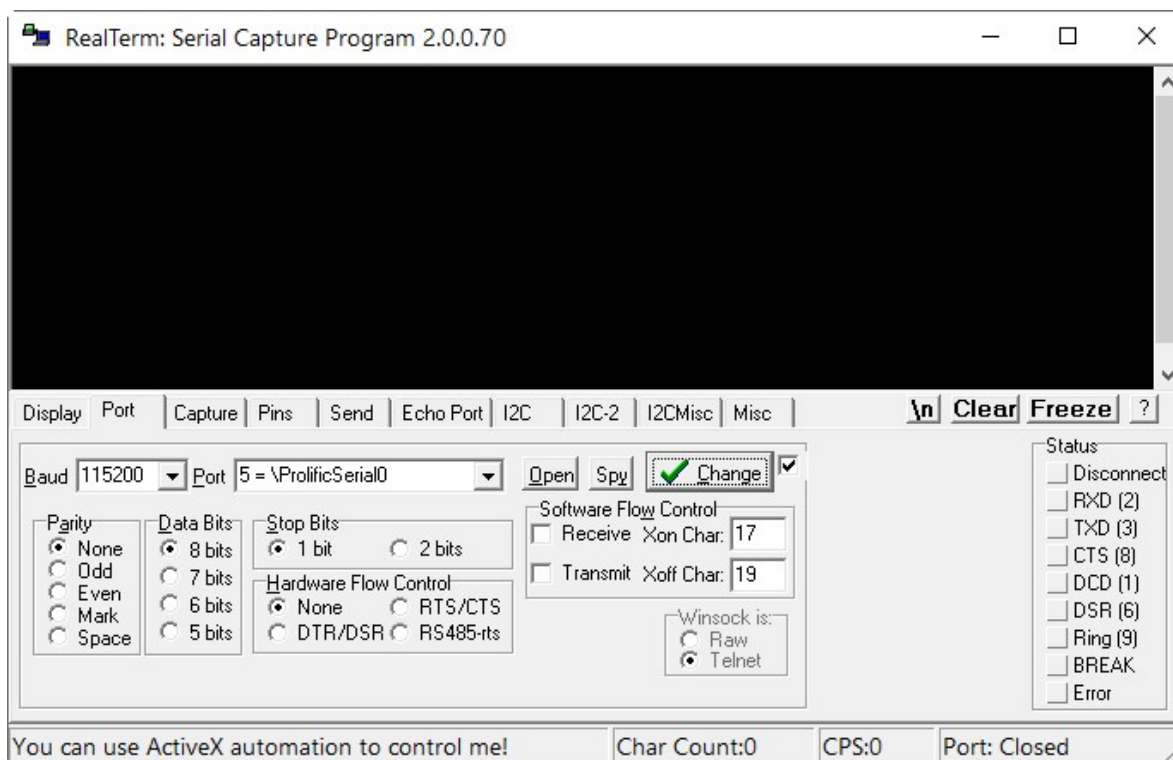


Figura 16: configurazione dati per la connessione seriale in RealTerm¹³.

Per la selezione della porta di comunicazione (Port), avendo utilizzato un adattatore USB to RS232, si deve identificare quale è il numero di porta (VCP = Virtual Com Port) assegnato dal sistema, nel caso in esempio è la COM5. Per identificare il numero di VCP su Windows si deve accedere al pannello Gestione dispositivi, un modo è quello di premere la combinazione di tasti Windows+X selezionando la voce **Gestione dispositivi**:

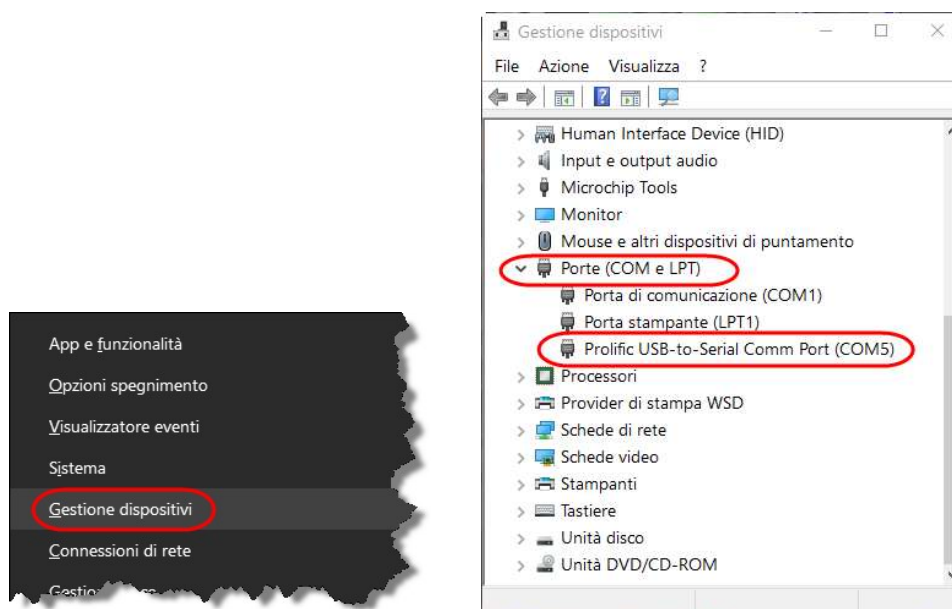


Figura 17: gestione dispositivi.

¹³ <https://sourceforge.net/projects/realterm/>

Determinato il numero di porta assegnato dal sistema all'adattatore lo si seleziona nel terminale di comunicazione seriale, a questo punto si può procedere con l'apertura della comunicazione, premendo il pulsante **Open**.

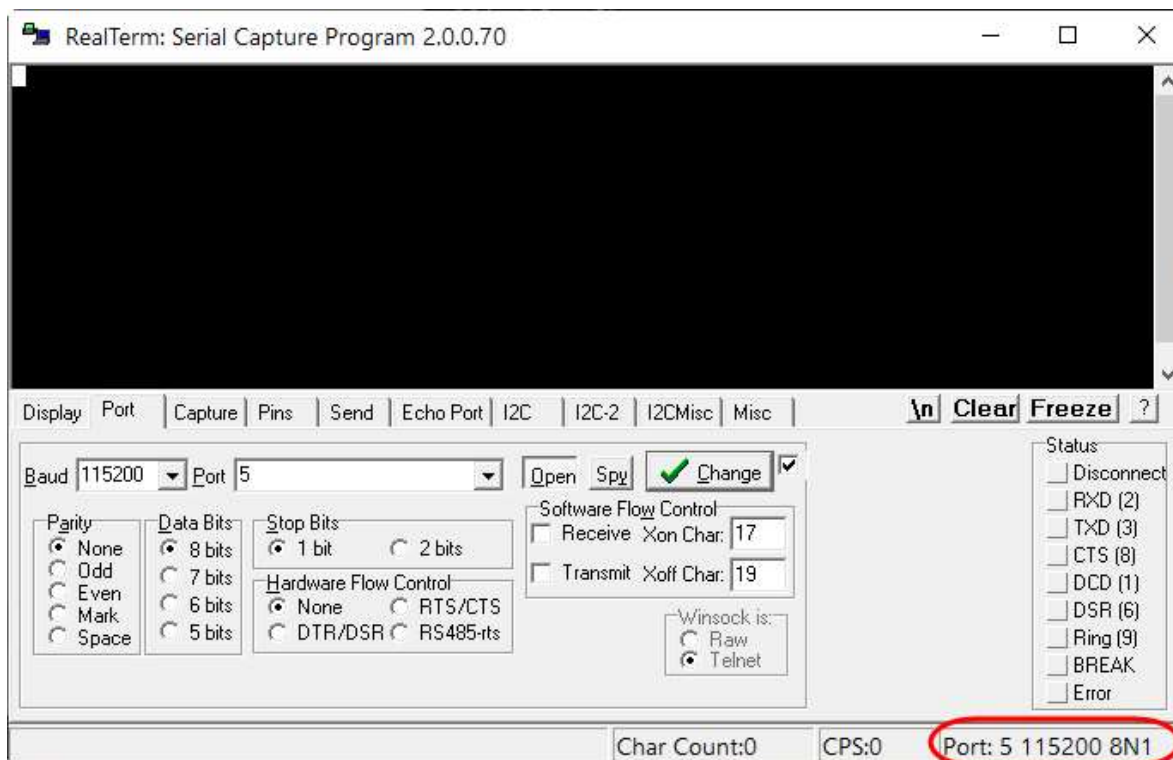


Figura 18: attivazione della comunicazione seriale.

