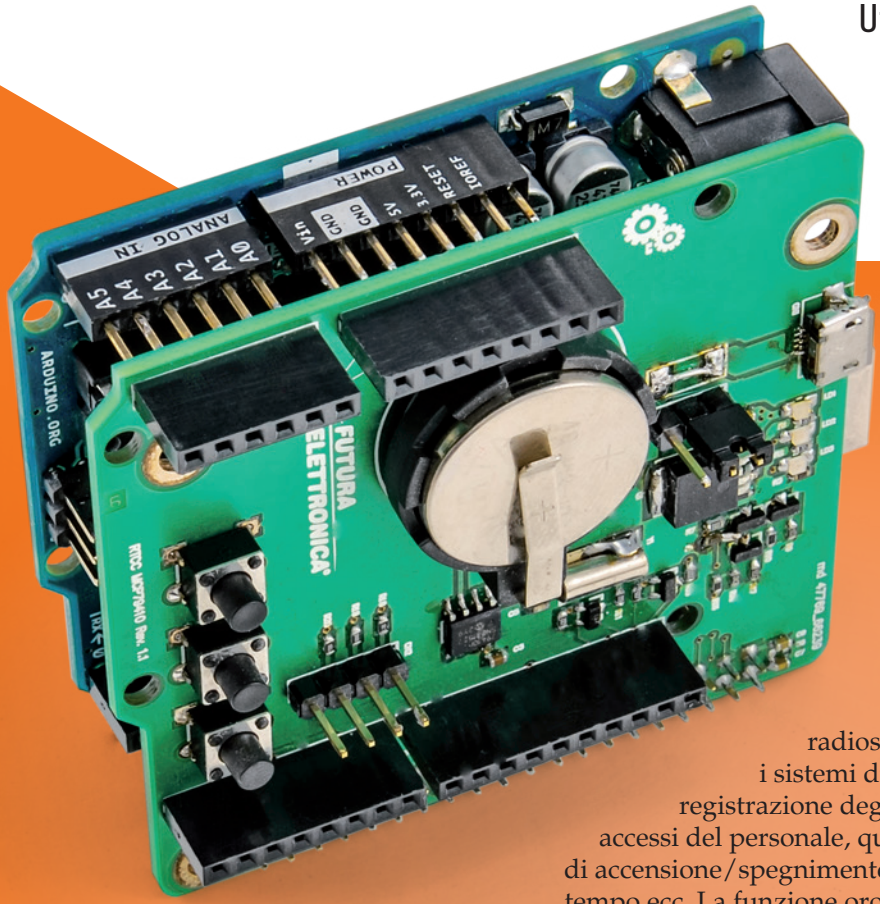

# RTC SHIELD PER ARDUINO E RASPBERRY PI

di MATTEO DESTRO

Utilizziamo un nuovo Real Time Clock di Microchip rendendolo disponibile su uno shield per le due schede di prototipazione. Prima puntata.

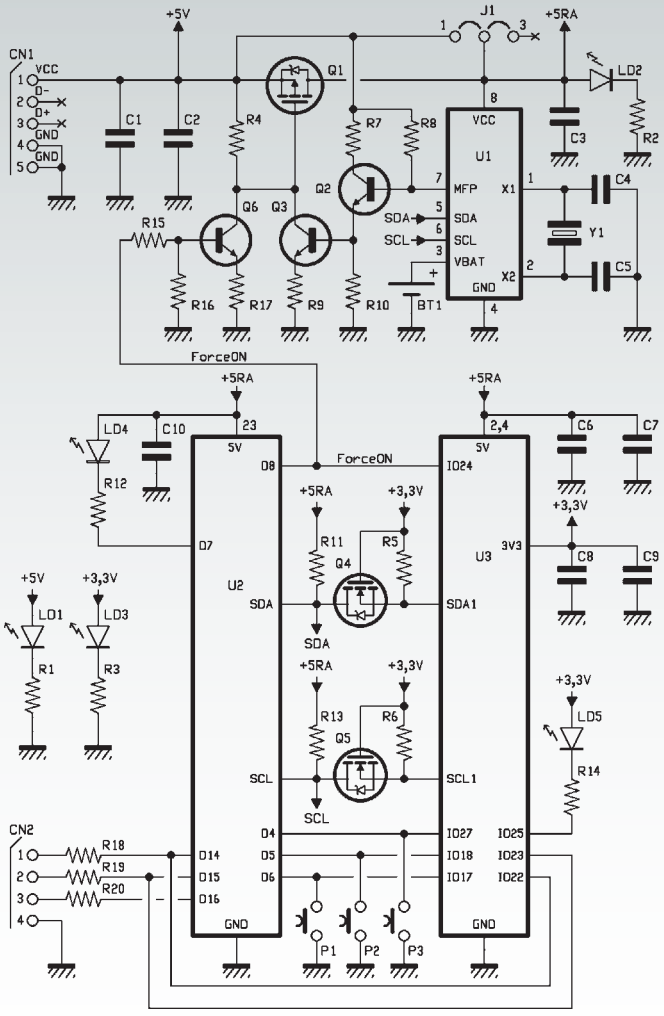



Molte applicazioni richiedono l'informazione oraria, che viene ottenuta localmente mediante circuiti chiamati RTC (Real Time Clock) o RTCC (Real Time Clock Calendar); tra esse le radiosveglie, i sistemi di registrazione degli accessi del personale, quelli di accensione/spegnimento a tempo ecc. La funzione orologio può essere implementata in maniera semplice grazie a circuiti integrati come il DS1307, da noi utilizzato più volte, o come il recente MCP79410 della Microchip; con questo abbiamo realizzato uno shield polivalente applicabile alle schede Arduino e Raspberry Pi. Cominciamo con il descrivere l'integrato Microchip e la libreria sviluppata per l'ecosistema Arduino. Nella seconda puntata vedremo l'impiego con la Raspberry Pi.

## L'INTEGRATO MCP79410

L'RTCC appartiene a una famiglia di integrati che comprende anche MCP79411 e MCP79412; questi ultimi due hanno memorizzato un MAC address univoco per le applicazioni ethernet. Più esattamente, l'MCP79411 ha un MAC address in formato EUI-48 mentre l'MCP79412 ce l'ha in formato EUI-64. La dicitura EUI sta per *Extended Unique Identifier* e può essere utilizzata per assegnare un indirizzo hardware a 48 o 64 bit univoco nelle applicazioni di networking. Solitamente si utilizzano i 24 bit più significativi per identificare

## [schema **ELETTRICO**]




- configurazione di ore, minuti e secondi sia nel formato 24h che nel 12h (AM/PM);
- configurazione di giorno, mese, anno e giorno della settimana;
- gestione automatica degli anni bisestili;
- oscillatore a 32.768 Hz;
- calibrazione/regolazione interna digitale con risoluzione di ±1ppm (range massimo ±129ppm);
- due allarmi programmabili;
- TimeStamp sia su power-up che su power-down;
- 64 Byte di memoria SRAM tamponata;
- 128 Byte di memoria EEPROM con possibilità di scrittura paginata a 8 byte alla volta;
- 8 Byte di memoria EEPROM protetta, per scrivere nella quale (si può un solo byte per volta) bisogna eseguire una sequenza di sblocco;
- interfaccia di comunicazione I²C fino a 400 kHz.

Inoltre l'integrato ha pin di uscita open-drain multifunzione che descriveremo più avanti.
La **Fig. 1** mostra lo schema a blocchi interno dell'integrato MCP79410 dove sono ben visibili i blocchi logici che vanno a formare il dispositivo.
L'interfaccia I²C prevede due indirizzi hardware, uno per la gestione del RTCC e l'altro per la gestione della memoria EEPROM. La **Fig. 2** mostra i due possibili control byte contenenti l'indirizzo hardware per la sezione RTCC o EEPROM. Il primo ha come valore ***0b1101111x*** dove al posto della "***x***" ci sarà "***0***" o "***1***" a seconda dell'operazione che si vuole eseguire ("***0***" - scrittura; "***1***" - lettura). Il secondo invece sarà ***0b1010111x***, dove "***x***" assume i valori descritti sopra.

in modo univoco un fornitore, produttore o organizzazione a livello globale: tale codice viene detto OUI (*Organizationally Unique Identifier*). L'acquirente di tale codice -di norma un costruttore di dispositivi elettronici- dispone dei rimanenti 24 Bit per completare il codice da assegnare ai propri prodotti (3 byte, per un totale di oltre 16 milioni di codici univoci possibili). Per fare un esempio, attualmente Microchip dispone di 3 codici OUI, che sono rispettivamente *0xD88039*, *0x001EC0* e *0x0004A3*. Con questi, Microchip può dare origine a ben 50 milioni di possibili indirizzi MAC univoci. Quindi acquistando un RTCC della serie MCP79411 o MCP79412 troverete un indirizzo MAC univoco utilizzabile senza dover pagare alcuna royalty. Non dovendo sviluppare un'applicazione di networking, abbiamo utilizzato l'MCP79410, che mette a disposizione:

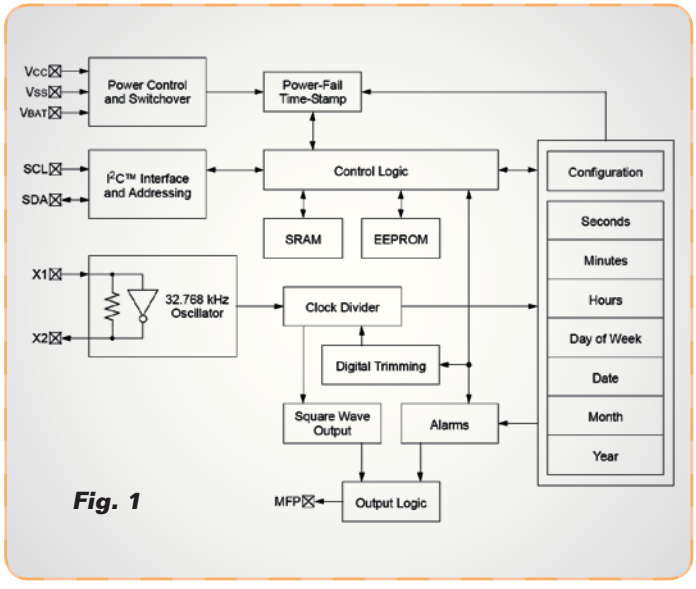
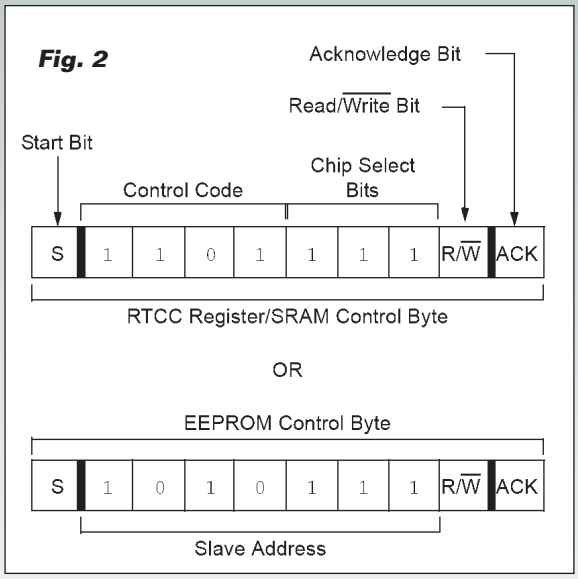



***Fig. 1***



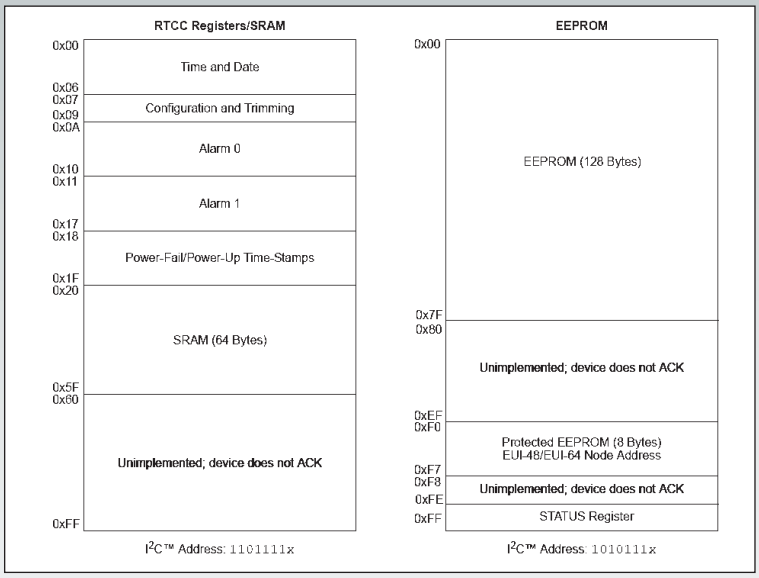
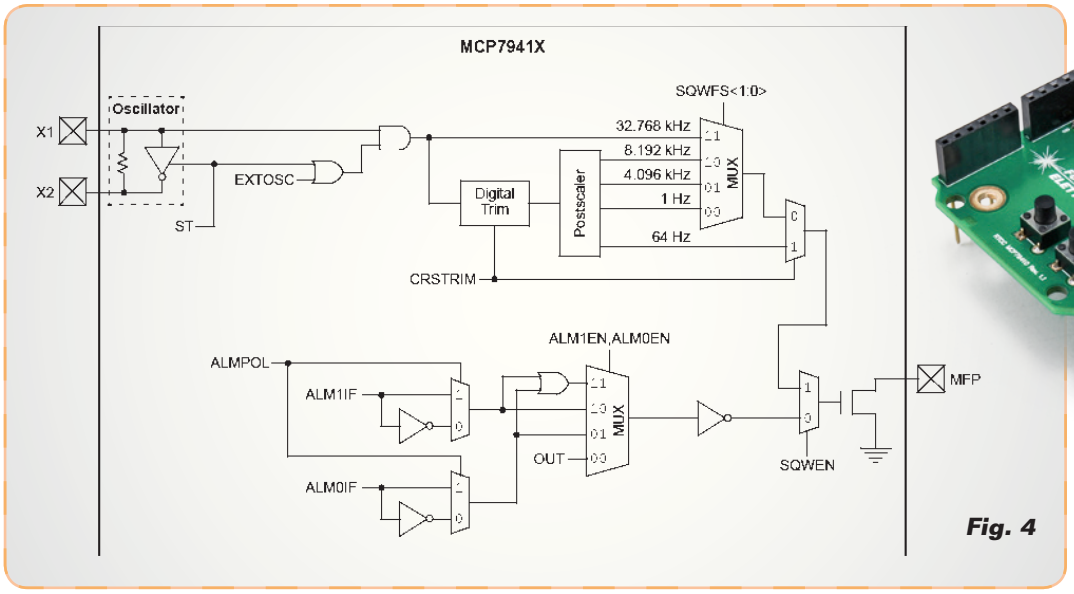




**Fig. 3**

La mappatura dei registri e delle memorie è riportata nella **Fig. 3**: gli indirizzi da ***0x00*** a ***0x1F*** contengono i registri di configurazione della data e dell'ora, nonché i registri per configurare gli allarmi 0 e 1. Concludono la sezione, i registri di gestione del TimeStamp al power-up e al power-down. Dall'indirizzo ***0x20*** all'indirizzo ***0x5F*** viene mappata la memoria SRAM tamponata. I restanti indirizzi fino a ***0xFF*** non sono utilizzati.
Quanto appena detto fa riferimento all'indirizzo hardware ***0b1101111x***. La EEPROM è invece mappata a partire dall'indirizzo ***0x00*** a ***0x7F***, ai quali seguono una serie di indirizzi non utilizzati (Da ***0x80*** a ***0xEF***); dall'indirizzo ***0xF0*** fino a ***0xF7*** troviamo la memoria EEPROM protetta, la quale può essere programmata solo un byte alla volta e solo dopo avere eseguito una sequenza di sblocco. Per ultimo, all'indirizzo ***0xFF*** trova posto il registro STATUS. Quanto appena detto fa riferimento all'indirizzo hardware ***0b1010111x.***

## SCHEMA ELETTRICO

Con l'integrato MCP79410 abbiamo realizzato uno shield applicabile sia a una scheda Arduino Uno R3 che a una Raspberry Pi 2.0/3.0/B+. L'elettronica e un'opportuna programmazione dell'Arduino piuttosto che della Raspberry Pi, permettono di configurare in toto l'MCP79410, soprattutto nelle sue funzionalità di gestione allarmi necessarie all'accensione e allo spegnimento automatico della scheda collegata allo shield. Tutte le funzioni necessarie a gestire l'integrato sono state raccolte in librerie sia per Arduino che per Raspberry Pi.
Iniziamo lo studio dello schema elettrico partendo dal connettore CN1 (microUSB) il quale porta l'alimentazione stabilizzata +5Vcc, al MOSFET Q1 (a canale P) utilizzato come interruttore hardware. Il Q1 è in interdizione fintantoché il gate non viene cortocircuitato a massa, allorché la $V_{GS}$ supera il valore di soglia aprendo il canale del MOSFET e quindi trasferendo l'alimentazione dal source al drain. La "*nuova*" alimentazione viene chiamata +5VRA. Per portare a massa il gate Q1 ci sono due possibilità: l'MCP79410, attraverso l'uscita open-collector "***MFP***" agisce sui due transistor Q2 e Q3; la linea di comando "***ForceOn***" polarizza il transistor Q6 che porta a massa il gate di Q1.
Torniamo sulla linea di uscita **MFP**, di cui la **Fig. 4** mostra la logica interna all'MCP79410: essa può essere configurata per lavorare come uscita generica, per fornire uno stato logico al verificarsi dei due allarmi configurabili, nonché come riporto della frequenza di clock. Se si vuole che l'uscita riporti la frequenza di clock dell'oscillatore o una delle frequenze derivate, si deve settare a "***1***" logico il bit "***SQWEN***" del registro "***CONTROL***". Per decidere quale frequenza debba essere riportata sull'uscita si devono configurare i bit "***SQWFS0***" e "***SQWFS1***" sempre del registro "***CONTROL***. Le frequenze possibili sono 1, 4.096, 8.192 e 32.768 Hz (consultate il data-sheet per i dettagli).
Volendo che MFP diventi un'uscita generica a livello logico si deve settare il bit "***SQWEN***" a zero e disabilitare entrambi gli allarmi. Il nostro caso, invece, prevede che MFP sia pilotata dagli allarmi 0 e 1, quindi "***SQWEN***" deve essere settato a zero e i bit "***ALM0EN***" e "***ALM1EN***" devono assumere entrambi, o almeno uno di essi, un valore logico alto. La MFP, essendo un'uscita open-collector, può essere intesa come linea di interrupt indicante che uno o entrambi gli allarmi sono scattati in seguito al raggiungimento delle condizioni impostate.



