Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Triennale in Ingegneria dell'Informazione

TESI DI LAUREA TRIENNALE

PROGETTAZIONE E IMPLEMENTAZIONE SU MODULO ARDUINO DI UN DATALOGGER PER IL MONITORAGGIO DI PARAMETRI AMBIENTALI DI UNA CELLA A COMBUSTIBILE

Laureando:
Stefano Cillo
Matricola INF-594610

Relatore:
Prof. Paolo Tenti

Correlatore:
Ing. Marco Stellini

Anno Accademico 2012-2013

Alla mia famiglia e ai miei amici.

A tutte le persone che mi sono state vicine nel sentiero percorso, e a quelle che condivideranno con me questo difficile quanto meraviglioso viaggio chiamato vita.

Sommario

Lo scopo di questo lavoro di tesi è quello di illustrare il modulo progettato per monitorare i parametri ambientali di una cella a combustibile: i valori di umidità e temperatura sono campionati ad intervalli regolari e vengono salvati in una memoria dedicata. Questi campioni potranno poi essere trasferiti su un PC tramite l'apposita interfaccia grafica, che permette di visualizzare gli andamenti dei parametri tramite dei comuni programmi spreadsheet come Microsoft® Excel o OpenOffice Calc.

Il progetto è stato sviluppato su piattaforma Arduino©nel linguaggio di programmazione C, utilizzando la toolchain AVR-GCC. La suite scelta permette di mantenere una programmazione meno astratta e più vicina al livello hardware, evitando così di allontanarsi troppo dalla realtà fisica gestita dal software.

Indice

Introduzione

La realizzazione di questo progetto nasce dalla necessità di garantire un continuo e rigoroso monitoraggio di alcuni dei parametri ambientali di una cella a combustibile. Infatti perché questa possa funzionare correttamente ed evitare indesiderati danni (della cella stessa e dello spazio circostante, data l'alta probabilità di esplosioni nel caso di rottura delle membrane della pila a combustibile) è necessario mantenere i valori di temperatura e umidità all'interno di determinati range.

Da qui ha origine l'idea dell'implementazione di un modulo capace di effettuare la trasduzione di temperatura e umidità in grandezze elettriche ed effettuarne la conversione analogico/digitale, e che sia in grado di impostare delle soglie, superate le quali venga attivato, a seconda della soglia, o un allarme o un dispositivo che deumidifichi l'aria circostante. Il tutto deve poter essere pilotato *in loco* da un utente, e quindi deve essere presente un'interfaccia grafica tramite la quale sia possibile interagire con il software di controllo. Oltre a queste specifiche si aggiunge il fatto che il modulo deve essere in grado di salvare un numero elevato di campioni, e che questi possano essere facilmente usufruibili ai tecnici manutentori della cella.

Il dispositivo hardware che meglio risponde alle esigenze richieste è la piattaforma open-source Arduino, sulla quale è montato un microcontrollore Atmel® ATmega328P: esso fornisce convertitori analogico/digitali integrati a 10 bit di risoluzione, numerose periferiche programmabili e porte di I/O tramite le quali è possibile dialogare con periferiche esterne, come una memoria EEPROM dedicata al salvataggio dei dati e un display LCD su cui visualizzare l'interfaccia utente. Arduino integra inoltre la possibilità di collegarsi ad un PC tramite porta USB, e questo permette di rendere facilmente disponibili i dati campionati dal microcontrollore.

Descrizione dei capitoli

Capitolo 1 - Le celle a combustibile

In questo primo capitolo viene richiamata la teoria di funzionamento di una cella galvanica, in particolare della pila Daniell. Il richiamo è d'obbligo per il fatto che le celle a combustibile derivano direttamente dalle celle galvaniche. Oltre alle celle galvaniche verranno trattate brevemente le celle a combustibile, e dopo alcuni cenni storici si discuterà delle celle a membrana polimerica, tecnologia della cella a combustibile Serenus 166 AirC, modello attualmente presente in dipartimento.