Per visualizzare i caratteri digitati nel terminale e che saranno quindi trasmessi al Raspberry, si seleziona il pannello **Display** configurando le impostazioni come riportato nella figura seguente (selezione opzione Half Duplex):

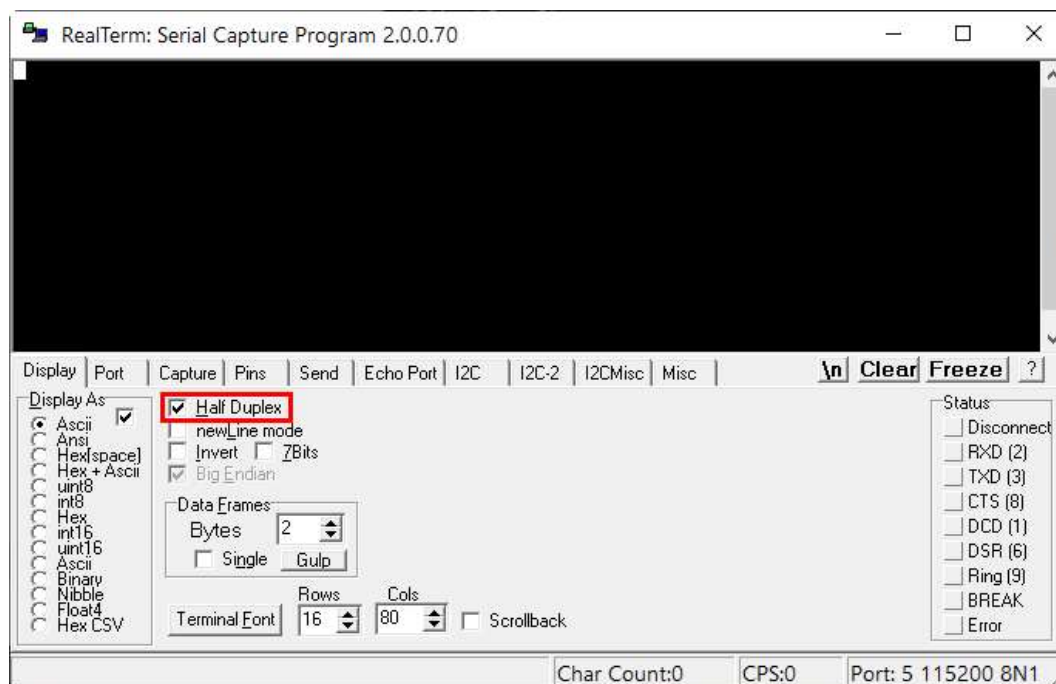


Figura 19: selezione opzioni visualizzazione Ascii e modalità Half Duplex.

Completate le impostazioni lato PC ci si sposta sul Raspberry. Dal repository si può scaricare il file **ex-serial.py** (per Python 2.x) oppure **ex-serial-py3.py** (per Python 3.X), il funzionamento è identico per entrambe le versioni.

Prima di eseguire lo script si installano alcuni pacchetti necessari per la comunicazione seriale:

```
sudo apt-get install -y python-serial
sudo apt-get install -y python3-serial
sudo apt-get install -y python3-serial-asyncio
```

alcuni pacchetti potrebbero essere già presenti se si utilizza una versione di sistema operativo recente.

Lo script visualizza semplicemente i caratteri ricevuti e rimane in attesa dei caratteri da trasmettere, la trasmissione avviene premendo il tasto Invio.

Note per utilizzo dello script di esempio su sistemi RPI3B+

Per utilizzare lo script di esempio su un RPI3B+ dopo avere configurato la porta seriale ttyAMA0 attraverso **raspi-config** disattivando l'utilizzo della stessa da parte della console di sistema (operazione che si deve fare per tutti i modelli di Raspberry al fine di disimpegnare la porta seriale dalla shell di sistema alla partenza) si deve anche disattivare il modulo Bluetooth andando a modificare il file **/boot/config.txt** aggiungendo al fondo dello stesso la seguente riga:

```
dtoverlay=pi3-disable-bt
```

si salva il file e si esce dall'editor.

In seguito si disattiva il servizio *systemd* che inizializza il modulo Bluetooth (che essendo stato disattivato con il comando precedente causerebbe un errore):

```
sudo systemctl disable hciuart
```

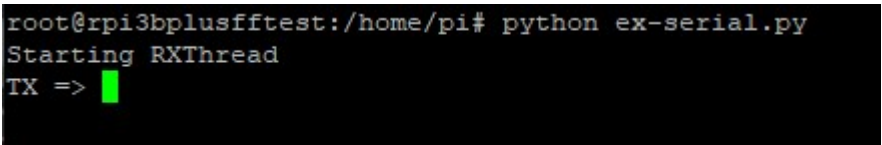
per rendere operative le modifiche si riavvia il Raspberry con il comando:

```
sudo reboot
```

Esecuzione dello script di esempio per la comunicazione seriale

Per eseguire lo script utilizzando l'interprete Python2.x si utilizza il seguente comando:

```
python ex-serial.py
```



```
root@rpi3bplusfftest:/home/pi# python ex-serial.py
Starting RXThread
TX => █
```

Per utilizzare lo script di esempio con interprete Python3.x si ha uno script leggermente modificato per poter essere correttamente eseguito anche con le versioni di Python 3.x:

```
python3 ex-serial-py3.py
```

Nel corso di questa analisi si utilizzerà lo script per Python2.x, quanto illustrato vale anche per lo script relativo all'interprete Python3.x si hanno solo differenze a livello di codice (associate essenzialmente al

fatto che certi aspetti sono stati modificati nel passaggio dalla versione 2.x alle 3.x e successive), non di funzionalità.