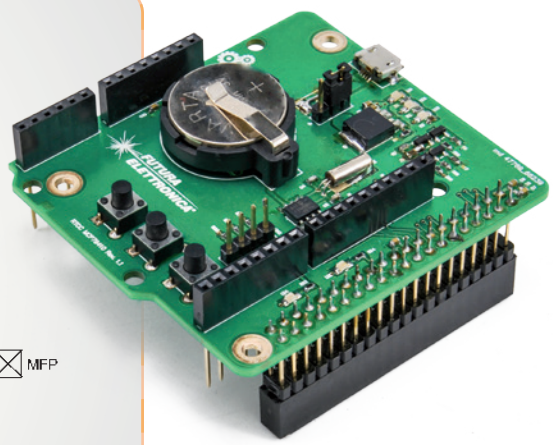



***Fig. 4***

Vedremo più avanti come configurare e sfruttare i due allarmi messi a disposizione. Per concludere, la MFP può essere portata a un microcontrollore per indicargli che si è attivato uno o entrambi gli allarmi configurati evitando di scrivere del codice di polling dello stato dei due allarmi. Della MFP si può inoltre definire se è attiva a livello 0 o 1; allo scopo si deve agire sul bit "***ALMPOL***" nei registri "***ALM0WKDAY***" e "***ALM1WKDAY***". Nella nostra applicazione l'uscita dev'essere attiva a livello alto. La **Tabella 1** mostra il comportamento della MFP in base alla configurazione del bit ALMPOL e dello stato del flag di interrupt dell'allarme *x*: quando ALMPOL è settato a "***1***" l'uscita MFP segue lo stato del flag di interrupt dell'allarme *n*, nel caso di un solo allarme attivo; se entrambi gli allarmi sono attivi e configurati, la linea MFP" segue l'andamento dell'OR logico dei due flag di interrupt degli allarmi 0 e 1. Se invece il bit ALMPOL è settato a 0 la MPF è il negato del flag di interrupt dell'allarme *n* impostato (un solo allarme impostato) oppure segue l'andamento del NAND logico dei due flag di interrupt degli allarmi 0 e 1. L'integrato MCP79410 viene alimentato tramite la tensione +5VAR, la quale viene anche portata alla scheda Arduino Uno R3 e alla scheda Raspberry Pi. L'MCP79410 ha anche un ingresso $V_{BAT}$ al quale si connette una batteria tampone che lo tiene alimentato in caso di mancanza di alimentazione. Durante il funzionamento a batteria, solo una parte dell'integrato rimane attiva: in particolare, il blocco RTCC (gestione data e ora) e i 64 Byte della SRAM. Inoltre rimane attiva la linea MFP se configurata per lavorare in abbinamento agli allarmi 0 e 1, altrimenti sarà disabilitata.
Le funzioni di gestione della data e dell'ora richiedono una frequenza di clock di 32.768 Hz che viene ricavata collegando un quarzo ai pin X1 e X2. Infine l'interfaccia di comunicazione $I^2C$ viene portata sia ad Arduino che a Raspberry Pi tramite le rispettive connessioni; notate che il bus $I^2C$ richiede un traslatore di livello per l'interconnessione alla Raspberry Pi, in quanto le sue linee di I/O lavorano con un livello logico +3V3. Ciò viene ottenuto sfruttando i MOSFET Q4 e Q5 che funzionano come in **Fig. 5**: il traslatore di livello bidirezionale siffatto vale anche per SPI o qualsiasi linea bidirezionale e in esso ogni sezione è alimentata con un livello di tensione differente; nel nostro caso +3V3 per l'elettronica della scheda Raspberry Pi +5V per l'integrato MCP79410. La sezione a bassa tensione, a sinistra, prevede due resistenze di pull-up collegate al source del MOSFET, per le linee SDA e SCL, mentre il gate viene collegato direttamente alla tensione di alimentazione più bassa: nel nostro caso +3V3. La sezione ad alta tensione prevede anch'essa due resistenze di pull-up collegate al drain del MOSFET. Il MOSFET utilizzato per entrambe le linee è uno a riempimento a canale N con il substrato internamente connesso al source (deve avere integrato il diodo che connette il substrato

***Tabella 1***

| ALMPOL | ALMXIF | MFP |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| ALMPOL | ALM0IF | ALM1IF | MFP |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

con il drain creando una giunzione NP); quelli da noi usati sono BSS123. Vediamo in dettaglio i tre stati di funzionamento, con la premessa che quanto detto vale per entrambe le linee:

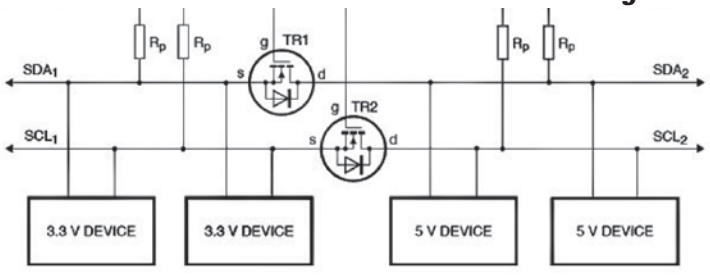
1) nessuna periferica impone un livello di tensione basso sulla linea, quindi sul lato a 3,3V la resistenza di pull-up impone l'1 logico e di conseguenza la $V_{GS}$ non supera la soglia del MOSFET, in quanto $V_G$ e Vs hanno la medesima tensione, quindi il MOSFET non entra in conduzione; sul lato a 5V la linea si ritrova a un livello logico alto imposto dalla rispettiva resistenza di pull-up;
2) la periferica a 3,3V impone sulla linea un livello di tensione basso, perciò il source del MOSFET va a livello logico basso; la $V_{GS}$ supera la soglia e manda in conduzione il MOSFET; con il MOSFET in conduzione, il livello logico basso viene imposto anche nellla sezione a 5V;
3) la periferica a 5V alta tensione impone sulla linea un livello di tensione basso; sulla linea a bassa tensione abbiamo la resistenza di pull-up che impone un livello logico alto e quindi manda in conduzione il diodo presente nel MOSFET (la $V_S$ scende e di conseguenza la $V_{GS}$ supera la soglia mandando in conduzione il MOSFET, quindi a questo punto le due sezioni hanno il medesimo livello di tensione sulla linea).

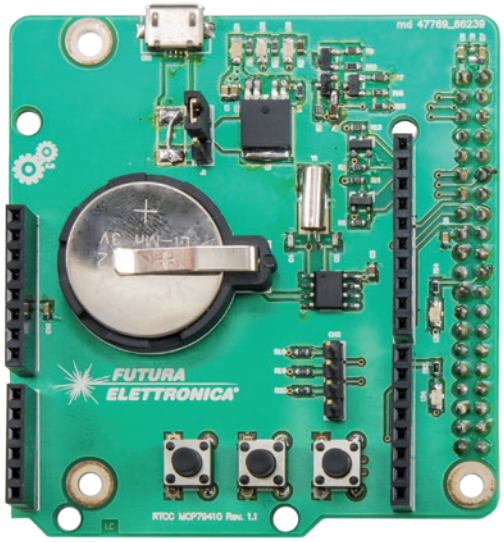
La massima frequenza ammessa sull'$I^2C$ è 400 kHz. I pulsanti P1, P2 e P3 collegati sia ad Arduino che a Raspberry Pi sono utilizzati durante le fase di configurazione del RTCC e degli allarmi. Per agevolare le fasi di sviluppo sono stati predisposti tre segnali di trigger portati al connettore CN2 per Arduino e due per la Raspberry Pi, anch'essi portati al connettore CN2. I segnali di trigger sono in comune tra le due schede, in quanto non possono essere collegate contemporaneamente allo shield. Sempre dalle due schede, parte il segnale "***ForceOn***" che serve a mantenere in stato di ON l'interruttore elettronico Q1. Completano gli schemi elettrici i LED LD4 (collegato ad Arduino) e LD5 (collegato a Raspberry Pi) utilizzati durante la fase di configurazione dell'MCP79410. Infine LD1 indica la presenza dell'alimentazione principale in arrivo dal microUSB; LD2 indica invece che è presente l'alimentazione secondaria +5VAR. LD3 indica la presenza dell'alimentazione +3V3.
Un'ultima nota riguarda il jumper J1, che serve a bypassare Q1: serve durante la configurazione dell'MCP79410 soprattutto se si collega la Raspberry Pi. Arduino in realtà è già auto-alimentata dal connettore USB tipo B connesso al PC, per la programmazione tramite l'IDE Arduino.

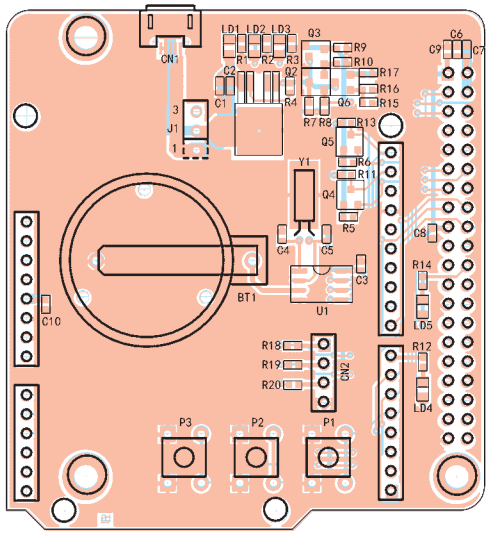
## I REGISTRI DELL'MCP79410

Prima di passare alla libreria per Arduino Uno R3 conviene conoscere i registri mappati dell'MCP79410. La **Fig. 6** evidenzia la mappatura in memoria di ognuno e il significato di ogni singolo bit o gruppi di bit; è suddivisa in cinque sezioni, la prima delle quali contiene i registri per configurare il TimeKeeper, ovvero data e ora del RTCC. In questi registri sono presenti anche i bit di configurazione per l'attivazione dell'oscillatore e per il funzionamento a batteria dell'integrato. Trovano posto anche i registri "***CONTROL***" per attivare/disattivare gli allarmi, per configurare il pin MFP e il registro OSCTRIM per tarare la frequenza del clock, oltre al registro "***EEUNLOCK***" per sproteggere la memoria EEPROM per la scrittura di un singolo byte alla volta (per sproteggere la EEPROM occorre scrivere una serie di byte ben definiti nel registro).
La seconda sezione contiene i registri di configurazione dell'allarme zero, mentre la terza contiene i registri di configurazione dell'allarme uno. La quarta sezione contiene i registri per la lettura del power-down timestamp. La quinta contiene i registri per la lettura del power-up timestamp. Notate che le informazioni di data e ora sono gestite sempre in formato BCD, quindi, ad esempio, il registro "***RTCSEC***" contiene l'informazione dei secondi e sfrutta i bit da 0 a 3 per memorizzare le unità, mentre i bit da 4 a 6 vengono usati per le decine. Il bit più significativo serve per attivare l'oscillatore.
Quindi un ipotetico valore dei secondi di 39 viene suddiviso in 9 unità e 3 decine.
L'impostazione in formato BCD visibile nella **Tabella 2** viene usata in tutti i registri che contengono un'informazione di data e ora sia che siano del TimeKeeper sia degli allarmi che dei timestamp.

***Fig. 5***






## Elenco Componenti:

R1÷R3: 1,2 kohm 1% (0603)
R4: 22 kohm 1% (0603)
R5÷R7: 2,2 kohm 1% (0603)
R8, R16: 10 kohm 1% (0603)
R9: 4,7 kohm 1% (0603)
R10, R12, R14: 680 ohm 1% (0603)
R11: 2,2 kohm 1% (0603)
R13: 2,2 kohm 1% (0603)
R15: 3,3 kohm 1% (0603)
R17: 4,7 kohm 1% (0603)
R18÷R20: 1 kohm 1% (0603)
C1, C2: 10 µF ceramico (0603)
C3: 100 nF ceramico (0603)
C4, C5: 10 pF ceramico (0603)
C6: -
C7÷C10: 10 µF ceramico (0603)
LD1: LED rosso (0805)
LD2: LED giallo (0805)
LD3÷LD5: LED verde (0805)
P1÷P3: Microswitch
Q1: SPD50P03LG
Q2, Q3: BC817
Q4, Q5: BSS123
Q6: BC817
CN1: Connettore micro-USB
CN2: Strip maschio 4 vie
J1: Strip maschio 3 vie
BT1: Porta batterie CR2032
U1: MCP79410-I/SN
U2: Arduino UNO Rev3
U3: RaspberryPi 2/3/B+
Y1: Quarzo 32768 Hz

Varie:
- Batteria CR2032
- Strip M/F 6 vie
- Strip M/F 8 vie (2 pz.)
- Strip M/F 10 vie
- Circuito stampato S1254

### LIBRERIA PER MCP79410

La libreria permette di configurare e gestire l'integrato MCP79410 ed è divisa in tre file; il primo contiene le funzioni sviluppate per l'occasione e ha estensione *.cpp* (lo abbiamo chiamato ***MCP79410.cpp***). Il secondo ha estensione *.h* e contiene le dichiarazioni di funzione del precedente più tutte le variabili e le strutture dati necessarie (***MCP79410.h***). Il terzo (***keywords.txt***) è un file di testo contenente le parole chiave delle funzioni pubbliche da utilizzare negli sketch Arduino. Tutti questi devono essere raggruppati sotto una cartella comune, all'interno della cartella di installazione dell'IDE Arduino, nominata ***MCP79410***. Così facendo, comparirà la libreria nello IDE Arduino sotto la voce di menu ***Sketch>Include Library***.
Quindi supponendo che lo IDE Arduino sia stato installato sotto ***C:\Program Files (x86)\Arduino*** la nostra libreria dovrà essere salvata nel seguente percorso: ***C:\Program Files (x86)\Arduino\libraries\MCP79410***. Oltre ai file di libreria è consuetudine aggiungere una cartella con degli sketch di esempio: nel nostro caso abbiamo creato la sotto-cartella ***Examples → MCP79410_SAdvancedSettings*** nella quale trovano posto i file del nostro sketch che ci permettono di configurare i registri di MCP79410.
Cominciamo a descrivere a grandi linee il file ***MCP79410.h*** che, come anzi detto, contiene le definizioni di tutte le funzioni utilizzate e la dichiarazione delle variabili e delle strutture dati pubbliche e private. In testa c'è una serie di dichiarazioni di costanti che identificano gli indirizzi hardware dell'integrato, per quanto riguarda sia la parte RTCC che la parte EEPROM,

***Tabella 2***

| OSC START | SECTEN | SECONE |
|---|---|---|
| X | 3 | 9 |

più tutti gli indirizzi dei registri presenti. Seguono una serie di costanti utili per settare o resettare i bit di configurazione presenti nei vari registri. Per ogni costante è stato pensato un commento che spiega come sfruttare la costante presentata. Ad esempio per attivare l'oscillatore del RTCC si deve agire sul registro "***RTCSEC***" e in particolare sul bit più significativo, eseguendo un'operazione booleana di tipo ***OR*** (*RTCSEC | OSCILLATOR_BIT_ON*). Invece per resettare tale bit si userà un'operazione booleana di tipo ***AND*** (*RTCSEC & OSCILLATOR_BIT_OFF*). Per eseguire una delle precedenti operazioni è necessario prima leggere il registro di interesse, eseguire l'operazione booleana desiderata e infine scrivere il nuovo valore nel registro.
Seguono una serie di costanti di configurazione con la descrizione di ogni singolo bit che li compone. Questa sezione, associata alla lettura attenta del data-sheet, aiuta a comprendere appieno come e cosa fare per configurare l'integrato. Le costanti possono essere modificate a piacere dall'utente a seconda delle proprie esigenze; nulla vieta di crearne di nuove. L'ultima sezione riguarda la gestione della EEPROM; ricordiamo che l'indirizzo hardware da utilizzare per accedere ad essa è diverso da quello del RTCC.
Dopo le dichiarazioni di costanti appena spiegate seguono le dichiarazioni di tutte le funzioni pubbliche disponibili all'utente finale per la realizzazione del proprio sketch, nonché la dichiarazione di tutte le variabili necessarie alla gestione della libreria. In particolare vale la pena soffermarci sulle strutture dati che ci danno la possibilità di agire a livello di bit per la configurazione dei registri. Vedremo tra poco che è possibile configurare i registri in due modi: uno prevede l'utilizzo di apposite funzioni dedicate e l'altro passa dalla configurazione delle strutture dati e successiva programmazione dei registri.
Ad esempio, la struttura dati nel **Listato 1** permette di configurare ogni singolo bit del registro "***CONTROL***". Supponendo di volere abilitare l'allarme 0 si dovrà settare il bit "***Alarm0_Enable***" con la seguente sintassi:

```
mcp79410.ControlReg.Bit.Alarm0_Enable = 1;
```