Capitolo 2 - Descrizione dell'hardware utilizzato

Dopo una panoramica sull'hardware generico necessario per il progetto, verranno esaminati in dettaglio i componenti utilizzati nella realizzazione di questo lavoro di tesi: dalla piattaforma Arduino al microcontrollore ATmega328P, dai sensori di trasduzione al chip per la memorizzazione dei dati campionati.

Capitolo 3 - Sviluppo del software di controllo

Il capitolo contiene la descrizione della politica di gestione del microcontrollore utilizzata per il progetto. Viene richiamato il concetto di *multitasking*, *con* o *senza prelazione*; verrà spiegato il ruolo delle principali routine di interrupt, la struttura utilizzata per il mantenimento dei dati in memoria EEPROM e verranno elencate le principali librerie software utilizzate.

Capitolo 4 - Risultati e conclusioni

Nell'ultimo capitolo verrà discusso dei principali problemi riscontrati nello sviluppo del progetto, in particolare del software; verrà poi trattata brevemente la realizzazione sperimentale del progetto stesso, e verranno elencate alcune possibilità di sviluppo per il modulo implementato.

Capitolo 1

Le celle a combustibile

Le *celle a combustibile* (o *pile* a combustibile) sono dispositivi il cui principio di funzionamento si basa prevalentemente su quello delle *celle galvaniche*: una reazione di ossidoriduzione spontanea avviene tra due semielementi (o elettrodi) separati da una membrana che consente il solo passaggio degli ioni[1], mentre gli elettroni liberati nel processo chiudono il circuito elettrico attraverso un conduttore esterno al quale è collegato un carico, fornendo lavoro utile a quest'ultimo. Questo tipo di reazioni fanno parte della branchia della chimica conosciuta come *elettrochimica*, disciplina che si occupa essenzialmente della trasformazione di energia chimica in energia elettrica, o, più correttamente, della variazione di energia libera in forza elettromotrice; alla base di queste trasformazioni ci sono appunto i processi ossidoriduttivi.

1.1 Le reazioni di ossidoriduzione

Le reazioni che avvengono con trasferimento di carica (elettroni) da una specie[2] all'altra sono dette reazioni di ossidoriduzione o processi redox, e sono caratterizzate dalla variazione dei numeri di ossidazione[3] di alcuni elementi presenti nei composti che reagiscono [1].

Ad esempio, reazioni di ossidoriduzione sono le seguenti:

$$H_2 + \frac{1}{2}O_2 \rightarrow H_2O$$

$$H_2 + Cl_2 \rightarrow 2HCl$$

[1] Ione: entità molecolare elettricamente carica; quando un atomo o una molecola cede o acquista uno o più elettroni si trasforma in uno ione.

[2] Specie chimica: insieme omogeneo di entità molecolari, ovvero formato dallo stesso tipo di elementi, siano essi atomi o molecole.

[3] Il numero di ossidazione è un parametro utile a visualizzare la ripartizione degli elettroni di legame tra i vari atomi dell'aggregato: è pari a 0 per un composto elementare, alla carica elettrica per i composti ionici, mentre per un composto covalente rappresenta la carica che l'atomo assumerebbe se il legame fosse completamente ionico.

dove la prima reazione avviene con un trasferimento parziale di carica dall'idrogeno all'ossigeno, mentre nella seconda il trasferimento avviene dagli atomi di idrogeno a quelli di cloro.