In questo caso non verrà effettuata un'analisi dettagliata del funzionamento dello script in sé stesso in quanto esulerebbe dallo scopo del presente manuale, sono comunque fornite le indicazioni necessarie relative alla configurazione software e hardware del Raspberry e del PC per poter instaurare correttamente la comunicazione seriale.

Lo script gestisce un semplice protocollo di comunicazione basato solo su caratteri ASCII. Tutte le informazioni ricevute devono iniziare con il carattere \$ e terminare con il carattere %, alla ricezione del terminatore viene visualizzata la stringa di caratteri appena ricevuta; per la trasmissione è invece sufficiente digitare nel terminale i caratteri desiderati e premere il tasto **Invio**.

L'implementazione del ciclo di ricezione è realizzata in modo non bloccante utilizzando un thread separato dal corpo dell'applicazione principale, in questo modo il ciclo principale può rimanere in attesa del comando utente senza bloccare la ricezione di eventuali caratteri in arrivo.

Alla partenza lo script trasmette un messaggio di avvio "**Serial port running...**", quindi se tutto è stato configurato correttamente lo stesso verrà visualizzato nel terminale in esecuzione sul PC:

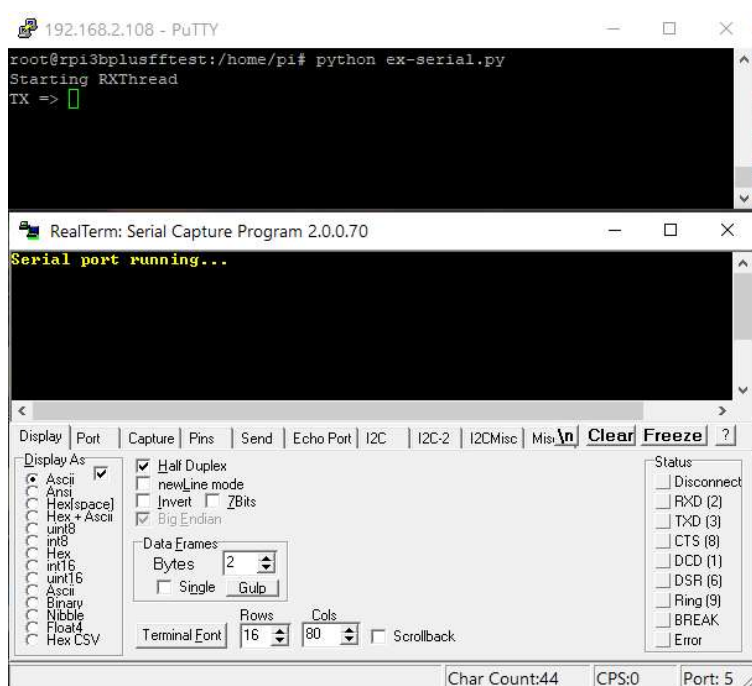


Figura 20: partenza dello script, il messaggio iniziale trasmesso dal Raspberry è visualizzato nella finestra terminale in esecuzione sul PC Windows.

Scrivendo una stringa di testo, ad esempio **\$CIAO%**, e si preme il tasto invio si attiva la trasmissione dei caratteri verso il Raspberry che provvede a visualizzarla.