Così facendo abbiamo solo settato un valore in SRAM del microcontrollore ATmega 328P presente sulla scheda Arduino Uno R3. Il passo successivo sarà richiamare un'apposita funzione per andare a scrivere il dato nell'apposita locazione di memoria dell'integrato MCP79410, ad esempio utilizzando la funzione di libreria "***WriteSingleReg***" alla quale vanno passati una serie di parametri, tra cui l'indirizzo e il valore del registro di cui si vuole modificare il valore. La sintassi completa sarà:

```
mcp79410.WriteSingleReg(RTCC_HW_ADD, CONTROL_ADD,
mcp79410.ControlReg.ControlByte);
```

Oltre alla struttura dati appena illustrata, ne seguono altre, più articolate, che permettono la configurazione dei registri TimeKeeper, gli allarmi e la lettura dei TimeStamp al PowerUp e PowerDown. Come esempio vediamo -nel **Listato 2**- la struttura dati per la gestione dei registri TimeKeeper. Come si può notare si è definita, per ogni registro di configurazione del RTCC (TimeKeeper), una struttura di tipo "***union***" che permette di agire su ogni singolo bit del registro da configurare. Ogni "*union*" è stata raggruppata in una struttura dati "***struct***" nominata "***TimeKeeper***". Con questo approccio si ha un unico "*contenitore*" al quale è possibile accedere per la configurazione di ogni singolo bit o gruppi di bit di ogni registro. Facciamo un esempio chiarificatore: supponiamo di dover settare il bit "***StartOsc***" del registro "***RTCSEC***". Questo bit è presente nella "***union***" "***TimeKeeperSeconds***" quindi per settarlo si deve usare la seguente sintassi:

***Fig. 6***

| Addr. | Register Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|---|---|
| Section 5.3 "Timekeeping" | | | | | | | | | |
| 00h | RTCSEC | ST | SECTEN2 | SECTEN1 | SECTEN0 | SECONE3 | SECONE2 | SECONE1 | SECONE0 |
| 01h | RTCMIN | — | MINTEN2 | MINTEN1 | MINTEN0 | MINONE3 | MINONE2 | MINONE1 | MINONE0 |
| 02h | RTCHOUR | — | 12/24 | AM/PM HRTEN1 | HRTEN0 | HRONE3 | HRONE2 | HRONE1 | HRONE0 |
| 03h | RTCWKDAY | — | — | OSCRUN | PWRFAIL | VBATEN | WKDAY2 | WKDAY1 | WKDAY0 |
| 04h | RTCDATE | — | — | DATETEN1 | DATETEN0 | DATEONE3 | DATEONE2 | DATEONE1 | DATEONE0 |
| 05h | RTCMTH | — | — | LPYR | MTHTEN0 | MTHONE3 | MTHONE2 | MTHONE1 | MTHONE0 |
| 06h | RTCYEAR | YRTEN3 | YRTEN2 | YRTEN1 | YRTEN0 | YRONE3 | YRONE2 | YRONE1 | YRONE0 |
| 07h | CONTROL | OUT | SQWEN | ALM1EN | ALM0EN | EXTOSC | CRSTRIM | SQWFS1 | SQWFS0 |
| 08h | OSCTRIM | SIGN | TRIMVAL6 | TRIMVAL5 | TRIMVAL4 | TRIMVAL3 | TRIMVAL2 | TRIMVAL1 | TRIMVAL0 |
| 09h | EEUNLOCK | Protected EEPROM Unlock Register (not a physical register) | | | | | | | |
| Section 5.4 "Alarms" | | | | | | | | | |
| 0Ah | ALM0SEC | — | SECTEN2 | SECTEN1 | SECTEN0 | SECONE3 | SECONE2 | SECONE1 | SECONE0 |
| 0Bh | ALM0MIN | — | MINTEN2 | MINTEN1 | MINTEN0 | MINONE3 | MINONE2 | MINONE1 | MINONE0 |
| 0Ch | ALM0HOUR | — | 12/24[2] | AM/PM HRTEN1 | HRTEN0 | HRONE3 | HRONE2 | HRONE1 | HRONE0 |
| 0Dh | ALM0WKDAY | ALMPOL | ALM0MSK2 | ALM0MSK1 | ALM0MSK0 | ALM0IF | WKDAY2 | WKDAY1 | WKDAY0 |
| 0Eh | ALM0DATE | — | — | DATETEN1 | DATETEN0 | DATEONE3 | DATEONE2 | DATEONE1 | DATEONE0 |
| 0Fh | ALM0MTH | — | — | — | MTHTEN0 | MTHONE3 | MTHONE2 | MTHONE1 | MTHONE0 |
| 10h | Reserved | Reserved – Do not use | | | | | | | |
| Section 5.4 "Alarms" | | | | | | | | | |
| 11h | ALM1SEC | — | SECTEN2 | SECTEN1 | SECTEN0 | SECONE3 | SECONE2 | SECONE1 | SECONE0 |
| 12h | ALM1MIN | — | MINTEN2 | MINTEN1 | MINTEN0 | MINONE3 | MINONE2 | MINONE1 | MINONE0 |
| 13h | ALM1HOUR | — | 12/24[2] | AM/PM HRTEN1 | HRTEN0 | HRONE3 | HRONE2 | HRONE1 | HRONE0 |
| 14h | ALM1WKDAY | ALMPOL[3] | ALM1MSK2 | ALM1MSK1 | ALM1MSK0 | ALM1IF | WKDAY2 | WKDAY1 | WKDAY0 |
| 15h | ALM1DATE | — | — | DATETEN1 | DATETEN0 | DATEONE3 | DATEONE2 | DATEONE1 | DATEONE0 |
| 16h | ALM1MTH | — | — | — | MTHTEN0 | MTHONE3 | MTHONE2 | MTHONE1 | MTHONE0 |
| 17h | Reserved | Reserved – Do not use | | | | | | | |
| Section 5.7.1 "Power-Fail Time-Stamp" | | | | | | | | | |
| Power-Down Time Stamp | | | | | | | | | |
| 18h | PWRDNMIN | — | MINTEN2 | MINTEN1 | MINTEN0 | MINONE3 | MINONE2 | MINONE1 | MINONE0 |
| 19h | PWRDNHOUR | — | 12/24 | AM/PM HRTEN1 | HRTEN0 | HRONE3 | HRONE2 | HRONE1 | HRONE0 |
| 1Ah | PWRDNDATE | — | — | DATETEN1 | DATETEN0 | DATEONE3 | DATEONE2 | DATEONE1 | DATEONE0 |
| 1Bh | PWRDNMTH | WKDAY2 | WKDAY1 | WKDAY0 | MTHTEN0 | MTHONE3 | MTHONE2 | MTHONE1 | MTHONE0 |
| Power-Up Time Stamp | | | | | | | | | |
| 1Ch | PWRUPMIN | — | MINTEN2 | MINTEN1 | MINTEN0 | MINONE3 | MINONE2 | MINONE1 | MINONE0 |
| 1Dh | PWRUPHOUR | — | 12/24 | AM/PM HRTEN1 | HRTEN0 | HRONE3 | HRONE2 | HRONE1 | HRONE0 |
| 1Eh | PWRUPDATE | — | — | DATETEN1 | DATETEN0 | DATEONE3 | DATEONE2 | DATEONE1 | DATEONE0 |
| 1Fh | PWRUPMTH | WKDAY2 | WKDAY1 | WKDAY0 | MTHTEN0 | MTHONE3 | MTHONE2 | MTHONE1 | MTHONE0 |

## Listato 1

```
union ControlReg {
    uint8_t  ControlByte;
    struct {
          uint8_t SquareWaveFreqOutput              :2;
          uint8_t CoarseTrimEnable                  :1;
          uint8_t ExtOscInput                       :1;
          uint8_t Alarm0_Enable                     :1;
          uint8_t Alarm1_Enable                     :1;
          uint8_t SquareWaveOutputEnable            :1;
          uint8_t LogicLevelOutput                  :1;
    } Bit;
} ControlReg;
```

```
mcp79410.TimeKeeper.Second.SecBit.StartOsc = 1;
```

Fatto ciò, si può richiamare la funzione di libreria "***WriteSingleReg***" per la scrittura di un singolo registro, oppure se si preferisce, configurare prima tutti i registri di cui sopra e poi chiamare la funzione di libreria "***WriteTimeKeeping***" alla quale dobbiamo passare un solo parametro ad indicare se il formato di gestione dell'ora debba essere nel formato 12h o 24h. Quindi la sintassi sarà:

```
mcp79410.WriteTimeKeeping(0);
```

Passiamo ora alle funzioni messe a disposizione dalla libreria. Sono state definite otto funzioni di uso generale, di cui tre servono a manipolare i singoli bit di un registro e le restanti lavorano direttamente sui singoli byte o gruppi di essi:

```
1) ToggleSingleBit(uint8_t ControlByte, uint8_t RegAdd, uint8_t Bit)
2) SetSingleBit(uint8_t ControlByte, uint8_t RegAdd, uint8_t Bit)
3) ResetSingleBit(uint8_t ControlByte, uint8_t RegAdd, uint8_t Bit)
4) WriteSingleReg(uint8_t ControlByte, uint8_t RegAdd, uint8_t RegData)
5) WriteArray(uint8_t ControlByte, uint8_t StartAdd, uint8_t Lenght)
6) ClearReg(uint8_t ControlByte, uint8_t RegAdd)
7) ReadSingleReg(uint8_t ControlByte, uint8_t RegAdd)
8) ReadArray(uint8_t ControlByte, uint8_t StartAdd, uint8_t Lenght)
```

La funzione (1) esegue la funzione "***Toggle***" sul bit desiderato del rispettivo registro. Quindi se il bit indicato è "***0***" diventa "***1***" e viceversa. I parametri da passare sono il "***ControlByte***" che identifica l'indirizzo hardware del nostro RTCC (consultare il file **MCP79410.h** per gli indirizzi hardware assegnati al device), il "***RegAdd***" che è l'indirizzo del registro su cui si vuole agire e infine il parametro "***Bit***" che identifica il bit che si vuole modificare (da 0 a 7). Il concetto appena esposto si può estendere alle successive due funzioni, (2) e (3), che eseguono rispettivamente un "***Set***" o un "***Reset***" del bit desiderato: i parametri da passare sono gli stessi. Le successive funzioni agiscono invece sui rispettivi byte sia in scrittura che in lettura. Per quanto riguarda le funzioni di scrittura (4) e lettura (7) di un singolo byte i parametri da passare alla funzione prevedono il solito "***ControlByte***", seguito dall'indirizzo su cui scrivere o da cui leggere "***RegAdd***" e, per quanto riguarda la sola scrittura il valore che si vuole scrivere ovvero "***RegData***". La funzione (6) "***ClearReg***" azzera un registro e necessita degli stessi parametri della funzione (7). Infine ci sono due funzioni di scrittura (5) e lettura (8) di n valori consecutivi le quali vogliono come parametri il "***ControlByte***", l'indirizzo di partenza "***StartAdd***" e infine il numero di byte "***Lenght***". Le funzioni (5) e (8) si appoggiano su un array di byte detto "***DataArray***" di lunghezza 16, più che sufficiente per le nostre esigenze. Le funzioni che fanno utilizzo dell'array di dati vengono utili per leggere o scrivere nella EEPROM o SRAM dell'integrato. Quando si deve scrivere nella EEPROM si può al massimo inviare otto byte alla volta, perché il buffer del device ha tale dimensione massima.
Passiamo ora a descrivere le funzioni più specializzate create apposta per gestire al meglio l'integrato MCP79410:

```
1) GeneralPurposeOutputBit(uint8_t SetReset)
2) SquareWaveOutputBit(uint8_t EnableDisable)
3) Alarm1Bit(uint8_t EnableDisable)
4) Alarm0Bit(uint8_t EnableDisable)
5) ExternalOscillatorBit(uint8_t EnableDisable)
6) CoarseTrimModeBit(uint8_t EnableDisable)
7) SetOutputFrequencyBit(uint8_t OutputFreq)
```

La funzione (1) serve per impostare a "***1***" logico o a "***0***" logico l'uscita MFP; ciò può avvenire nel solo caso che entrambi gli allarmi siano disabilitati e che l'uscita non sia impostata per riportare la frequenza di clock dell'oscillatore. Il parametro "***SetReset***", come suggerisce il nome, determina se l'uscita debba essere settata o resettata ("***1***" setta l'uscita, "***0***" viceversa). La funzione (2) abilita/disabilita la possibilità di riportare sulla linea MFP la frequenza di clock. Il parametro "***EnableDisable***" abilita o disabilita questa funzione ("***1***" abilita il riporto sull'uscita del clock, "***0***" viceversa). Le funzioni (3) e (4) servono per abilitare gli allarmi e, come per la precedente, l'unico parametro da passare è "***EnableDisable***". La funzione (5) serve per configurare l'integrato a ricevere un segnale di clock sul pin X1. Questo è utile se non si vuole usare il quarzo da 32.768Hz sui pin X1 e X2. Il solo parametro da passare è "***EnableDisable***" con il consueto significato.
La funzione (6) serve per attivare/disattivare il "***Coarse Trim mode***", il quale abbinato al registro "***OSCTRIM***" permette una regolazione grossolana della base dei tempi per la gestione del RTCC. La regolazione viene applicata con una cadenza di 128

Hz, apportando una notevole influenza sulla base dei tempi. È quindi preferibile lasciare disabilitata questa funzione e, se necessario, regolare la base dei tempi sfruttando il solo registro ***OSCTRIM***, ovviamente seguendo una apposita procedura (vedi riquadro *Digital Trimming*).
Le funzioni finora esposte modificano un solo bit per volta del registro "***CONTROL***" e per fare ciò sfruttano le funzioni generiche "***SetSingleBit***" e "***ResetSingleBit***" descritte precedentemente.
La funzione (7) serve per selezionare la frequenza da riportare sul pin MFP; la selezione perde di significato se l'uscita è configurata per la gestione degli allarmi o come uscita digitale generica.
Seguono ora un'altra serie di funzioni specializzate per la configurazione di alcune importanti caratteristiche di funzionamento messe a disposizione dell'integrato MCP79410. Queste impostazioni riguardano il TimeKeeper, gli allarmi e le funzioni di Timestamp:

```
1) StartOscillatorBit(uint8_t EnableDisable)
2) Hour12or24TimeFormatBit(uint8_t SetHourType)
3) AmPmBit(uint8_t SetAmPm)
4) VbatEnBit(uint8_t EnableDisable)
5) AlarmHour12or24TimeFormatBit(uint8_t SetHourType, uint8_t Alarm0_1)
6) AlarmAmPmBit(uint8_t SetAmPm, uint8_t Alarm0_1)
7) AlarmIntOutputPolarityBit(uint8_t SetReset, uint8_t Alarm0_1)
8) AlarmMaskBit(uint8_t Alarm0_1, uint8_t Mask)
9) ResetAlarmIntFlagBit(uint8_t Alarm0_1)
10) PowerHour12or24TimeFormatBit(uint8_t SetHourType, uint8_t PowerDownUp)
11) PowerAmPmBit(uint8_t SetAmPm, uint8_t PowerDownUp)
12) ResetPwFailBit(void)
```

La funzione (1) è fondamentale in quanto serve per attivare o disattivare l'oscillatore. Se l'oscillatore è spento non si ha nessuna attività da parte del RTCC e quindi nessuna gestione di data, ora e relativi allarmi. Il parametro "***EnableDisable***" abilita o disabilita l'oscillatore ("***1***" abilita l'oscillatore, "***0***" disabilita). Le funzioni (2), (5) e (10) servono per impostare il formato dell'ora rispettivamente per il timekeeper, gli allarmi e il timestamp al power-up/power-down. La rappresentazione dell'ora può essere del tipo 12h, accompagnato dalla dicitura AM/PM, oppure 24h; quindi a seconda del formato che si desidera utilizzare è conveniente allineare tutte e tre le possibili configurazioni in modo da non essere spaiati. Non ha molto senso avere il timekeeper impostato per lavorare in formato 24h e gli allarmi nel formato 12h, a meno di particolari applicazioni. La funzione (2) necessita di un solo parametro, "***SetHourType***", per selezionare uno dei due formati a disposizione: "***1***" imposta il formato 12h mentre "***0***" il formato 24h. Le funzioni (5) e (10) richiedono un parametro aggiuntivo per identificare rispettivamente a quale allarme/