Viene chiamata ***reazione di ossidazione*** una reazione che porta alla diminuzione del numero di elettroni presenti in una specie[4]; la specie che si ossida è chiamata *agente riducente* ed è da notare che l'ossidazione di una specie porta all'aumento del suo numero di ossidazione.
Ad esempio:

$$Na \rightarrow Na^{+} + e^{-}$$

La reazione inversa a quella di ossidazione è detta ***reazione di riduzione*** e porta all'aumento (anche parziale) della carica elettronica di una specie; la specie che si riduce è detta *agente ossidante* e al contrario dell'ossidazione quando una specie si riduce diminuisce il suo numero di ossidazione.
Un esempio di reazione di riduzione è:

$$Cl_2 + 2e^{-} \rightarrow 2Cl^{-}$$

1.2 Celle galvaniche

1.2.1 La Pila Daniell

Le ***pile*** o ***celle galvaniche*** sono dispositivi che trasformano in energia elettrica la variazione di energia libera che accompagna una reazione *spontanea* di ossidoriduzione; si parla in questo caso di *pile chimiche*[5]. Affinché la trasformazione possa aver luogo è necessario che gli elettroni ceduti dalla specie che si ossida vengano trasferiti a quella che si riduce non in modo diretto, ma *attraverso un conduttore metallico* che li trasporta da una specie all'altra, tenute opportunamente separate[1].

Si consideri ad esempio la Figura 1.1, rappresentazione di una pila Daniell[6], dove avviene la seguente reazione di ossidoriduzione tra zinco metallico e ioni rame (II) in soluzione acquosa:

$$Zn_{(s)} + Cu^{++}_{(aq)} \rightarrow Zn^{++}_{(aq)} + Cu_{(s)} \tag{1.1}$$

Tale reazione può essere considerata come il risultato di una ossidazione e di una riduzione rappresentate dalle seguenti semireazioni:

$$\textit{Ossidazione: } Zn_{(s)} \rightarrow Zn^{++}_{(aq)} + 2e^{-}$$
$$\textit{Riduzione: } Cu^{++}_{(aq)} + 2e^{-} \rightarrow Cu_{(s)}$$

[4]Se una specie si impoverisce di elettroni contemporaneamente una specie se ne arricchisce.

[5]Pile dove il processo sfruttato è di diluizione di una soluzione o di espansione di un gas prendono il nome di *pile di concentrazione*.

[6]Messa a punto nel 1836 da John Frederic Daniell sfruttando i primi studi effettuati da Alessandro Volta e Luigi Galvani.

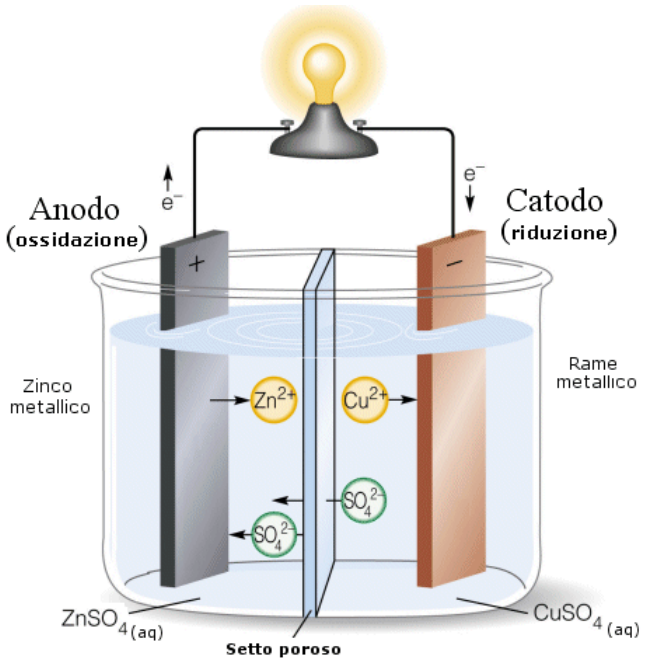


Figura 1.1: Schema di una pila Daniell

Se la reazione (1.1) viene realizzata in modo che le due semireazioni avvengano in compartimenti opportunamente separati il risultato è che gli elettroni migrano in modo ordinato dal luogo dove ha sede l'ossidazione a quello dove si ha la riduzione attraverso il conduttore metallico esterno, *generando così una corrente elettrica*. Ognuno di questi compartimenti prende il nome di **semielemento**.