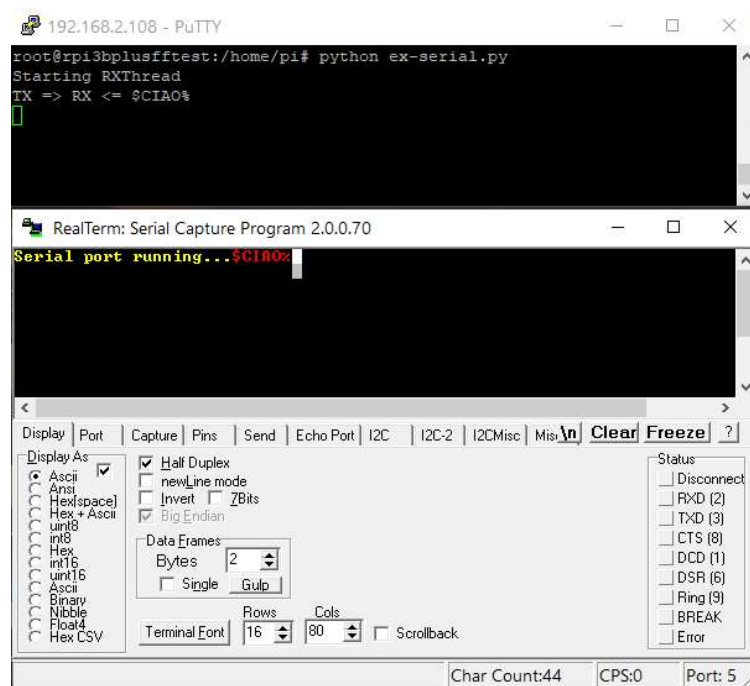


Figura 21: trasmissione messaggio da PC verso Raspberry.

Viceversa è possibile scrivere un testo nella finestra della shell sul Raspberry (in questo caso è sufficiente una qualsiasi sequenza di caratteri) seguita dalla pressione del tasto Invio per attivare la trasmissione verso il PC.

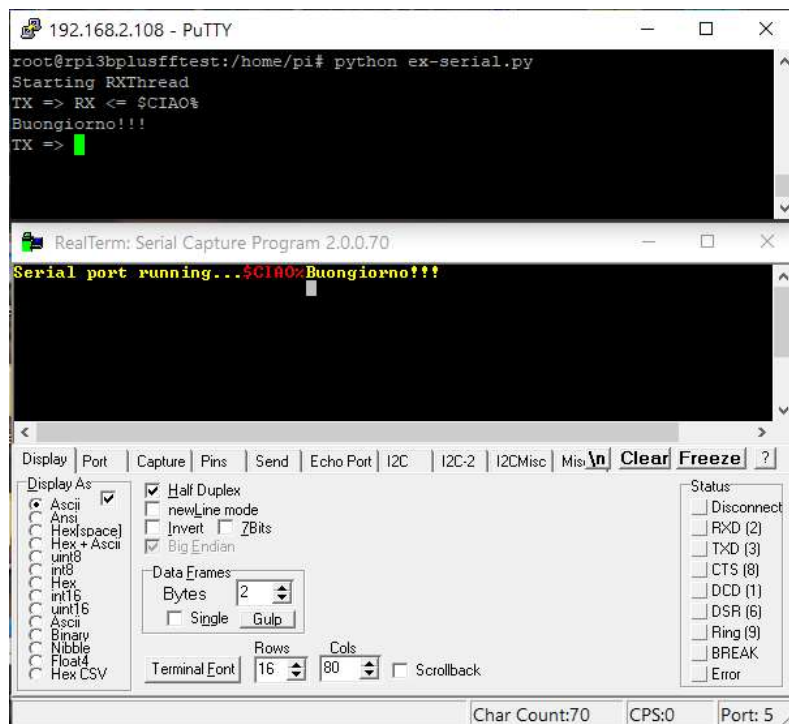


Figura 22: trasmissione messaggio da Raspberry verso PC.

Per terminare lo script è sufficiente premere la combinazione di tasti CTRL+C (^C).