## Listato 2

```
typedef union TimeKeeperSecond {
    uint8_t  SecByte;
    struct {
        uint8_t SecOne          :4;
        uint8_t SecTen          :3;
        uint8_t StartOsc  :1;
    } SecBit;
} TimeKeeperSeconds;
typedef union TimeKeeperMinute {
    uint8_t  MinByte;
    struct {
        uint8_t MinOne          :4;
        uint8_t MinTen          :3;
        uint8_t Free            :1;
    } MinBit;
} TimeKeeperMinute;
typedef union TimeKeeperHour12 {
    uint8_t  Hour_12Byte;
    struct {
        uint8_t HrOne           :4;
        uint8_t HrTen           :1;
        uint8_t AmPm            :1;
        uint8_t _12_24          :1;
        uint8_t Free            :1;
    } Hour_12Bit;
} TimeKeeperHour12;
typedef union TimeKeeperHour24 {
    uint8_t  Hour_24Byte;
    struct {
        uint8_t HrOne           :4;
        uint8_t HrTen           :2;
        uint8_t _12_24          :1;
        uint8_t Free            :1;
    } Hour_24Bit;
} TimeKeeperHour24;
typedef union TimeKeeperWeekDay {
    uint8_t  WkDayByte;
    struct {
        uint8_t WkDay           :3;
        uint8_t VbatEn          :1;
        uint8_t PwrFail   :1;
        uint8_t OSCrun          :1;
        uint8_t Free            :2;
    } WkDayBit;
} TimeKeeperWeekDay;
typedef union TimeKeeperDate {
    uint8_t  DateByte;
    struct {
        uint8_t DateOne   :4;
        uint8_t DateTen   :2;
        uint8_t Free            :2;
    } DateBit;
} TimeKeeperDate;
typedef union TimeKeeperMonth {
    uint8_t  MonthByte;
    struct {
        uint8_t MonthOne  :4;
        uint8_t MonthTen  :1;
        uint8_t LeapYear  :1;
        uint8_t Free            :2;
    } MonthBit;
} TimeKeeperMonth;
typedef union TimeKeeperYear {
    uint8_t  YearByte;
    struct {
        uint8_t YearOne   :4;
        uint8_t YearTen   :4;
    } YearBit;
} TimeKeeperYear;

struct {
    TimeKeeperSeconds     Second;
    TimeKeeperMinute      Minute;
    TimeKeeperHour12      Hour12;
    TimeKeeperHour24      Hour24;
    TimeKeeperWeekDay     WeekDay;
    TimeKeeperDate        Date;
    TimeKeeperMonth       Month;
    TimeKeeperYear        Year;
} TimeKeeper;
```

timestamp applicare la modifica del parametro. Le funzioni (3), (6) e (11) servono per impostare, in merito al formato 12h dell'ora, se l'ora impostata fa riferimento al mattino o al pomeriggio (AM per la mattina e PM per il pomeriggio). Il parametro da passare è "***SetAmPm***", se "***1***" imposta PM invece se "***0***" imposta AM. Le funzioni (6) e (11) necessitano di un parametro aggiuntivo, come già spiegato per le funzioni (5) e (10). La funzione (4) serve per attivare la gestione della alimentazione a batteria in caso di mancanza di alimentazione primaria; quindi per alimentare a batteria l'RTCC si deve attivare il rispettivo bit e per fare ciò si utilizza la funzione appena introdotta: il parametro da passare è il consueto "***EnableDisable***".
La funzione (7) serve per impostare la polarità dell'uscita MFP (Tabella 1). I parametri da passare sono due: il primo ("***SetReset***") imposta la polarità, ovvero "***1***" polarità con logica ad alto livello e "***0***" polarità con logica a basso livello. Il secondo parametro ("***Alarm0_1***") serve per selezionare su quale allarme eseguire la modifica della polarità.
La funzione (8) serve per impostare la maschera di confronto per gli allarmi. Quando si configurano gli allarmi, oltre a impostare la data e l'ora desiderate per la generazione dell'interrupt, si deve anche impostare la maschera di confronto la quale ci permette di decidere su cosa eseguire il test per scatenare l'evento di allarme. Le opzioni disponibili sono:

- confronto solo sui secondi;
- confronto solo sui minuti;
- confronto solo sull'ora (tiene conto del formato impostato ovvero 12h o 24h);
- confronto solo sul giorno della settimana;
- confronto solo sulla data;
- confronto totale (vengono confrontati i secondi, i minuti, le ore, il giorno della settimana, la data e il mese).

Quando si genera l'interrupt dell'allarme n viene settato il rispettivo flag il quale deve essere azzerato manualmente da codice. Non va mai lasciato pendente. Per assolvere al compito si può sfruttare la funzione (9) a cui si deve passare un unico parametro che identifica l'allarme 0 o 1.
Per ultima la funzione (12) la quale serve per resettare il bit "***PwFail***" il quale indica che è mancata l'alimentazione principale e che quindi sono stati memorizzati i timestamp al power-down e al power-up. In altre parole viene salvata la data e l'ora del momento in cui è mancata/ritornata l'alimentazione. Quindi dopo avere letto i registri dei due timestamp si deve resettare obbligatoriamente questo flag se si vuole che alla prossima mancanza/ritorno alimentazione vengano nuovamente aggiornati i registri.
Rimangono da descrivere le ultime funzioni le quali, abbinate alle strutture dati precedentemente descritte, ci permettono di configurare agevolmente blocchi di registri associati alle varie funzioni messe a disposizione dal nostro RTCC:

```
1) WriteTimeKeeping(uint8_t Hour12or24Format)
2) ReadTimeKeeping(void)
3) WriteAlarmRegister(uint8_t Alarm0_1, uint8_t Hour12or24Format)
4) ReadAlarmRegister(uint8_t Alarm0_1)
5) WritePowerDownUpRegister(uint8_t PowerDownUp, uint8_t Hour12or24Format)
6) ReadPowerDownUpRegister(uint8_t PowerDownUp)
```

La funzione (1) serve per programmare la data e l'ora del RTCC, come già ampiamente detto sono compresi anche i bit di attivazione dell'oscillatore e della gestione dell'alimentazione a batteria. Quindi sfruttando la struttura dati apposita si configurano i registri con i valori corretti e poi si programma l'integrato usando la funzione (1). L'unico parametro che richiede è la selezione del formato dell'ora (12h o 24h). La funzione (2) serve per leggere la configurazione assegnata al TimeKeeper, la funzione legge i registri dell'integrato e li trasferisce nella struttura dati apposita che può essere poi utilizzata per le proprie applicazioni. Lo stesso discorso si applica alle funzioni (3) e (4) che servono rispettivamente per configurare i registri degli allarmi o leggerne il loro contenuto. Le due funzioni fanno riferimento alla loro apposita struttura dati. Essendo due i possibili allarmi la struttura dati è un array di strutture.
Chiudono la rassegna le funzioni (5) e (6) per la configurazione e lettura dei registri inerenti il timestamp al power-up e power-down. Anche in questo caso le funzioni sono associate ad una apposita struttura dati più precisamente un array di strutture. In realtà la libreria prevede altre due funzioni che riguardano la gestione della memoria EEPROM. Come già discusso l'integrato ha una memoria EEPROM interna da 128 Byte scrivibile a blocchi di otto Byte alla volta. Questa memoria è liberamente scrivibile dall'utente senza nessuna restrizione. Tuttavia si può decide di proteggere delle sezioni di EEPROM al fine di preservarne il contenuto dalla sovrascrittura. Questo si può fare configurando i bit "***BP1***" e "***BP0***" del registro "***STATUS***" mappato all'indirizzo 0xFF. Si può quindi decidere di proteggere tutta la memoria, metà memoria (Da 0x40 a 0x7F) o un quarto di memoria (Da 0x60 a 0x7F). La funzione che si occupa di configurare il registro "***STATUS***" è:

```
Set_EEPROM_WriteProtection(uint8_t Section)
```

alla quale si deve passare un solo parametro che identifica quale sezione proteggere. L'ultima funzione disponibile serve per sbloccare la memoria EEPROM protetta, per intenderci quella mappata dall'indirizzo 0xF0 a 0xF7, e scrivere un byte all'indirizzo desiderato. La funzione prevede due parametri ovvero l'indirizzo in cui scrivere e il dato da scrivere. La funzione è:

```
WriteProtected_EEPROM(uint8_t RegAdd, uint8_t RegData)
```

## SKETCH COMPLETO

Per studiare le funzionalità messe a disposizione da MCP79410 abbiamo scritto uno sketch di esempio nel quale mostriamo come configurare la sezione TimeKeeper e gli allarmi 0 e 1. Lo sketch è suddiviso in sei file di cui il principale è "***MCP79410_AdvancedSettings.ino***", quindi con un doppio clic su questo si apre lo IDE Arduino e tutti i file ad esso associati. I file dello sketch sono:

- *MCP79410_AdvancedSettings* → File principale contenente le funzioni "***setup()***" e "***loop()***" tipiche di uno sketch Arduino, nonché le dichiarazioni di variabile, le costanti stringa, le costanti di tempo, le dichiarazioni delle macchine a stati ecc;
- *DigitalInput* → File di gestione degli ingressi digitali, ovvero i pulsanti P1, P2 e P3;
- *DigitalOutput* → File di gestione delle uscite digitali (il LED D4 e l'uscita "***ForceON***");
- *RTCC_Management* → File di gestione dell'integrato MCP79410 e relativa configurazione;
- *RTCC_Settings* → file per la programmazione dell'integrato MCP79410 TimeKeeper e allarmi;
- *TimersInt* → File di gestione dell'interrupt per la gestione dei timer.

Grazie al monitor seriale, attivabile dallo IDE Arduino, è possibile impartire una serie di comandi stringa per la configurazione del Timekeeper e degli allarmi 0 e 1. Inoltre durante il funzionamento "*normale*", ovvero quando non si stanno configurando i registri dell'integrato, il sistema è programmato per stampare sul monitor seriale alcune informazioni sullo stato del TimeKeeper e dei due allarmi. Lo schema di flusso del nostro sketch è illustrato nella **Fig. 7**: c'è una sequenza di inizializzazione che comprende sia gli ingressi che le uscite digitali, le macchine a stati, l'interrupt per le basi dei tempi e l'inizializzazione della libreria di gestione di MCP79410 (ciò identifica la funzione "***setup()***"). Seguono le funzioni di gestione, ovvero la lettura e il debouncing degli ingressi digitali, le macchine a stati di sistema e la funzione di lettura della configurazione assegnata ai registri di MCP79410 (ciò identifica la funzione "***loop()***").
Parliamo ora di come configurare i registri del nostro RTCC; prima di tutto è necessario collegare la scheda Arduino Uno R3 al PC tramite il connettore USB, così si assicura un'alimentazione stabile sia ad Arduino sia allo shield; in più si mette a disposizione la seriale e il relativo monitor per inviare i comandi da PC ad Arduino.
Attivato il monitor seriale si nota che il sistema esegue una lettura della configurazione dei registri ogni quindici secondi; la **Fig. 8** ne mostra il risultato. In testa, il sistema evidenzia che l'oscillatore del RTCC è attivo e mostra data e ora correnti. Seguono le configurazioni dei due allarmi:

1) stato dell'allarme = Abilitato/Disabilitato;
2) polarità uscita = livello logico alto/basso;
3) Flag interrupt = indica se è scattato l'allarme a seconda della sua configurazione;
4) ora impostata = mostra l'ora configurata per l'allarme;
5) data impostata = mostra la data impostata per l'allarme (l'anno perde di significato in quanto irrilevante per la gestione degli allarmi);
6) maschera = mostra la maschera di confronto per la generazione dell'interrupt a seconda della configurazione sopra esposta.

A questo punto possiamo decidere di fare due cose: configurare la data e l'ora dell'RTCC e quindi, se è la prima programmazione, attivare di conseguenza l'oscillatore e la gestione della alimentazione da batteria; la seconda è configurare gli allarmi 0 e 1.
Cominciamo col descrivere la configurazione di data e ora del RTCC: prima di tutto si deve mandare il sistema in configurazione di data e ora e per fare ciò si deve premere il pulsante P1 per più di due secondi; LD4 emetterà un lampeggio a indicare l'avvenuto riconoscimento della pressione del tasto. Entrati nella configurazione, LD4 lampeggia e sul monitor seriale viene stampata una stringa a indicare che il sistema è entrato in configurazione di data e ora. Per uscire da questa modalità, premere nuovamente il pulsante P1 per più di due secondi; anche in questo caso una stringa indicherà quanto appena fatto (**Fig. 9**).
Il primo passo è configurare la data e il giorno della settimana: allo scopo bisogna scrivere la

## Digital **Trimming**

Per correggere l'imprecisione introdotta dall'oscillatore esterno si può sfruttare il registro ***"OSCTRIM"*** il quale può introdurre un fattore correttivo fino a ±129ppm, per sfruttare questa funzione è obbligatorio resettare il bit ***"CRSTRIM"*** del registro ***"CONTROL"***. Il registro ***"OSCTRIM"*** può anche essere sfruttato per correggere gli errori dell'oscillatore dovuti alle variazioni di temperatura. Il bit più significativo del registro indica il segno del valore correttivo mentre i restanti bit (***"TRIMVAL"***) contengono il valore da aggiungere o sottrarre. Ad ogni valore corrispondono due impulsi di clock che vengono aggiunti o sottratti ogni minuto. Se invece si ha il bit "OSCTRIM" settato si ha che il valore indicato viene aggiunto o sottratto 128 volte al secondo.
Il valore da caricare in ***"TRIMVAL"*** deve essere calcolato e per fare ciò ci sono due metodi. Il primo consiste nel misurare la frequenza di clock riportata sul pin di uscita ***"MFP"*** e calcolare di quanto è spostata rispetto al valore aspettato. Il secondo metodo consiste nel verificare l'ammontare di secondi guadagnati o persi durante un periodo di tempo. Con il primo metodo, l'equazione per calcolare il valore di ***"TRIMVAL"*** è:

$$TRIMVAL = \frac{\frac{(F_{IDEAL} - F_{MEAS}) * 32768}{F_{IDEAL}} * 60}{2}$$

Dove ***"$F_{IDEAL}$"*** è la frequenza che ci si aspetta di misurare mentre ***"$F_{MEAS}$"*** è la frequenza realmente misurata. Per quanto riguarda il secondo metodo si deve usare le seguenti formule:

$$PPM = \frac{SecDeviation}{ExpectedSec} * 1000000$$

$$TRIMVAL = \frac{PPM * 32768 * 60}{2000000}$$

Dove ***"ExpectedSec"*** sono i secondi che ci si aspetta di leggere in un dato periodo, per eseguire una calibrazione accurata lavorare su lunghi periodi. Mentre ***"SecDeviation"*** è il numero di secondi guadagnati o persi durante il periodo.

stringa di comando seguente, sfruttando l'apposita textbox del monitor seriale e cliccare sul pulsante "*Invia*":

```
DateSet: 12/12/2015 - SAT
```

dove "`DateSet:`" è il comando da attuare e "`12/12/2015 - SAT`" sono la data e il giorno della settimana. Il giorno della settimana è in inglese (`MON, TUE, WED, THU, FRI, SAT, SUN`).
Nel caso in cui il comando contenga degli errori il sistema ritornerà una stringa di avviso invitando l'utente ad editare nuovamente il comando.
Il secondo passo prevede di configurare l'ora, quindi digitare il seguente comando:

```
TimeSet: 14:30:10
```

Dove "`TimeSet:`" è il comando da attuare e "`14:30:10`" è l'orario (formato 24h) da impostare. Se invece si vuole impostare lo stesso orario ma nel formato 12h, si deve scrivere la seguente:

```
TimeSet: 02:30:10 - PM
```

Il sistema riconosce il formato desiderato e configura di conseguenza i registri del RTCC.
Passiamo ora alla configurazione degli allarmi 0 e 1: per attivare la configurazione degli allarmi, premere per più di due secondi il pulsante P2, anche in questo caso LD4 lampeggerà e a monitor verrà stampata una apposita stringa sia per l'attivazione che disattivazione dell'impostazione degli allarmi (**Fig. 10**). Come primo passo dovete configurare la data, la maschera e la polarità dell'allarme 0 o 1. Supponiamo di lavorare con l'allarme 0:

```
DateSetAlarm(0): 12/12/2015 - SAT - SecondsMatch - LHL
```

dove "`DateSetAlarm(0):`" è il comando di configurazione della data per l'allarme 0. Il terzo parametro è la maschera di confronto, che può essere impostata per lavorare solo sui secondi, solo sui minuti, solo sulle ore, solo sui giorni della settimana, solo sulla data, oppure imporre il confronto su tutti i parametri contemporaneamente. Nel nostro esempio scegliamo che il confronto debba avvenire solo sui secondi; gli altri a disposizione sono "`MinutesMatch`", "`HoursMatch`", "`DayOfWeekMatch`", "`DateMatch`" e "`AllMatch`". Il quarto parametro serve per selezionare la polarità dell'uscita dove "`LHL`" indica polarità positiva mentre "`LLL`" polarità negativa. Il secondo passo serve per configurare l'ora:

```
TimeSetAlarm(0): 14:30:10
```

dove "`TimeSetAlarm(0):`" è il comando di configurazione dell'ora per l'allarme 0. In questo caso, per congruenza con la configurazione dell'ora del RTCC il formato è 24h. Se si volesse impostare il formato 12h, la stringa diventerebbe:

```
TimeSetAlarm(0): 02:30:10 - PM
```