Una cella galvanica è costituita quindi da due *semielementi* (spesso indicati, anche se impropriamente, con il termine ***elettrodi***) e, quando la pila è in funzione, in uno di essi si realizza un processo di ossidazione che libera elettroni e nell'altro quello di riduzione che li utilizza. I due semielementi sono separati da un ***setto poroso***: la funzione di questo dispositivo è quello di impedire il mescolamento delle due soluzioni, pur consentendo il passaggio di ioni e quindi assicurando la continuità del circuito elettrico, mantenendo quindi l'*elettroneutralità* delle soluzioni dei due semielementi.

Ogni semielemento è formato da un *elettrodo*, cioè un conduttore di prima specie, a contatto con una soluzione elettrolitica[7]: nel caso specifico della pila Daniell un semielemento è costituito da un elettrodo di Zn immerso in una soluzione di solfato di zinco, $ZnSO_4$, contenente quindi ioni Zn^{2+} e SO_4^{2-}, mentre l'altro semielemento da un elettrodo di rame Cu immerso in una soluzione di solfato di

[7]Gli *elettroliti* sono sostanze a struttura ionica (come i sali) o a struttura covalente, che in soluzione di opportuni solventi polari o in fase liquida allo stato puro sono totalmente o parzialmente scissi in ioni positivi(*cationi)* e negativi (*anioni*); gli ioni possono muoversi liberamente in soluzione, ma interagiscono per via elettrostatica con le molecole polari del solvente. Gli elettroliti si dicono *forti* se in soluzione sono completamente dissociati, si dicono *deboli* altrimenti. Un elettrolita quindi ha la capacità di condurre corrente elettrica grazie all'intervento di ioni: gli elettroliti costituiscono quindi dei *conduttori ionici* o *di seconda specie*.

rame, $CuSO_4$, contenente quindi ioni Cu^{2+} e SO_4^{2-}. La pila è completata poi da un circuito elettrico esterno costituito da un conduttore di prima specie che assicura il collegamento dei due elettrodi.

A circuito aperto (ovvero quando non è collegato un carico) la pila non è in funzione e non si ha circolazione di corrente. Si stabiliscono quindi i seguenti equilibri tra gli atomi metallici ed i corrispondenti ioni in soluzione:

$$Zn_{(s)} \rightleftharpoons Zn^{++}_{(aq)} + 2e^- \tag{1.2}$$

$$Cu_{(s)} \rightleftharpoons Cu^{++}_{(aq)} + 2e^- \tag{1.3}$$

All'equilibrio viene a crearsi un eccesso di elettroni sui due elettrodi metallici e di ioni positivi nelle due soluzioni: si genera così in ciascun semielemento una differenza di potenziale tra la soluzione (positiva) e l'elettrodo (negativo), chiamata *potenziale assoluto del semielemento*.
Per sua natura lo zinco ha una tendenza maggiore del rame a passare in soluzione (e quindi ad ossidarsi), perciò l'equilibrio di (1.2) risulta più spostato verso destra dell'equilibrio (1.3). Ciò significa che a circuito aperto l'elettrodo Zn risulta carico più negativamente di quello Cu: per questo l'elettrodo Zn viene convenzionalmente considerato negativo.

Se si chiude il circuito l'eccesso di elettroni presente sull'elettrodo di Zn passa attraverso il circuito esterno all'elettrodo di Cu, provocando uno spostamento continuo dell'equilibrio (1.2) verso destra (viene allontanato continuamente un prodotto della reazione, utilizzando un reagente: se si guarda la pila mentre questa sta erogando energia si nota che la lamina di zinco si assottiglia, perché viene utilizzato dello zinco metallico) ed un continuo spostamento dell'equilibrio (1.3) verso sinistra (viene infatti continuamente aggiunto un reagente della reazione, gli elettroni: durante il funzionamento questo elettrodo invece aumenta di dimensione, perché ioni Cu^{++} si legano a due elettroni per formare un atomo di Cu ed attaccarsi all'elettrodo). Così, fintantoché la pila non è scarica la semireazione (1.2) procede come ossidazione e la (1.3) come riduzione:

$$Zn_{(s)} \rightarrow Zn^{++}_{(aq)} + 2e^- \tag{1.4}$$

$$Cu_{(s)} \rightarrow Cu^{++}_{(aq)} + 2e^- \tag{1.5}$$

Per cui durante il funzionamento della pila avviene complessivamente la reazione di ossidoriduzione (1.1):

$$Zn_{(s)} + Cu^{++}_{(aq)} \rightarrow Zn^{++}_{(aq)} + Cu_{(s)}$$

Il semielemento di Zn, sede della semireazione di ossidazione, viene chiamato **anodo**, mentre il semielemento di Cu, sede della semireazione di riduzione, viene chiamato **catodo**.

La somma delle semireazioni (1.4) e (1.5) dà la reazione totale di ossidoriduzione (1.1) che avviene nella pila: è l'energia chimica prodotta in questa reazione ad essere trasformata in energia elettrica.

Ciò avviene finché la pila non è *scarica*, ovvero quando la reazione (1.1) ha raggiunto l'equilibrio:

$$Zn_{(s)} + Cu^{++}_{(aq)} \rightleftharpoons Zn^{++}_{(aq)} + Cu_{(s)}$$

La condizione di equilibrio è raggiunta per l'impossibilità di Zn di scindersi in ioni Zn^{++} e quindi liberare elettroni (tutto lo zinco è stato consumato), o perché l'elettrodo di Cu non ha più atomi da mettere a disposizione per la riduzione (perché la concentrazione di ioni Cu^{++} all'interno dell'elettrolita è diventata praticamente nulla), oppure per entrambe le situazioni in contemporanea.

Durante il funzionamento della pila, in vicinanza dell'elettrodo di Zn la soluzione anodica (elettrolita contenente Zn^{++} e SO_4^{--}) si carica positivamente per la formazione di ioni $Zn^{++}_{(aq)}$ dalla reazione di ossidazione, mentre la soluzione catodica (elettrolita contenente Cu^{++} e SO_4^{--}) si carica negativamente per la presenza di un eccesso di ioni negativi SO_4^{--} formatosi per la reazione di riduzione. Questo processo porterebbe all'arresto del funzionamento della pila, dal momento che sarebbero impedite, per azione elettrostatica, l'ulteriore formazione di ioni Zn^{++} e l'ulteriore scarica di ioni Cu^{++}. A riequilibrare questo sbilanciamento di cariche ioniche e a ristabilire quindi l'elettroneutralità del sistema interviene la diffusione degli ioni delle soluzioni elettrolitiche dei due semielementi attraverso il *setto poroso*: si ha infatti una migrazione di ioni SO_4^{--} dalla zona catodica dove sono in eccesso, attraverso il setto poroso, fino alla zona anodica, e di ioni Zn^{++} dalla zona anodica a quella catodica.

In definitiva, durante il funzionamento di una pila, si ha un movimento di elettroni (condizione elettronica *senza trasporto di materia*) nel circuito esterno dall'anodo (negativo) al catodo (positivo) e una contemporanea migrazione di ioni (conduzione elettrolitica *con trasporto di materia*) all'interno della pila, attraverso il setto poroso[1].

1.2.2 Tipi di semielementi

I ***semielementi*** (od ***elettrodi***, come sono spesso chiamati) che costituiscono una cella galvanica possono essere di diversi tipi. I più comuni sono:

Semielementi di prima specie. Essi sono costituiti da un *elettrodo metallico* immerso in una soluzione elettrolitica contenente i suoi ioni: ne sono un esempio quelli che costituiscono la pila Daniell appena descritta. In tali semielementi l'elettrodo partecipa attivamente al funzionamento della cella, venendo consumato o incrementato mano a mano che la reazione procede.