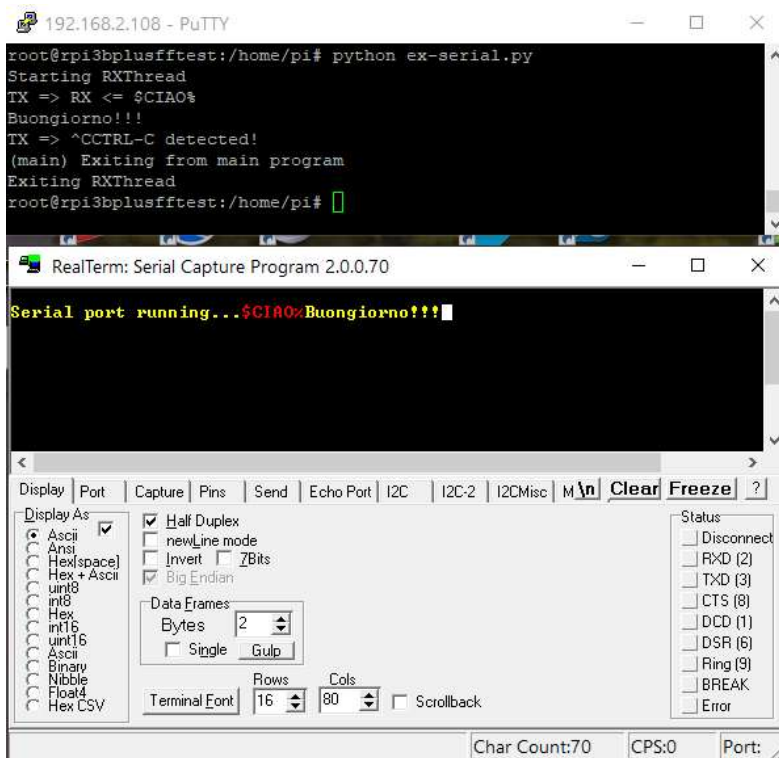


Figura 23: termine esecuzione script.

Tutti i file presentati nel manuale sono disponibili per lo scaricamento dal repository GitHub della scheda raggiungibile al seguente indirizzo:

<https://github.com/Italsensor/RPI-MINI-UPS-R0>

Per scaricare in locale tutto il materiale in formato zip è sufficiente premere il pulsante **Code** e selezionare l'opzione **Download ZIP**.

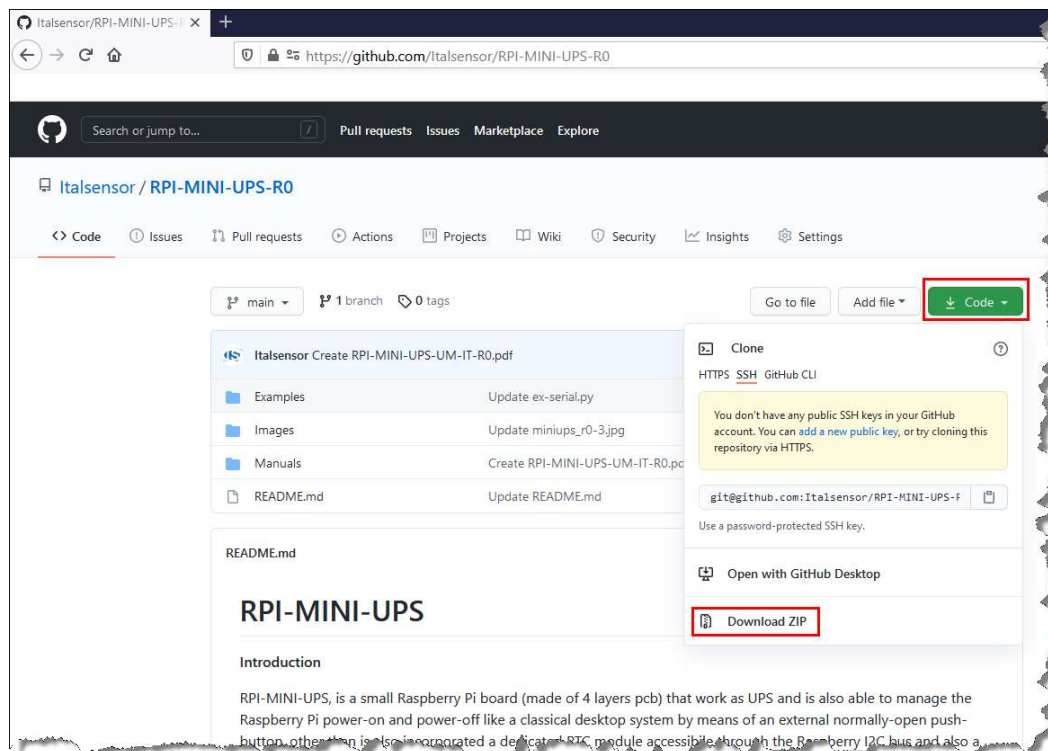


Figura 24: repository file di esempio e manuale.

Per scaricare direttamente su Raspberry attraverso una connessione internet, si può utilizzare `git`, il primo passo consiste nell'installare il tool:

```
sudo apt-get update
sudo apt-get -y install git
```

per visualizzare la versione corrente di `git`:

```
git --version
minuti
```

In seguito è possibile scaricare tutto il repository in formato zip con il seguente comando:

```
git clone http://github.com/Italsensor/RPI-MINI-UPS-R0
```

il pacchetto verrà scaricato nella cartella locale da dove è stato lanciato il comando, se si desidera che i file vengano scaricati in una cartella differente è sufficiente specificare il percorso alla fine del comando sopra riportato.

```
git clone http://github.com/Italsensor/RPI-MINI-UPS-R0 <cartella_locale>
```