Come per la configurazione del RTCC il

sistema riconosce automaticamente il formato desiderato. Va da sè che se si volesse impostare l'allarme 1 i comandi sono identici ai precedenti ma con riferimento all'allarme 1; ad esempio `DateSetAlarm(1):.....` ecc. Volendo è anche possibile resettare gli allarmi impostati, ovvero cancellare completamente la loro programmazione, disabilitandoli. Per fare ciò basta impartire il comando:

`ResetAlarm(0):` oppure `ResetAlarm(1):`

Se invece si desidera semplicemente disabilitare un allarme usare:

`DisableAlarm(0):` oppure `DisableAlarm(1):`

Per abilitarlo, il comando è:

`EnableAlarm(0):` oppure `EnableAlarm(1):`

Tutte le stringhe, sia di comando che di avviso, sono memorizzate nella memoria Flash del microcontrollore Atmel al fine di risparmiare la memoria SRAM; infatti ricordate che tutte le volte che si utilizza la funzione "***println***" con del testo da stampare questo viene caricato in SRAM, occupando spazio prezioso.
I comandi sopra indicati servono per configurare le strutture dati di cui abbiamo parlato durante la descrizione della libreria e successiva programmazione dei registri; per questo le varie programmazioni sono divise in due step, di cui è sempre il secondo che congela i dati nei registri dell'RTCC. Ad esempio, durante la configurazione di data e ora dell'RTCC (TimeKeeper) una volta ricevuti i dati corretti il sistema richiama la funzione di programmazione seguente:

```
mcp79410.WriteTimeKeeping(0);
```

Il parametro "0" indica che si è selezionato il formato 24h. Invece durante la configurazione degli allarmi si useranno le seguenti:

```
mcp79410.WriteAlarmRegister(AlarmIndex, 0);
mcp79410.Alarm0Bit(1);
```

La prima funzione salva i dati nei registri per l'allarme n, dipende dal valore della variabile "***AlarmIndex***", 0 per l'allarme 0 e 1 per l'allarme 1, con formato 24h; la seconda attiva l'allarme 0. Configurando opportunamente i due allarmi

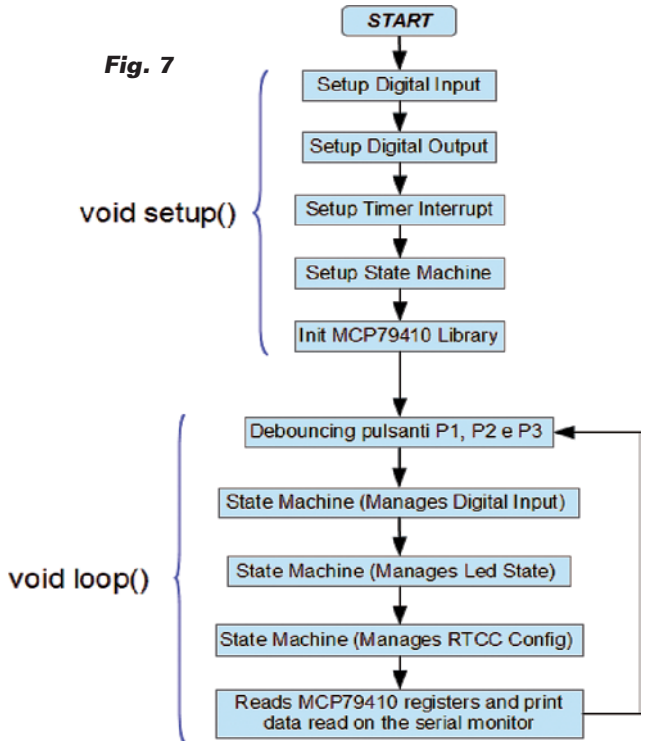



***Fig. 7***

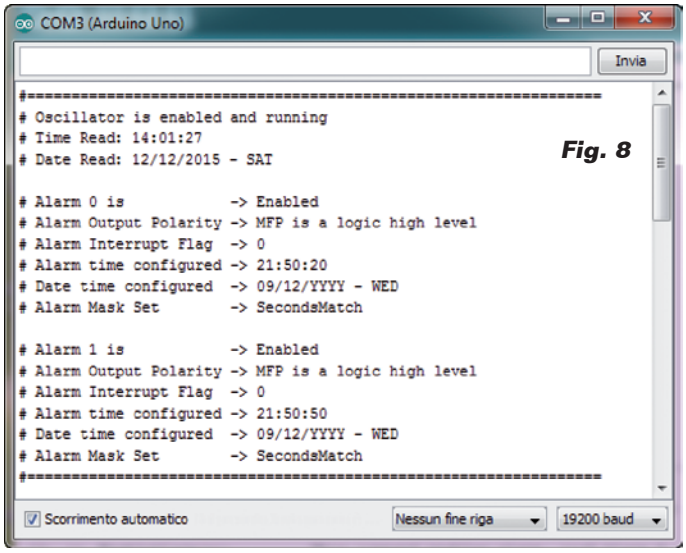



***Fig. 8***

si arriva ad avere una situazione di lavoro dove l'allarme 0 forza l'accensione e l'allarme 1 lo spegnimento. Chiariamo meglio quanto appena detto: una volta configurati i registri dei due allarmi si deve scollegare il cavo USB dal PC togliendo quindi alimentazione a tutta l'elettronica compreso lo shield. Poi si deve fornire alimentazione a quest'ultimo tramite il connettore microUSB assicurandosi che il jumper J1 sia in posizione 2-3. A questo punto l'elettronica a valle del MOSFET Q1 verrà alimentata solo se si attiverà l'interrupt dell'allarme 0 intervenendo sull'uscita MFP; in tal

```
#============================================================
# TimeKeeper settings in progress. System wait a command to set date and time
#============================================================

#------------------------------------------------------------
# TimeKeeper settings has been stopped
#============================================================
```

*Fig. 9*

caso lo sketch si accorge che il flag dell'interrupt dell'allarme 0 è andato a "*1*" logico e quindi forza alto il segnale "*ForceOn*" mantenendo sempre accesa l'alimentazione indifferentemente dallo stato dell'uscita MFP. Il sistema rimane quindi in attesa che venga intercettato l'evento per l'allarme 1, quando questo accade lo sketch se ne accorge e quindi forza basso il segnale "*ForceOn*" spegnendo automaticamente tutta l'elettronica. Tutte le volte che il sistema rileva che è scattato un allarme dovrà lui stesso provvedere al reset del flag corrispondente seguendo una apposita procedura. Per i dettagli vedere il codice dello sketch in particolare le due funzioni:

```
void ResetAlarmFlag_0(void)
void ResetAlarmFlag_1(void)
```

Per vedere questo andamento perpetuo di accensione e spegnimento automatici dell'alimentazione è sufficiente configurare i due allarmi a fare il confronto solo sui secondi, come evidenziato prima, e impostare i secondi dell'ora in modo appropriato ad esempio: 12:10:10 per l'allarme 0 (Accensione) e 12:10:40 per l'allarme 1 (Spegnimento). Così facendo l'elettronica rimane accesa per 30 secondi e spenta per altrettanti 30. Ovviamente questo è solo uno sketch di esempio e nei casi reali si vorrebbe che la scheda si accendesse a una determinata ora della giornata per poi spegnersi ad un'altra. Quindi si potrebbe pensare di configurare il confronto non sui secondi ma sulle ore e impostare ad esempio l'ora come segue: 08:00:00 per l'allarme 0 (Accensione) e 18:00:00 per l'allarme 1 (Spegnimento). Così per tutti i i giorni dell'anno si avrebbe una accensione e uno spegnimento alle ore desiderate. Le possibili combinazioni a disposizione sono veramente tante e modificando il codice dello sketch in modo opportuno si può rendere tutto più dinamico. In altre parole è il sistema a decidere quando avverrà la prossima accensione o spegnimento a seconda dei casi; dipende solo dalla fantasia dell'utente e dal codice implementato. La nostra libreria e la nostra elettronica servono solo per darvi il giusto mezzo per realizzare le vostre più disparate applicazioni.

*Fig. 10*

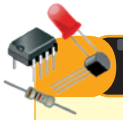
```
#============================================================
# Alarms settings in progress. System wait a command to set alarm 0/1
#============================================================

#============================================================
# Alarms settings has been stopped
#============================================================
```

## REALIZZAZIONE PRATICA

Spendiamo ora qualche parola sulla costruzione dello shield, che richiede un circuito stampato a doppia ramatura le cui tracce sono disponibili sul nostro sito *www.elettronicain.it*. Lo shield è composto praticamente tutto da componenti SMD; fanno eccezione il portapile, gli strip maschio e femmina e i pulsanti. Per il montaggio dotatevi di un saldatore da 20W a punta molto fine, di pasta flussante, di una pinzetta e una lente d'ingrandimento per posizionare i componenti e verificare le saldature. I primi componenti da montare sono il MOSFET Q1 e l'U1, da posizionare con i pin al centro delle rispettive piazzole e stagnare un pin per lato; seguono tutti i passivi, il fusibile e poi il quarzo e il connettore micro USB. Fatto ciò si passa al montaggio dei pulsanti e del portapila, quindi si inseriscono e si saldano i pin-strip laterali per Arduino e quello per Raspberry Pi. Per l'orientamento dei componenti polarizzati fate riferimento al piano di montaggio che trovate nelle pagine precedenti. Completate le saldature e verificate con la lente che non siano cortocircuiti, inserite la pila CR2032.

## CONCLUSIONI

In queste pagine abbiamo presentato lo shield basato sull'MCP79410 e lo sketch di gestione per Arduino. Nella prossima puntata descriveremo degli sketch specifici per singole funzioni e affronteremo l'applicazione e il software per l'utilizzo dello shield con la Raspberry Pi.



## per il MATERIALE

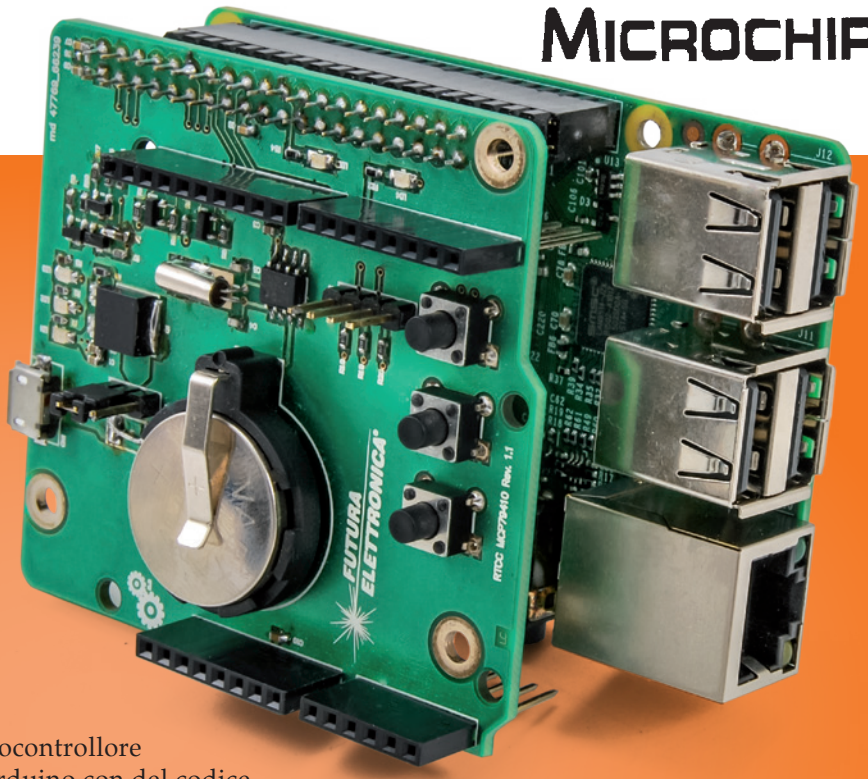
Lo shield RTC Raspberry-Arduino (cod. FT1254M) viene venduto montato e collaudato ed è disponibile presso Futura Elettronica al prezzo di Euro 20,00. Il prezzo si intende IVA compresa.

Il materiale va richiesto a:
Futura Elettronica, Via Adige 11, 21013 Gallarate (VA)
Tel: 0331-799775 • Fax: 0331-792287 - www.futurashop.it


# RTC SHIELD PER ARDUINO E RASPBERRY PI

di MATTEO DESTRO

**Concludiamo il discorso su Arduino e vediamo come utilizzare lo shield con Raspberry Pi mediante un'apposita libreria. Seconda e ultima puntata.**




Ci siamo lasciati dopo aver descritto un nuovo shield RTCC basato sull'integrato Microchip MCP79410 ed aver esaminato lo sketch per Arduino; ora, come anticipato nella prima puntata, completiamo il discorso su Arduino e passiamo "a bomba" all'applicazione con la Raspberry Pi.
Iniziamo presentandovi quattro mini sketch molto semplici e minimali per acquisire familiarità con la nostra libreria e imparare ad usarla. Questi ulteriori sketch vi aiuteranno a studiare e comprendere meglio lo sketch completo descritto il mese scorso e li abbiamo preparati come alternativa per chi desidera gestire solo alcune funzioni dell'integrato Microchip e non desidera riempire la memoria del microcontrollore di Arduino con del codice non utilizzato.

**SKETCH PER TIMEKEEPR**

Veniamo allo sketch per configurare solo il TimeKeeper del nostro RTCC. Anche in questo caso abbiamo suddiviso per semplicità lo sketch su tre file distinti in particolare abbiamo:

- *MCP79410_SetTimeKeeper* → file principale contenente le funzioni "***setup()***" e "***loop()***" nonché tutte le dichiarazioni di variabili, le costanti stringa, le costanti di tempo, le dichiarazioni delle macchine a stati ecc.;

- *RTCC_Settings* → file per la configurazione del TimeKeeper;
- *TimersInt* → file di gestione dell'interrupt per il controllo dei timer.

Concentriamoci sul file "***RTCC_Settings***" e in particolare sulla funzione "`void RTCC_TimeKeeperSettings(void)`" la quale ci permette di configurare la data e l'ora del nostro RTCC. Questa funzione viene richiamata solo durante l'esecuzione del "***setup()***" e mai in altra occasione. La funzione lavora sul formato 24h, ma volendo si può modificare per lavorare sul formato 12h; infatti ci sono delle parti di codice commentate che servono appunto per la configurazione dei registri nel secondo formato.
La funzione è impostata per configurare come data il Venerdì del 25/12/2015 e per fare ciò carica nella struttura dati del TimeKeeper i nuovi valori della data e il giorno della settimana, tenendo ben presente il formato, BCD dopodiché carica nella struttura dati i nuovi valori per l'ora nel formato 24h. L'ora scelta per l'esempio è 14:25:30. Infine, sempre della struttura dati, si attiva l'oscillatore agendo sull'apposito bit e si attiva la modalità batteria nel caso in cui venga a mancare l'alimentazione principale.
A questo punto per rendere effettive le modifiche si deve richiamare la funzione per programmare il RTCC e in particolare i registri del TimeKeeper:

```
mcp79410.WriteTimeKeeping(0);
```

Il parametro 0 indica che si sta lavorando in formato 24h. Durante l'esecuzione del "***loop()***" viene richiamata la sola funzione di lettura dei registri del TimeKeeper, una volta ogni 10 secondi, i quali vengono decodificati e stampati sul monitor seriale con la stessa filosofia adottata dal precedente sketch. Si può quindi vedere l'avanzare dei secondi e dei minuti. Tutte le volte che lo sketch verrà fatto ripartire, si avrà una riconfigurazione di data e ora con i valori detti prima.

## SKETCH PER ALLARME 0

Vediamo ora uno sketch che riguarda semplicemente come configurare gli allarmi del nostro RTCC. Anche in questo caso abbiamo suddiviso per semplicità lo sketch su tre file distinti:

- *MCP79410_SetAlarm*→ file principale contenente le funzioni "***setup()***" e "***loop()***" nonché tutte le dichiarazioni di variabili, le costanti stringa, le costanti di tempo, le dichiarazioni delle macchine a stati ecc.;
- *RTCC_Settings* → file per la configurazione degli allarmi;
- *TimersInt* → file di gestione dell'interrupt per la gestione dei timer.