Semielementi a gas. Essi sono costituiti da un *elettrodo inerte* (ad esempio Pt, Au), cioè costituito da un metallo che rimane chimicamente inalterato nel corso della reazione elettrodica e serve solo come conduttore elettronico, e da un *gas* a contatto con l'elettrodo, con l'elettrolita contenente contenente

l'anione o il catione formato dal gas.
Un esempio tipico è costituito dall'*elettrodo a idrogeno*, in cui è presente l'equilibrio

$$2H_3O^+_{(aq)} + 2e^- \rightleftharpoons H_{2(g)} + 2H_2O$$

1.3 Celle a combustibile

1.3.1 Generalità

Una ***cella a combustibile*** (*fuel cell, FC*) è un generatore elettrochimico (cioè una *cella galvanica*) che, per produrre forza elettromotrice sfrutta, come reazione globale di ossidoriduzione, una *reazione di combustione*. Sarà presente quindi un combustibile (in genere idrogeno H_2, ma anche metanolo CH_3OH, metano CH_4, ecc.) ed un ossidante (O_2 puro oppure aria)[1].

La reazione su cui si basano le celle a combustibile è la seguente:

$$2H_2 + O_2 \rightleftharpoons 2H_2O \tag{1.6}$$

Nella cella a combustibile sono presenti due semielementi a gas, solitamente formati da elettrodi inerti in platino separati da una membrana di vario tipo; la reazione avviene tra i due gas a contatto con i rispettivi elettrodi: idrogeno al catodo e ossigeno all'anodo, così da generare la reazione di ossidoriduzione (1.6). I gas sono mantenuti in serbatoi esterni: uno dei problemi derivanti dall'utilizzo dell'idrogeno è la sua scarsa densità energetica su base volumetrica, che richiede per il suo stoccaggio il mantenimento del combustibile a pressioni molto elevate o a temperature molto basse. In questa direzione si muovono ricerche per l'utilizzo agevole di combustibili alternativi all'H_2.
A differenza delle celle galvaniche (le quali presentano un semielemento di prima specie), dove la saturazione ionica dell'elettrolita avviene naturalmente per la spontaneità della reazione (ioni Zn^{++} da una parte e SO_4^{--} dall'altra), nelle pile a combustibile è necessaria la presenza di un catalizzatore che aiuti il processo di ionizzazione: il tipo e la modalità con cui avviene questo processo dipende dalla tecnologia della cella. Ad esempio, nelle PEMFC[8] il catalizzatore per il combustibile e il comburente è il platino, ma data la poca convenienza del platino si stanno cercando materiali alternativi ugualmente performanti.

La prima cella a combustibile fu ideata da William Grove nel 1839, fisico dilettante, che mise a punto una cella del tipo H_2/O_2: era composta da elettrodi porosi di platino ed acido solforico come bagno elettrolita (rappresentata in Figura 1.2). La miscela di idrogeno e ossigeno produceva elettricità, e come unico scarto di emissione, acqua.

[8] Vedi sezione 1.3.2.