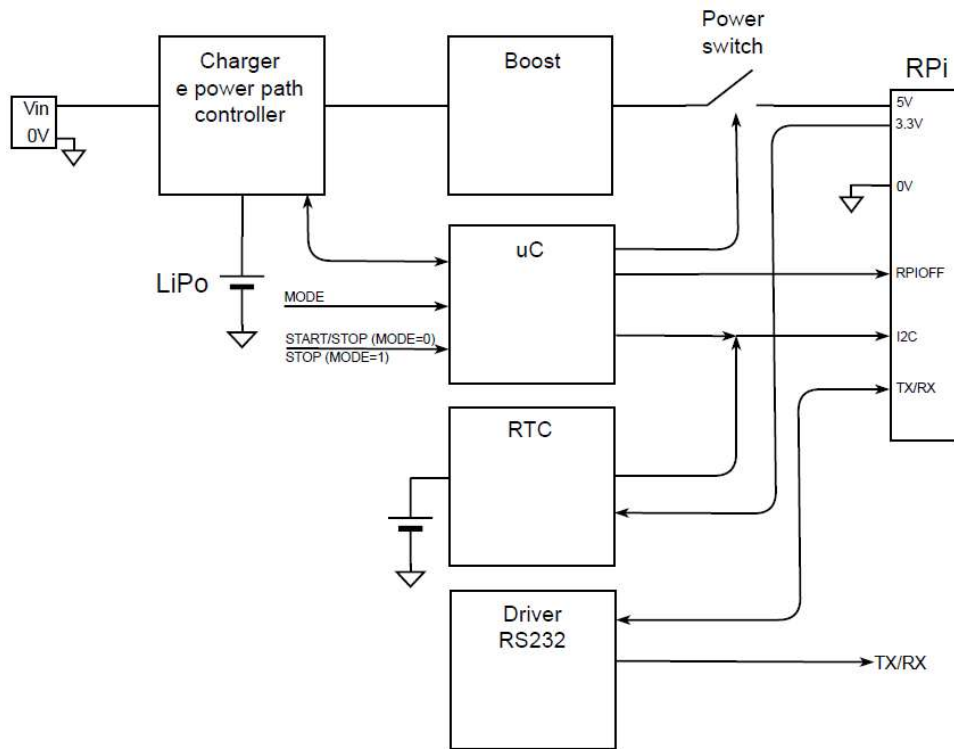
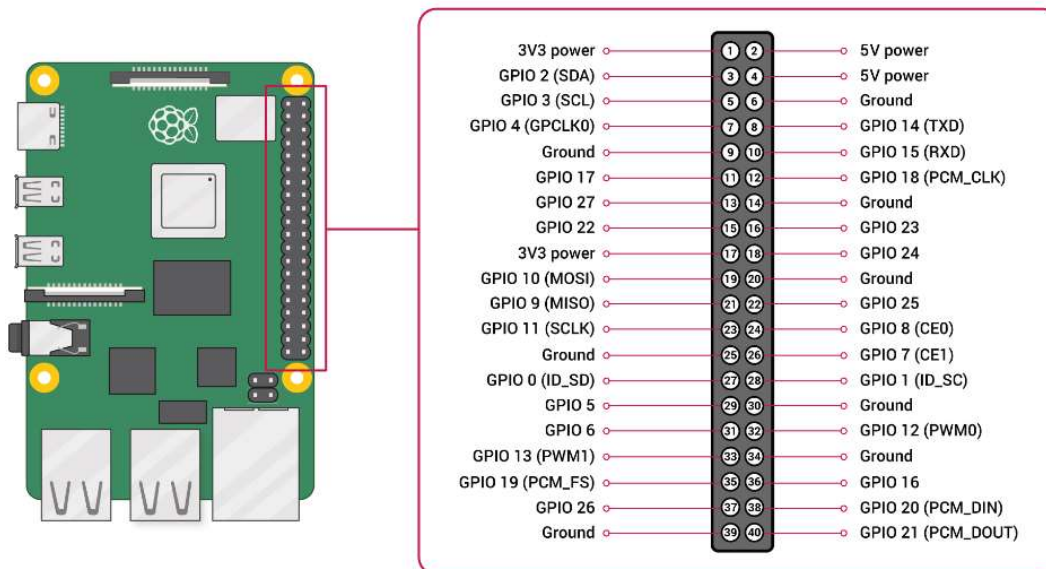


Figura 25: schema a blocchi RPi-MINI-UPS.

Figura 26: connettore GPIO 40 vie per i modelli RPi B e B+ e versioni Zero¹⁴.