Concentriamoci sul file "***RTCC_Settings***" dove incontriamo nuovamente la funzione "`void RTCC_TimeKeeperSettings(void)`", vista nello sketch precedente, per la configurazione di data e ora e la funzione "`void RTCC_AlarmSettings(void)`" per la configurazione dell'allarme 0; per semplicità abbiamo omesso la configurazione dell'allarme 1.
La funzione di configurazione dell'allarme inizia disabilitando entrambi gli allarmi per poi passare a configurare la struttura dati apposita; questa è un array di strutture con dimensione 2, ossia allarme 0 e allarme 1. Per prima cosa si deve configurare la data omettendo l'anno che perde di significato nella configurazione degli allarmi. Al passo successivo si configura la maschera di confronto, noi scegliamo di eseguire il confronto solo sui secondi (le righe commentate mostrano che valori assegnare nel caso in cui si decidesse di utilizzare una diversa maschera di confronto).
Segue la configurazione della polarità dell'uscita MFP e per ultimo si imposta l'ora ovviamente nel formato 24h congruente con la data e l'ora impostata per il TimeKeeper.
Per rendere effettive le modifiche apportate alla struttura dati richiamare in sequenza le funzioni:

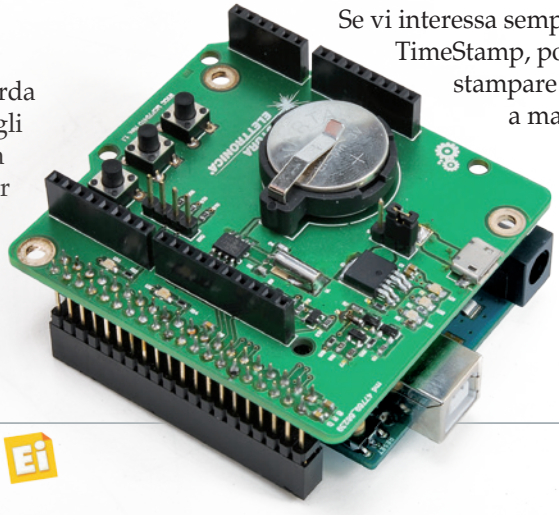
```
mcp79410.WriteAlarmRegister(0, 0);
mcp79410.Alarm0Bit(1);
```

dove la prima invia i dati appena inseriti ai registri di configurazione dell'allarme 0 mentre la seconda abilita l'allarme stesso.

## SKETCH PER LEGGERE IL TIMESTAMP

Se vi interessa semplicemente leggere i registri TimeStamp, power-up e power-down, per stampare a monitor quando è venuta a mancare l'alimentazione e quando è ritornata, usate lo sketch descritto qui di seguito, che è suddiviso in due file:

- *MCP79410_timeStamp* → file principale contenente le funzioni "***setup()***" e "***loop()***"



nonché tutte le dichiarazioni di variabili, le costanti stringa, le costanti di tempo, le dichiarazioni delle macchine a stati e le funzioni di gestione dei TimeStamp;
- *TimersInt* → file di gestione dell'interrupt per la gestione dei timer.

In questo sketch non si eseguono configurazioni ma semplicemente si va a leggere il contenuto dei registri del timestamp. Questo è utile per stampare a video data e ora di quando è venuta a mancare l'alimentazione della scheda e data e ora di quando è ritornata. Durante il "***setup()***" non ci sono configurazioni particolari, carichiamo soltanto un timer per ritardare la lettura dei registri in modo da dare tempo di attivare il monitor seriale dopo il collegamento del cavo USB alla scheda Arduino UNO R3. La funzione che si occupa di leggere i registri viene eseguita una volta ogni dieci secondi, allo scadere del timer vengono letti sia i registri di data e ora correnti e, se il bit "***PwFail***" è a uno logico anche i registri del TimeStamp in quanto c'è stata sicuramente una mancanza di alimentazione. Tutte queste informazioni vengono poi stampate sul monitor seriale con la consueta formattazione. In questo sketch si sfruttano le funzioni di libreria:

```
ReadTimeKeeping(void)
ReadSingleReg(uint8_t ControlByte, uint8_t RegAdd)
ReadPowerDownUpRegister(uint8_t PowerDownUp)
ResetPwFailBit(void)
```

## SKETCH PER GESTIRE LA EEPROM

Vediamo in ultimo lo sketch per utilizzare la EEPROM del nostro RTCC, sia quella protetta (8 Byte) che non protetta (128 Byte). Lo sketch è suddiviso in due file:
- *MCP79410_EEPROM* → file principale contenente le funzioni "***setup()***" e "***loop()***" nonché tutte le dichiarazioni di variabili, le costanti stringa, le costanti di tempo, le dichiarazioni delle macchine a stati e le funzioni di gestione della EEPROM;
- *TimersInt* → file di gestione dell'interrupt per la gestione dei timer.

Per testare la memoria EEPROM da 128 Byte proviamo a scrivere in memoria la scritta "***ElettronicaIn***" partendo dall'indirizzo 0x00, sfruttando la libreria e in particolare la funzione di scrittura:

```
WriteSingleReg(uint8_t ControlByte, uint8_t RegAdd, uint8_t RegData)
```

Sfruttando l'array presente nella nostra libreria e scrivendo un piccolo ciclo do-while, riusciamo a scrivere la nostra stringa nella memoria EEPROM. Quanto esposto è tutto raccolto in una funzione nominata "`void Set_EEPROM(void)`" la quale viene chiamata solo durante l'esecuzione del "***setup()***". Per verificare l'avvenuta scrittura abbiamo scritto una piccola funzione "`void Read_EEPROM(void)`" per rileggere, ogni dieci secondi, il contenuto delle prime locazioni di memoria EEPROM per poi stamparne il risultato sul monitor seriale. Anche in questo caso la funzione è composta da poche righe di codice compreso un ciclo do-while per la stampa. La funzione di libreria utilizzata per leggere la EEPROM è:

```
ReadArray(uint8_t ControlByte, uint8_t StartAdd, uint8_t Lenght)
```

Oltre alla memoria EEPROM classica abbiamo testato la scrittura e lettura della memoria EEPROM protetta (8 Byte) per memorizzare un ipotetico MAC address. Anche in questo caso abbiamo scritto due piccole funzioni, la prima per scrivere in EEPROM e la seconda per leggere la EEPROM. Quindi con "`void Set_Protected_EEPROM(void)`" scriviamo la memoria EEPROM solo durante il "***setup()***" e poi ogni dieci secondi andiamo a rileggerne il contenuto con la funzione "`void Read_Protected_EEPROM(void)`".
Per la scrittura si usa la funzione di libreria:

```
WriteProtected_EEPROM(uint8_t RegAdd, uint8_t RegData)
```

Si può scrivere un solo byte alla volta, anche in questo caso si sfrutta l'array della nostra libreria più un piccolo ciclo do-while.
Bene, detto ciò abbiamo concluso il discorso sull'applicazione con Arduino.

## ED ORA...RASPBERRY PI

Lo shield RTCC dispone di una zoccolatura doppia che ne permette l'inserzione di volta in volta sulla scheda desiderata. Dopo avervi descritto il firmware per abbinarla ad Arduino, in quest'ultima puntata mostreremo come utilizzare il nostro shield RTCC con Raspberry Pi 2.0/B+ sfruttando la libreria in codice Python sviluppata per l'evenienza. Prima di procedere riepiloghiamo alcuni dettagli dello shield, che si basa sull'integrato Microchip MCP79410; nello specifico, diamo uno sguardo alle interconnessioni tra esso e la Raspberry Pi. A tale scheda vengono collegate diverse linee tra cui il bus di comunicazione $I^2C$ per la gestione dell'integrato U1 (tra Raspberry Pi e l'integrato U1 è interposto un traslatore di livello meglio descritto nella prima puntata, in quanto Raspberry Pi lavora con livello di tensione +3,3V mentre U1 con livello

+5V), le linee dei pulsanti P1, P2 e P3, le linee di trigger, il LED e la linea di forzatura ON. Se necessario, il bus I²C, oltre a essere portato all'integrato U1, può essere prolungato verso altre schede connesse a Raspberry Pi tramite il solito connettore di espansione: l'importante è che non ci siano delle sovrapposizioni con gli indirizzi hardware già impegnati. Quindi, riassumendo, i seguenti pin della Raspberry Pi hanno questo utilizzo:

- GPIO 2 e 3 sono usati per collegare il bus I²C;
- GPIO 17, 18 e 27 sono usati per collegare i tre pulsanti utilizzati dallo sketch realizzato in codice Python;
- GPIO 22 e 23 sono usati per collegare le due linee di trigger utili durante le fasi di debug;
- GPIO 24 è usato per gestire la linea di forzatura ON dell'elettronica tramite il transistor NPN Q6 e il relativo MOSFET Q1, usato come interruttore;
- GPIO 25 serve per pilotare il LED LD5.

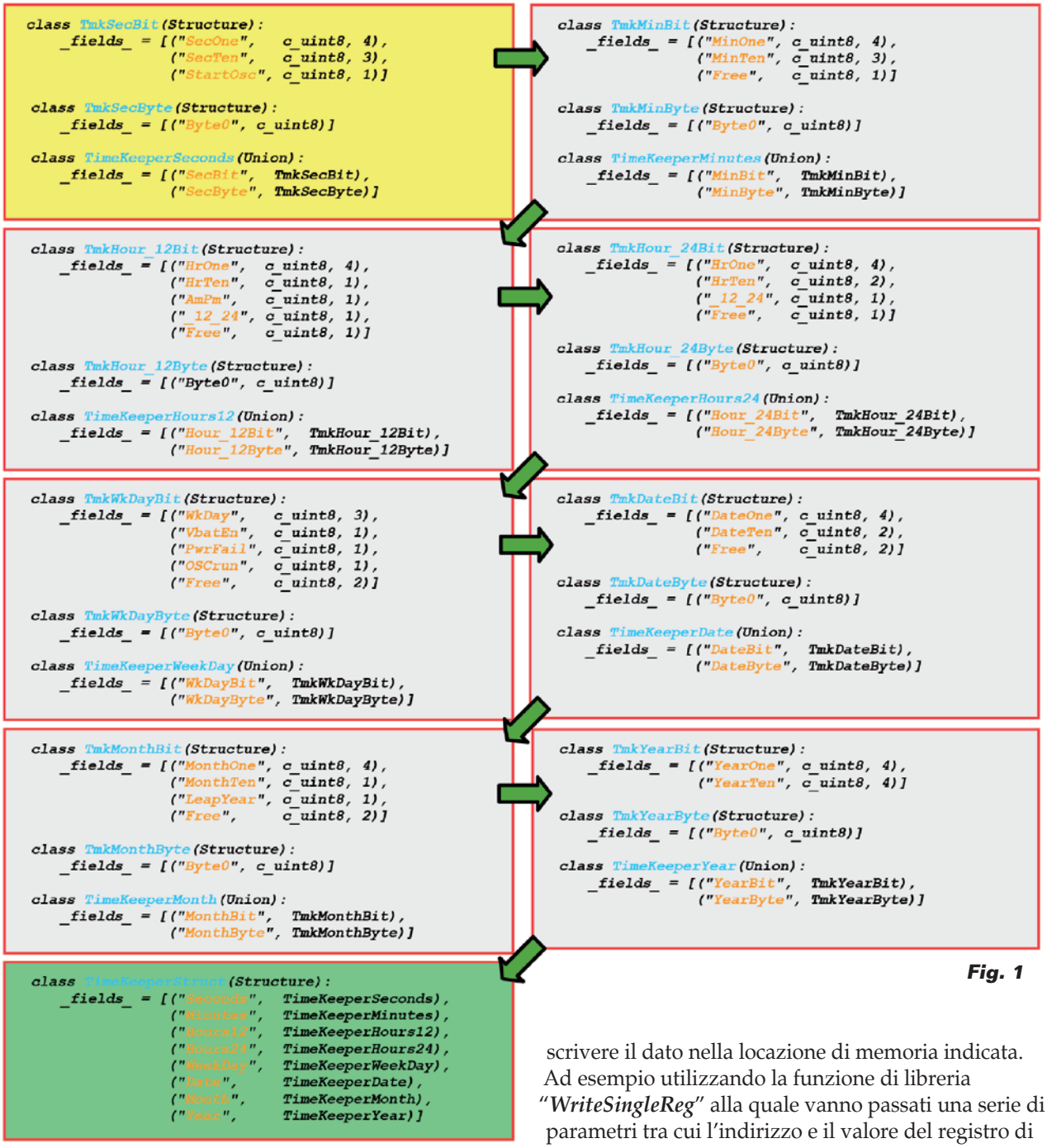
Concludiamo qui la descrizione dell'hardware messo a disposizione della nostra scheda, se ritenete opportuno approfondire l'argomento vi invitiamo a rileggere la precedenta puntata per i dettagli.

## LIBRERIA PER RASPBERRY PI

La libreria sviluppata per l'occasione permette di configurare e gestire l'integrato MCP79410 in tutte le sue funzioni rendendo agevole configurarlo per le proprie esigenze, che possono essere diverse da quelle sviluppate nel nostro sketch di esempio di cui discuteremo più avanti.
Diversamente dal mondo Arduino la libreria è scritta in Python ed è composta da soli due file con estensione "***.py***".
Il primo si chiama "***MCP79410.py***" e contiene tutte le funzioni di libreria mentre il secondo è "***MCP79410_DefVar.py***" e contiene la dichiarazione di tutte le costanti e le strutture dati. I due file fanno quindi parte di un pacchetto che può essere installato sulla vostra Raspberry Pi e utilizzato nei vostri sketch. Vedremo in seguito come viene generato il pacchetto da distribuire e come si deve fare per installarlo nella propria distribuzione.
Oltre ai due file di libreria abbiamo scritto uno sketch per mostrare come utilizzare le funzioni implementate. Lo sketch è composto da tre file distinti: "***MCP79410_Main.py***" dove trova posto il main, "***MCP79410_PrintFunc.py***" dove troviamo tutte le funzioni di stampa a video dei dati letti dall'integrato MCP79410 e infine "***MCP79410_SetRegisters.py***", dove sono memorizzate le funzioni di configurazione dell'integrato MCP79410.

Cominciamo a descrivere a grandi linee il file "***MCP79410_DefVar.py***" il quale, come detto sopra, contiene le definizioni di tutte le funzioni utilizzate, nonché la dichiarazione delle variabili e delle struttura dati pubbliche e private. Da una prima occhiata si può vedere che è una copia quasi identica del relativo file realizzato per Arduino con alcune differenze dovute al differente linguaggio di programmazione utilizzato.
In testa al file ci sono le dichiarazioni delle costanti come ad esempio gli indirizzi hardware dell'integrato, ricordiamo che MCP79410 possiede due indirizzi hardware distinti, oppure gli indirizzi dei registri a cui puntare per la lettura/programmazione dei parametri operativi dell'integrato. Seguono una serie di costanti per la programmazione dei registri con dei valori standard, questa sezione, associata alla lettura attenta del data-sheet, aiuta a comprendere appieno come si deve fare e cosa si deve configurare per il corretto funzionamento dell'integrato. Tutte le costanti inserite possono essere modificate a piacere dall'utente a seconda delle proprie esigenze; per ogni costante è presente un commento utile a comprenderne il significato.
Infine trovano posto le costanti per la gestione della EEPROM; ricordiamo che per accedere ad essa l'indirizzo hardware da utilizzare è diverso da quello del RTCC. Oltre alle costanti appena descritte trovano parte le strutture dati fondamentali per una gestione nidificata dei dati per la programmazione/lettura dei vari registri presenti.
Le strutture dati sono suddivise in sezioni distinte, in particolare abbiamo la sezione per il TimeKeepr, la sezione per gli allarmi e la sezione per i TimeStamp. Come per il mondo Arduino, è possibile configurare i registri in due modi distinti: un metodo prevede l'utilizzo di apposite funzioni dedicate l'altro tramite la configurazione delle strutture dati e successiva programmazione dei registri. Ad esempio la struttura dati seguente permette di configurare ogni singolo bit del registro "***CONTROL***":

```
class CtrlBit(Structure):
      _fields_ = [("SquareWaveFreqOutput",    c_uint8,
2),
                  ("CoarseTrimEnable",        c_uint8, 1),
                  ("ExtOscInput",             c_uint8, 1),
                  ("Alarm0_Enable",           c_uint8, 1),
                  ("Alarm1_Enable",           c_uint8, 1),
                  ("SquareWaveOutputEnable", c_uint8, 1),
                  ("LogicLevelOutput",        c_uint8, 1)]

class CtrlByte(Structure):
      _fields_ = [("Byte0", c_uint8)]

class CtrlReg(Union):
      _fields_ = [("ControlBit",  CtrlBit),
                  ("ControlByte", CtrlByte)]
```



```
class TmkSecBit(Structure):
    _fields_ = [("SecOne",   c_uint8, 4),
                ("SecTen",   c_uint8, 3),
                ("StartOsc", c_uint8, 1)]

class TmkSecByte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperSeconds(Union):
    _fields_ = [("SecBit",  TmkSecBit),
                ("SecByte", TmkSecByte)]
```

```
class TmkMinBit(Structure):
    _fields_ = [("MinOne", c_uint8, 4),
                ("MinTen", c_uint8, 3),
                ("Free",   c_uint8, 1)]

class TmkMinByte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperMinutes(Union):
    _fields_ = [("MinBit",  TmkMinBit),
                ("MinByte", TmkMinByte)]
```

```
class TmkHour_12Bit(Structure):
    _fields_ = [("HrOne",  c_uint8, 4),
                ("HrTen",  c_uint8, 1),
                ("AmPm",   c_uint8, 1),
                ("_12_24", c_uint8, 1),
                ("Free",   c_uint8, 1)]

class TmkHour_12Byte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperHours12(Union):
    _fields_ = [("Hour_12Bit",  TmkHour_12Bit),
                ("Hour_12Byte", TmkHour_12Byte)]
```

```
class TmkHour_24Bit(Structure):
    _fields_ = [("HrOne",  c_uint8, 4),
                ("HrTen",  c_uint8, 2),
                ("_12_24", c_uint8, 1),
                ("Free",   c_uint8, 1)]

class TmkHour_24Byte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperHours24(Union):
    _fields_ = [("Hour_24Bit",  TmkHour_24Bit),
                ("Hour_24Byte", TmkHour_24Byte)]
```

```
class TmkWkDayBit(Structure):
    _fields_ = [("WkDay",   c_uint8, 3),
                ("VbatEn",  c_uint8, 1),
                ("PwrFail", c_uint8, 1),
                ("OSCrun",  c_uint8, 1),
                ("Free",    c_uint8, 2)]

class TmkWkDayByte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperWeekDay(Union):
    _fields_ = [("WkDayBit",  TmkWkDayBit),
                ("WkDayByte", TmkWkDayByte)]
```

```
class TmkDateBit(Structure):
    _fields_ = [("DateOne", c_uint8, 4),
                ("DateTen", c_uint8, 2),
                ("Free",    c_uint8, 2)]

class TmkDateByte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperDate(Union):
    _fields_ = [("DateBit",  TmkDateBit),
                ("DateByte", TmkDateByte)]
```

```
class TmkMonthBit(Structure):
    _fields_ = [("MonthOne", c_uint8, 4),
                ("MonthTen", c_uint8, 1),
                ("LeapYear", c_uint8, 1),
                ("Free",     c_uint8, 2)]

class TmkMonthByte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperMonth(Union):
    _fields_ = [("MonthBit",  TmkMonthBit),
                ("MonthByte", TmkMonthByte)]
```

```
class TmkYearBit(Structure):
    _fields_ = [("YearOne", c_uint8, 4),
                ("YearTen", c_uint8, 4)]

class TmkYearByte(Structure):
    _fields_ = [("Byte0", c_uint8)]

class TimeKeeperYear(Union):
    _fields_ = [("YearBit",  TmkYearBit),
                ("YearByte", TmkYearByte)]
```

```
class TimeKeeperStruct(Structure):
    _fields_ = [("Seconds",  TimeKeeperSeconds),
                ("Minutes",  TimeKeeperMinutes),
                ("Hours12",  TimeKeeperHours12),
                ("Hours24",  TimeKeeperHours24),
                ("WeekDay",  TimeKeeperWeekDay),
                ("Date",     TimeKeeperDate),
                ("Month",    TimeKeeperMonth),
                ("Year",     TimeKeeperYear)]
```