¹⁴ <https://www.raspberrypi.org/documentation/usage/gpio/>

A titolo puramente indicativo, in riferimento a delle prove effettuate con un RPi3B+ Rev 1.3, sistema operativo Raspbian 10 (buster), desktop grafico, browser aperto su due pagine di YouTube con filmati in esecuzione, LibreOffice con applicazione Writer aperta, gioco precaricato Soccer in esecuzione, mouse e tastiera USB connesse, monitor HDMI connesso e partendo da una condizione di batteria LiPo carica, l'autonomia complessiva risulta mediamente di 10', la tensione della batteria all'inizio della fase di shutdown è pari a circa 3.6 V.

In modalità UPS, al ripristino della tensione di alimentazione, il sistema riparte dopo circa 1' 15", la batteria non è ovviamente completamente carica, ma la tensione ai suoi capi, in presenza della tensione di alimentazione principale è tale da permettere il superamento della soglia di accensione pari a circa 3.9 V.

Ulteriore test è stato condotto con un RPi4 model B Rev 1.1, sistema operativo Raspbian 10 (buster), desktop grafico, mouse e tastiera USB connesse, monitor HDMI connesso, doppio stress test in contemporanea (CPU e GPU) eseguito utilizzando i due tool `stress-ng` e `glxgears`.

Per l'installazione dei tool:

```
sudo apt install stress-ng mesa-utils
```

Per eseguire `stress-ng` si utilizza il seguente comando:

```
stress-ng --cpu 0 --cpu-method fft
```

Per lo stress test grafico, all'interno del desktop grafico, si apre una shell e si utilizza il seguente comando:

```
glxgears -fullscreen
```

partendo da una condizione di batteria LiPo carica, l'autonomia complessiva risulta mediamente di 7' 50", all'inizio della fase di shutdown è pari a circa 3.6 V. Temperatura ambiente rilevata intorno al RPi circa 28°C, la temperatura misurata con termocoppia in corrispondenza del centro del package metallico del SoC, dopo tre ore di funzionamento continuo dall'inizio della prova, ha fornito una lettura media pari a 56°C.

Sempre nel corso della prova si riporta un esempio di lettura rilevata eseguendo lo script `python3` (disponibile nel repository) `RPI-HW-Checker.py`. Per interrompere l'esecuzione dello script si utilizza la combinazione di tasti CTRL+C.

```
SoC temperature      : 83.7
ARM clock            : 1000265600
Core voltage         : 0.85
ARM memory allocation: 948
GPU memory allocation: 76
-----
Throttled state
-----
Under-voltage detected      : False
Arm frequency capped       : True
Currently throttled        : False
Soft temperature limit active : False
Under-voltage has occurred  : False
Arm frequency capping has occurred : True
Throttling has occurred    : True
Soft temperature limit has occurred: False
```

Figura 27: dettaglio stato sistema RPi4B sotto stress test multiplo.



MUPS-V4-000 scheda con connettore GPIO non saldato, fornito a parte,
batteria LiPo inclusa (*),
batteria CR1025 inclusa, ma non installata nel relativo battery holder
n°2 jumper passo 2.54 mm,
n°2 Dupont wires (F/M, F/F)

MUPS-V4-001 scheda con connettore GPIO non saldato, fornito a parte,
batteria LiPo inclusa (*),
n°2 jumper passo 2.54 mm,

MUPS-V4-002 scheda con morsettiera GPIO montata,
batteria LiPo inclusa (*),
Batteria CR1025 inclusa, ma non installata nel relativo battery holder,
n°2 jumper passo 2.54 mm,
n°2 Dupont wires (F/M, F/F)

MUPS-V4-003 scheda con morsettiera GPIO montata,
batteria LiPo inclusa (*),
n°2 jumper passo 2.54 mm,

(*) Per le specifiche tecniche fare riferimento a quanto riportato a pagina 3 del presente manuale.
La batteria LiPo è fornita di serie, ma non è connessa alla scheda.



La batteria LiPo deve sempre essere agganciata alla scheda prima di inserire il jumper di reset/accensione nel connettore J2 (JP1, figura 8).



Prima di collegare la scheda ad un carico esterno leggere attentamente tutte le informazioni contenute nel presente manuale di utilizzo.

MUPS-V4-MECH-KIT particolari meccanici per montaggio scheda su RPi2B, RPi3B, RPi3B+, RPi4B
n°2 viti M2.5x18 testa cilindrica impronta a croce UNI 7687 / ISO 7045 / DIN 7985
n°2 dadi M2.5 DIN 934
n°2 distanziali plastici altezza 11 mm, foro interno 2.7 mm, diametro esterno 5 mm