***Fig. 1***

Supponendo di volere abilitare l'allarme 0 si dovrà settare il bit "*Alarm0_Enable*" con la seguente sintassi:

```
MCP79410._CtrlReg.ControlBit.Alarm0_Enable = 1;
```

dove "*_CtrlReg*" è la dichiarazione globale della struttura dati in esame visibile dallo sketch. Ovviamente così facendo abbiamo solo settato un valore in RAM. Il passo successivo sarà quello di richiamare un'apposita funzione per andare a scrivere il dato nella locazione di memoria indicata. Ad esempio utilizzando la funzione di libreria "*WriteSingleReg*" alla quale vanno passati una serie di parametri tra cui l'indirizzo e il valore del registro di cui si vuole modificare il valore.
La sintassi completa sarà:

```
mcp79410.WriteSingleReg(RTCC_HW_ADD, CONTROL_ADD,
MCP79410._CtrlReg.ControlByte);
```

Oltre alla struttura dati appena illustrata ne seguono delle altre, più articolate, che permettono la configurazione dei registri TimeKeeper, gli allarmi e la lettura dei TimeStamp al PowerUp e PowerDown. A titolo d'esempio vediamo la struttura dati per la gestione dei registri TimeKeeper come già visto per

## Listato 1

```
1)  ToggleSingleBit(ControlByte, RegAdd, Bit)
2)  SetSingleBit(ControlByte, RegAdd, Bit)
3)  ResetSingleBit(ControlByte, RegAdd, Bit)
4)  WriteSingleReg(ControlByte, RegAdd, RegData)
5)  WriteArray(ControlByte, StartAdd, Lenght)
6)  ClearReg(ControlByte, RegAdd)
7)  ReadSingleReg(ControlByte, RegAdd)
8)  ReadArray(ControlByte, StartAdd, Lenght)
9)  GeneralPurposeOutputBit(SetReset)
10) SquareWaveOutputBit(EnableDisable)
11) Alarm1Bit(EnableDisable)
12) Alarm0Bit(EnableDisable)
13) ExternalOscillatorBit(EnableDisable)
14) CoarseTrimModeBit(EnableDisable)
15) SetOutputFrequencyBit(OutputFreq)
16) StartOscillatorBit(EnableDisable)
17) Hour12or24TimeFormatBit(SetHourType)
18) AmPmBit(SetAmPm)
19) VbatEnBit(EnableDisable)
20) AlarmHour12or24TimeFormatBit(SetHourType, Alarm0_1)
21) AlarmAmPmBit(SetAmPm, Alarm0_1)
22) AlarmIntOutputPolarityBit(SetReset, Alarm0_1)
23) AlarmMaskBit(Alarm0_1, Mask)
24) ResetAlarmIntFlagBit(Alarm0_1)
25) PowerHour12or24TimeFormatBit(SetHourType, PowerDownUp)
26) PowerAmPmBit(SetAmPm, PowerDownUp)
27) ResetPwFailBit()
28) WriteTimeKeeping(SetHourType)
29) ReadTimeKeeping()
30) WriteAlarmRegister(Alarm0_1, SetHourType)
31) ReadAlarmRegister(Alarm0_1)
32) WritePowerDownUpRegister(PowerDownUp, SetHourType)
33) ReadPowerDownUpRegister(PowerDownUp)
34) Set_EEPROM_WriteProtection(Section)
35) WriteProtected_EEPROM(RegAdd, RegData)
```

Arduino (**Fig. 1**). Si noti la differenza di impostazione del codice Python (usato per Raspberry Pi) rispetto al codice C usato per Arduino: per ogni registro del TimeKeeper sono state definite due strutture dati seguite da una "***union***" per consentire l'accesso al singolo registro sia a livello di byte che di singolo bit; vedete, a riguardo, la prima struttura dati evidenziata in giallo e riguardante l'impostazione dei secondi più il bit di attivazione dell'oscillatore; seguono le altre strutture dati evidenziate in grigio.
Di queste sezioni ce ne sono otto solo per la gestione dei registri del TimeKeeper.
Tutte e otto le sezioni vengono poi raggruppate assieme in un'ulteriore struttura dati in modo da avere un unico accesso strutturato ai registri (sezione evidenziata in verde). Con questo approccio si ha un unico "*contenitore*" dal quale è possibile modificare un singolo bit o un intero byte dei registri relativi al TimeKeeper.
Per chiarire meglio il concetto facciamo qualche esempio, supponiamo di voler settare il bit "***StartOsc***" del registro "***RTCSEC***":

```
MCP79410._TimeKeeper.Seconds.SecBit.StartOsc = 1
```

dove "***_TimeKeeper***" è la dichiarazione globale della struttura dati in esame visibile dallo sketch.

Se invece si volesse settare il bit "***VbatEn***", per l'abilitazione della gestione a batteria dell'integrato, si dovrà procedere in questo modo:

```
MCP79410._TimeKeeper.WeekDay.WkDayBit.Vbaten = 1
```

Una volta configurati i registri della struttura dati sopra descritta si può procedere alla programmazione effettiva di ogni singolo registro dell'integrato, sfruttando la funzione di libreria "***WriteSingleReg***", cui bisogna passare tre parametri (indirizzo hardware, indirizzo del registro e valore da scrivere nel registro), oppure alla programmazione completa di tutti i registri del TimeKeeper sfruttando la funzione di libreria "***WriteTimeKeeping***" alla quale dobbiamo passare un solo parametro ad indicare se il formato di gestione dell'ora debba essere nel formato 12h o 24h. Quindi la sintassi per programmare i registri in un colpo solo con formato dell'ora in 24h sarà:

```
MCP79410.WriteTimeKeeping(0)
```

Le funzioni messe a disposizione dalla libreria sono le medesime di quelle della precedente per Arduino. Quindi eviteremo di descrivere ogni singola funzione ma ci limiteremo a farne un elenco per concentrarci sul codice e le sue funzionalità. L'elenco completo è illustrato nel **Listato 1**.
Ricordiamo che la variabile "***ControlByte***" identifica l'indirizzo hardware della periferica I²C in questo caso abbiamo due indirizzi uno per il RTCC e l'altro per la EEPROM, la variabile "***RegAdd***" identifica l'indirizzo del registro che si vuole leggere o scrivere, la variabile "***Bit***" indica quale bit del registro desiderato si vuole modificare mentre "***RegData***" indica il valore a 8 bit che si vuole scrivere nel registro.
Le variabili "***SetReset***" e "***EnableDisable***" servono rispettivamente per settare o resettare una funzione dell'integrato oppure per abilitare o disabilitare una funzione.
"***SetHourType***" serve per impostare il formato dell'ora (12H oppure 24H), "***SetAmPm***" serve per indicare se l'ora indicata è prima di mezzogiorno oppure dopo, "***Alarm0_1***" indica su quale allarme si vuole lavorare mentre "***Mask***" è la maschera di confronto per la generazione di una condizione di allarme.
Infine "***PowerDownUp***" indica se si vuole leggere/scrivere i registri con i riferimenti di data e ora rilevate durante le fasi di PowerUp o PowerDown.

### LA LIBRERIA DIVENTA UN PACCHETTO DATI

Per agevolare l'utilizzo della libreria abbiamo creato un pacchetto di distribuzione detto "***package***". Con

questo accorgimento l'utente installa la libreria Python creata sul proprio Raspberry Pi, questa viene posizionata in un apposito percorso sul disco, per poi essere richiamata nei propri sketch con la funzione "***import***".
Per creare un pacchetto si deve organizzare il codice nel seguente modo:

1) creare una cartella base con un nome significativo. Nel nostro caso "***MCP79410***";
2) all'interno della cartella di cui sopra creiamone un'altra, sempre con nome "***MCP79410***" e all'interno di essa copiamo i nostri due file di libreria "***MCP79410.py***" e "***MCP79410_DefVar.py***".
3) creiamo un ulteriore file Python vuoto con il seguente nome "***__init__.py***"; in questo modo l'interprete Python vedrà questa cartella come un pacchetto di distribuzione (tecnicamente viene chiamato "***package***");
4) nella cartella base (vedere punto 1), creare il seguente file "***setup.py***", il quale deve contenere il seguente codice Python:

```
from distutils.core import setup

setup(name             = "MCP79410",
      version          = "1.0",
      description      = "MCP79410 Library",
      long_description = "This package is usefull
                          to manage the Real Time clock
                          Calendar MCP79410 developed
                          by Microchip",
      author           = "Matteo Destro",
      author_email     = "info@open-electronics.org",
      url              = "www.open-electronics.org",
      license          = "GPL",
      platform         = "RaspberryPi",
      packages         = ["MCP79410"])
```

Siccome quando si crea un file "***setup.py***" si deve sempre importare, dalla libreria "***distutils***", la voce "***setup***" e procedere successivamente alla compilazione di alcuni campi, nel nostro caso abbiamo dato un nome alla nostra distribuzione con la voce "***name***", ne abbiamo evidenziato la versione con la voce "***version***", ne abbiamo dato una breve descrizione con la voce "***description***" seguita da una descrizione più esaustiva con la voce "***long_description***". Segue la voce "***author***" e relativo indirizzo e-mail di supporto "***author_email***". Segue la voce "***url***" con il link dello sviluppatore e relativa licenza software "***license***". Infine la voce "***platform***" per cui è stato sviluppato il codice e la voce "***package***" ad identificare che quello in essere è un pacchetto di distribuzione. Per creare il pacchetto di distribuzione si deve eseguire il seguente comando:

```
sudo python setup.py sdist
```

che crea un file compresso contenente il pacchetto di distribuzione; il file creato è "***MCP79410-1,0.tar.gz***". Quindi l'utente che vorrà installare la libreria dovrà prima scompattare il file in un direttorio a piacere e poi eseguire il seguente comando per l'installazione della libreria:

```
sudo python setup.py install
```

Per importare e usare la libreria nei vostri sketch dovrete inserire in testa al file le seguenti righe di codice:

```
import sys
sys.path.insert(0, "/usr/local/lib/python2.7/dist-packages/MCP79410")
import MCP79410
```

## UN ESEMPIO DI CODICE

Il codice che abbiamo realizzato ha due scopi distinti: il primo è mostrare come utilizzare la libreria messa a disposizione, nonché le procedure per la configurazione dei registri di interesse; il secondo è spiegarvi come interagire con l'hardware della nostra scheda demo, in quanto l'obiettivo è l'accensione e lo spegnimento automatizzato del Raspberry Pi in determinate condizioni dettate da come si sono configurati i registri degli allarmi.
Iniziamo la trattazione dal file principale ovvero quello che contiene la routine di "***main()***".
Facciamo notare che in testa al file ci sono le famose righe di codice con le istruzioni di "***import***" per inglobare la libreria appena trattata più altre librerie di sistema necessarie alla realizzazione dello sketch.
Subito dopo seguono le dichiarazioni delle variabili globali, la dichiarazione delle costanti che identificano i pin GPIO utilizzati e la loro configurazione come ingressi o uscite. Come ingressi abbiamo i soli pulsanti P1, P2 e P3 a cui viene imposto un pull-up interno più un evento sul fronte di discesa, con tempo di debouncing di 100 millisecondi, per intercettare la pressione del pulsante.
Per andare a leggere con cadenza regolare i registri di interesse del nostro MCP79410 abbiamo predisposto due timer, uno con periodo di 5 secondi (`def ReadsAndPrintsRegisters():`) e l'altro con periodo pari a 1 secondo (`def CheckAlarmsFlag():`).
Con cadenza pari a 5 secondi si vanno a leggere i registri del TimeKeeper, allo scopo di mostrare a video la data e l'ora correnti, i registri di allarme per visualizzarne la configurazione e, se necessario, i registri di TimeStamp. Questi ultimi vengono letti una sola volta all'accensione della nostra scheda Raspberry Pi.

# Configurare **Raspberry Pi**

Se è la prima volta che utilizzate Raspberry Pi o non avete mai provato a scrivere del codice Python conviene eseguire una serie di passi per configurare Raspberry Pi in modo da utilizzare le librerie di gestione dei GPIO e del bus I²C. Vediamo passo-passo come procedere:

1) Per prima cosa installare la libreria Python per la gestione degli I/O di Raspberry Pi e se già installata provvedere ad aggiornarla all'ultima release disponibile che attualmente è la 0.5.11.
   - Per verificare l'attuale versione della libreria GPIO eseguire il seguente comando:

```
find /usr | grep -i gpio
```

   La **Fig. 2** evidenzia il feedback del comando inviato, in questo caso la libreria risulta aggiornata all'ultima release. Se la release della libreria dovesse essere inferiore alla 0.5.11 provvedere ad un suo aggiornamento usando i seguenti comandi:

```
sudo apt-get update
sudo apt-get upgrade
```

   oppure impartendo i seguenti:

```
sudo apt-get install python-rpi.gpio
sudo apt-get install python3-rpi.gpio
```

2) Passiamo ora alla configurazione della periferica I²C messa a disposizione da Raspberry Pi:
   - procediamo con l'installazione della libreria Python "***smbus***" per la gestione del bus I²C:

```
sudo apt-get install python-smbus
```

   - seguita da quella dei tools I²C:

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

3) Dobbiamo poi abilitare il supporto del bus I²C da parte del kernel; per fare ciò è necessario richiamare la finestra di configurazione digitando il comando:

```
sudo raspi-config
```

   - come in **Fig. 3**, selezioniamo la voce "***Advanced Options***";
   - nella nuova schermata selezioniamo la voce ***"A7 I2C Enable/Disable automatic loading"***, come in **Fig. 4**;
   - nelle seguenti schermate rispondere in sequenza ***"Yes"***, ***"Ok"***, ***"Yes"*** e infine ancora ***"Ok"***;
   - usciiamo quindi dalla finestra di configurazione selezionando la voce ***"Finish"***;

4) Procediamo con un reboot del sistema eseguendo il comando:

```
sudo reboot
```

   - dopo avere riavviato Raspberry Pi aprire con un editor di testo generico il file ***"modules"***;

```
sudo nano /etc/modules
```

   - Aggiungere, se mancanti, le seguenti due righe come evidenziato nella **Fig. 5.**

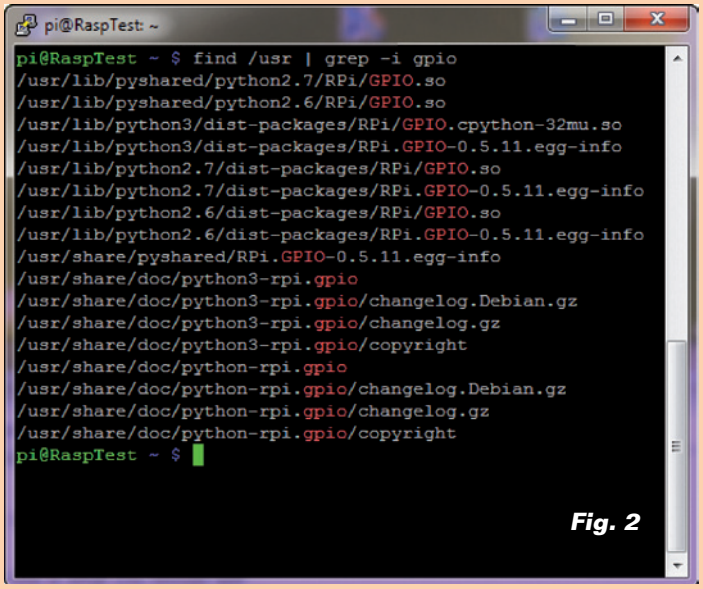
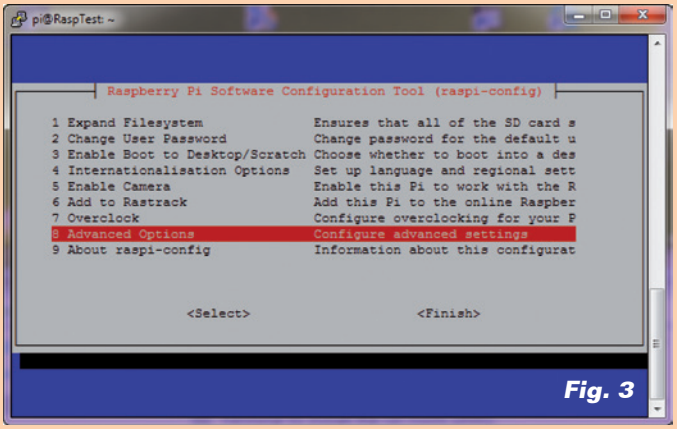
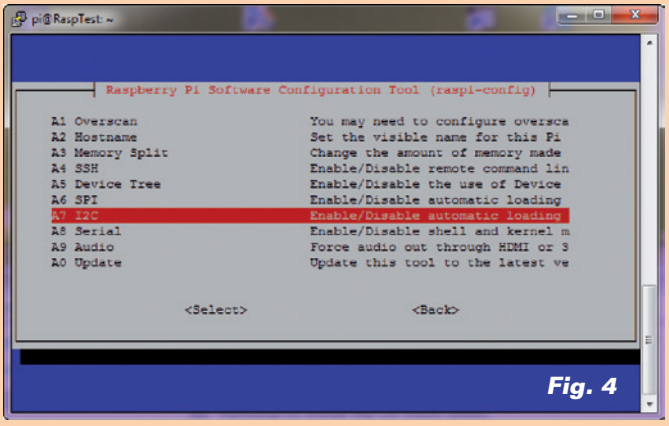


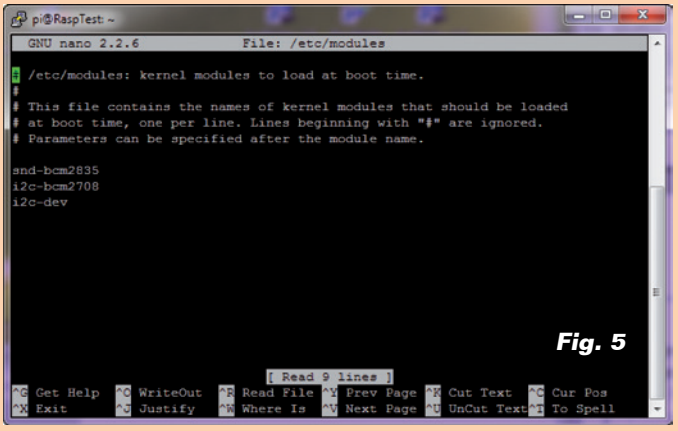

***Fig. 2***

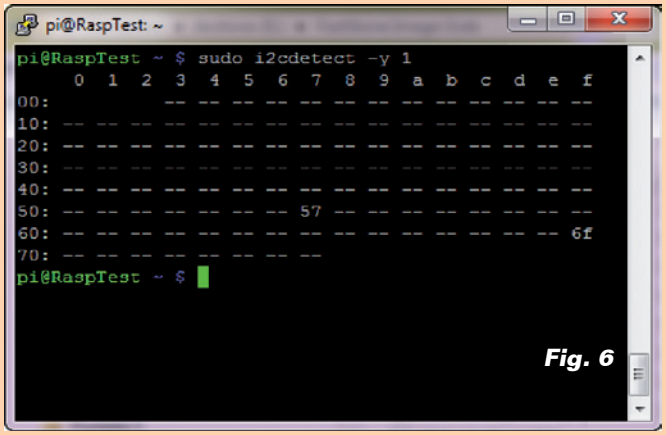
Il secondo timer, impostato con cadenza pari a 1 secondo, serve per monitorare gli allarmi; nel caso in cui uno dei due allarmi sia attivo, si provvederà ad eseguire una delle due azioni di seguito elencate:

1) **Azione accensione programmata scheda**. L'allarme 0 è impostato per accendere a un dato orario l'elettronica della nostra scheda e di conseguenza anche la Raspberry Pi. Quindi in questo caso lo sketch rileva che il flag dell'allarme 0 è a 1, forza l'alimentazione ad ON tramite la linea "*FORCE_ON*" e resetta il flag di allarme. Prima di concludere il processo, disabilita l'allarme 0 e attiva l'allarme 1.
2) **Azione spegnimento programmato scheda**. L'allarme 1 è stato impostato per forzare lo spegnimento della scheda Raspberry Pi e quindi anche della nostra elettronica, a un determinato orario. In questo caso lo sketch rileva che il flag dell'allarme 1 è a 1 logico e di conseguenza abilita l'allarme 0 e disabilita l'allarme 1, resetta il flag pendente dell'allarme 1 e invia il comando di shutdown alla Raspberry Pi il quale inizierà la propria procedura di spegnimento. Concluso




```
i2c-bcm2708
i2c-dev
```

5) Per ultimo dovete verificare se nella vostra distribuzione è presente il file:

```
/etc/modprobe.d/raspi-blacklist.conf
```

- Se dovesse essere presente apritelo con un editor di testo:

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

- Se presenti commentate, utilizzando il carattere **"#"**, le seguenti righe:

```
blacklist spi-bcm2708
blacklist i2c-bcm2708
```

6) A questo punto non rimane che riavviare nuovamente la Raspberry Pi e verificare che la comunicazione I²C sia attiva. Per fare ciò eseguite il comando:

```
sudo i2cdetect -y 1
```

Se tutto è andato a buon fine, il comando inviato ritorna una tabella come quella mostrata in **Fig. 6**. In pratica il comando analizza il bus I²C in cerca di periferiche e nel nostro caso ha individuato l'integrato MCP79410 con i suoi due indirizzi 0x57 e 0x6F.

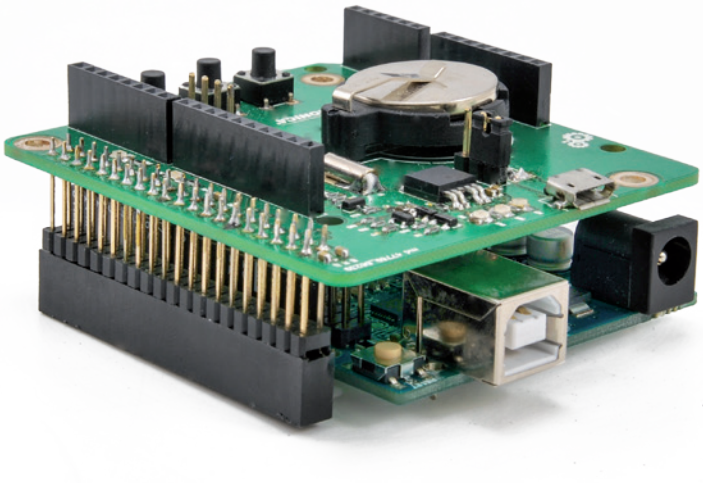
lo shutdown della Raspberry Pi, la linea "***FORCE_ON***" verrà rilasciata, con conseguente spegnimento della nostra elettronica.

I due time esaminati sono dei "***thread***" e per avviarli si deve usare la seguente linea di codice:

```
threading.Timer(1.0, CheckAlarmsFlag).start()
```

dove il primo parametro indica la cadenza, in questo caso 1 secondo, mentre il secondo parametro indica a quale funzione è associato. Il primo avvio avviene semplicemente richiamando la funzione "`CheckAlarmsFlag()`" dal main prima di entrare nel loop infinito.
Osserviamo ora, più nel dettaglio, il codice presente nei due timer: ad esempio quello in "`CheckAlarmsFlag()`". Si noteranno più richiami alla nostra libreria MCP79410, sia a livello di lettura della sola struttura dati sia a livello di chiamata di funzione. Ad esempio, si testano i flag dei due allarmi sfruttando la struttura dati:

```
if (MCP79410._Alarm.Alm[0].WeekDay.WkDayBit.AlarmIF == 1):
```



Questa struttura dati è un array di strutture perché gli allarmi sono due. Nel codice esposto, si punta all'allarme 0, in particolare al registro WeekDay (***ALM0WKDAY***) e al relativo flag di allarme (***ALM0IF***). Per i dettagli sui registri consultate il data-sheet. Oltre al test sopra esposto ci sono delle chiamate a funzione, come ad esempio "`MCP79410.Alarm0Bit(0)`" per disabilitare l'allarme 0 oppure "`MCP79410.ResetAlarmIntFlagBit(0)`" per resettare il flag dell'allarme 0.
Analizziamo ora la routine "***main()***": in testa ad essa si esegue una lettura della EEPROM interna all'MCP79410, esclusivamente per scopo didattico, e per fare ciò si sfrutta la funzione di libreria per leggere un array di byte a partire da un dato indirizzo. La funzione prevede tre parametri:

```
MCP79410.ReadArray(MCP79410.EEPROM_HW_ADD, 0, 13)
```

Il primo è l'indirizzo hardware che come già ampiamente detto è diverso da quello del RTCC, segue l'indirizzo di partenza da cui iniziare la lettura dei dati e infine il numero di byte che si vuole leggere. Il risultato della lettura viene messo in un array a cui si punta con la seguente:

```
MCP79410.DataArray[i]
```

dove "***i***" è l'indirizzo dell'array. La dimensione massima è impostata a 16 byte. Il contenuto viene poi stampato a video facendo comparire la scritta "***ElettronicaIn***". La scrittura in EEPROM è stata fatta con l'omonima funzione:

```
MCP79410.WriteArray(MCP79410.EEPROM_HW_ADD, n, m)
```

alla quale si passano gli stessi parametri della precedente. Segue l'attivazione dei due Timer, di cui abbiamo parlato prima, e infine il ciclo "***while***" infinito.
All'interno del ciclo "***while***" trovano posto la gestione dei tre pulsanti P1, P2 e P3 ai quali sono state assegnate tre funzioni differenti.
Il pulsante P1 serve per programmare la data e l'ora di MCP79410 e per fare ciò richiama una funzione apposita che legge data e ora di sistema e la trasferisce all'integrato. La funzione che si occupa di programmare data e ora di MCP79410 si chiama "`SetTimeKeeperByLocalDateTime("24H")`" a cui va passato un solo parametro, ovvero se il formato dell'ora debba essere 12h o 24h. Il parametro è di tipo stringa (la funzione è memorizzata nel file "***MCP79410_SetRegister.py***"). Se invece si vuole impostare la data e l'ora manualmente, va richiamata la seguente funzione:

```
ManualSetTimeKeeper(Hours, Minutes, Seconds,
Set_12H_24H, Set_AM_PM, Date, Month, Year, WkDay)
```

la quale richiede una serie di parametri. Un esempio di utilizzo potrebbe essere il seguente:

```
ManualSetTimeKeeper(15, 30, 00, "24H", "AM", 25, 12, 15, 5)
```

Così si imposta l'ora nel formato "***24H***" alle 15:30:00, la data è Venerdì 25/12/2015 e la dicitura "***AM***" rispetto a "***PM***" perde di significato in quanto il formato dell'ora è "***24H***".
Se fosse stata in "***12H***" avremmo dovuto passare i parametri nel seguente modo:

```
ManualSetTimeKeeper(03, 30, 00, "12H", "PM", 25, 12, 15, 5)
```

In entrambi i casi le funzioni riempono la struttura dati inerente i registri del TimeKeeper per poi richiamare la funzione di libreria che esegue la scrittura fisica sul dispositivo.
Il pulsante P2 serve per programmare gli allarmi 0 e 1; allo scopo richiama due funzioni distinte, una per l'allarme 0 e l'altra per l'allarme 1.
La configurazione è solo di tipo manuale e viene eseguita solo da codice; ciò significa che non abbiamo predisposto un sistema di inserimento dati da tastiera. La funzione da chiamare per impostare gli allarmi è la seguente:

```
ManualSetAlarm0(Index, Hours, Minutes, Seconds,
Set_12H_24H, Set_AM_PM, Date, Month, WkDay, AlarmMask, AlarmPol)
```

Come si può notare, ci sono parecchi parametri da

passarle; un esempio potrebbe essere:

```
ManualSetAlarm0(0, 12, 40, 00, "24H", "AM", 25, 12, 1, 1, "LHL")
```

Così facendo stiamo dicendo alla funzione di lavorare sull'allarme 0, l'ora è nel formato "***24H***" impostata alle 12:40:00. La data è impostata a 25/12/****. L'anno non è specificato perché privo di significato per gli allarmi. Il giorno della settimana è lunedì, la maschera per gli allarmi è fissata sui minuti e la polarità è "***Logic High Level***".
Anche in questo caso la funzione riempe una struttura dati per poi richiamare una funzione di libreria che esegue la scrittura fisica sul dispositivo. Suggeriamo di consultare il data-sheet per i dettagli sui registri.
Il pulsante P3 viene usato per disabilitare e resettare gli allarmi 0 e 1. Dopo il reset degli allarmi e la loro disabilitazione lo sketch si limita a mostrare la data e l'ora letta dal RTCC con cadenza di 5 secondi.
Per la stampa a video di tali informazioni si richiama una funzione di stampa contenuta nel file "***MCP79410_PrintFunc.py***" alla quale si deve passare una serie di parametri per decidere cosa stampare e cosa no.
La funzione in esame si chiama:

```
PrintDataMCP79410(PrintTimeKeeper, PrintAlarm0,
PrintAlarm1, PrintPowerDown, PrintPowerUp)
```

I parametri da passargli non sono altro che valori booleani -vero o falso- e dicono alla funzione cosa si vuole stampare a video.
Come suggeriscono i nomi, in sequenza abbiamo:

- stampa a video di data e ora;
- stampa a video impostazioni Allarme 0;
- stampa a video impostazioni Allarme 1;
- stampa a video valori di TimeStamp PowerDown;
- stampa a video valori di timeStamp PowerUp.

Se tutti i valori booleani sono posti a "***False***" niente verrà stampato a video.

## AVVIO AUTOMATICO DELLO SKETCH DOPO IL BOOT

Il passo successivo consiste nel rendere auto-avviante il nostro codice dopo il boot del sistema operativo. Allo scopo, prima di tutto si deve decidere se si intende mantenere il login attivo oppure no; nel caso in cui si decida di saltare la fase di login si deve procedere così:

1) utilizzando il terminale, eseguire il seguente comando:

```
sudo nano /etc/inittab
```

2) cercare nel file la seguente riga:

```
1:2345:respawn:/sbin/getty -noclear 38400 tty1
```

3) commentarla utilizzando il carattere "**#**" in testa alla riga; sotto ad essa aggiungere la seguente ("***pi***" è lo user-name):

```
1:2345:respawn:/bin/login -f pi tty1</dev/tty1>/
dev/tty1 2>&1
```

4) uscire e salvare il file così modificato premendo "***ctrl+x***" e successivamente "***y***";
5) salvato il file, eseguire il seguente comando:

```
sudo nano /etc/profile
```

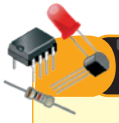
6) andare in fondo al file e aggiungere la seguente:

```
sudo python /home/pi/PythonProject/MCP79410_
Lib/WK/MCP79410_SetAndReadRegisters.py
```

Ovviamente come percorso dovete mettere quello del vostro sistema; nel nostro caso il path in cui si trovano i tre file dello sketch di esempio è "***/home/pi/PythonProject/MCP79410_Lib/WK/***". Alla fine riavviate il sistema.

## CONCLUSIONI

Con questo abbiamo concluso la trattazione della gestione della nostra scheda con RTCC tramite Raspberry Pi. Vi abbiamo dato diversi spunti di riflessione per realizzare le vostre più disparate applicazioni sfruttando la nostra libreria Python appositamente scritta per l'occasione.
Non ci rimane che augurarvi buon lavoro!



## per il MATERIALE

Lo shield RTC Raspberry-Arduino (cod. FT1254M) viene venduto montato e collaudato ed è disponibile presso Futura Elettronica al prezzo di Euro 20,00 IVA compresa.

Il materiale va richiesto a:
Futura Elettronica, Via Adige 11, 21013 Gallarate (VA)
Tel: 0331-799775 · Fax: 0331-792287 - www.futurashop.it