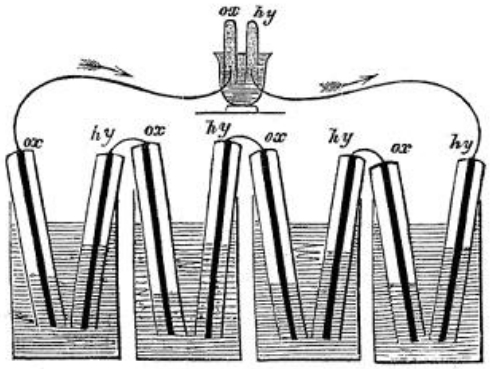


Figura 1.2: Rappresentazione schematica della cella di William Grove

Vennero effettuati numerosi esperimenti sulla varia composizione delle celle: William White Jacques sostituì l'acido solforico con acido fosforico, con scarsi risultati; nel 1932 il Dr. Francis Bacon utilizzò elettrodi in nickel (molto meno costoso del platino) e un elettrolita alcalino meno corrosivo dell'acido solforico, arrivando a sviluppare una potenza di 5 Kilowatt con la sua *Bacon Cell*; furono utilizzate celle a combustibile come generatori di energia elettrica all'interno di navicelle spaziali.
Il Dr. Lawrence H. DuBois ideò una cella a combustibile che poteva essere alimentata da diversi idrocarburi in forma liquida (come metanolo, etanolo, ecc.) e sviluppò questa tecnologia sotto il nome di DMFC (Direct-Methanol Fuel Cell), la quale ebbe una buona diffusione, limitata però dal fatto che questi combustibili alternativi richiedono un'energia di attivazione molto più alta di quella dell'idrogeno.

La reazione elettrochimica si basa infatti sull'idea di spezzare le molecole del combustibile o del comburente in ioni positivi ed elettroni: l'idrogeno si presta bene a questo tipo di reazione, dato che presenta un'energia di legame relativamente debole, ed è quindi una specie in grado di essere ionizzata con una bassa energia di attivazione. Altri combustibili sono caratterizzati da legami più forti di quelli dell'idrogeno, e conseguentemente necessitano di un'energia di attivazione molto più alta.

Il comburente tipicamente più usato è l'ossigeno presente nell'aria (O_2): ha la caratteristica di reagire con l'idrogeno liberando come prodotto di scarto acqua, ed è presente in grandi quantità nell'atmosfera. Tuttavia le sue molecole presentano un legame doppio (O=O) più energico di quello dell'idrogeno, rappresentando un ostacolo nella catalisi della reazione chimica: una parte della tensione generata dalla cella serve infatti a rompere le molecole d'ossigeno, abbassandone l'efficienza; questo fenomeno viene chiamato *sovratensione catodica*[2].

Il grande interesse per questo tipo di celle è dovuto al fatto che esse consentono di trasformare direttamente in energia elettrica l'energia chimica delle reazioni di combustione, con rendimenti che superano anche il 75%, valore molto alto se confrontato con il 35% dei motori a combustione interna che trasformano l'energia

termica in meccanica e poi elettrica. La loro applicazione è però limitata dalla difficoltà di trovare combinazioni convenienti combustibile-elettrodi-elettrolita che permettano una rapida ossidazione del combustibile.

1.3.2 Celle ad elettrolita polimerico solido (Polymer Electrolyte Membrane Fuel Cell, PEMFC)

Le caratteristiche principali di questo tipo di celle sono la possibilità di funzionare a bassi valori di temperatura e la presenza di una membrana polimerica ad alta conducibilità protonica che separa i due semielementi (vedi Figura 1.3).

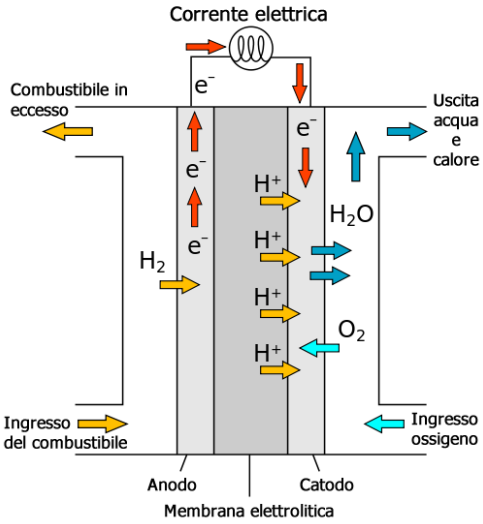


Figura 1.3: Schema di una PEM-FC

Un flusso di idrogeno viene fornito all'anodo, dove un catalizzatore provoca l'ossidazione dell'idrogeno generando ioni H^+ ed elettroni. La semireazione di ossidazione è:

$$2H_{2(g)} \rightarrow 4H^+_{(aq)} + 4e^-$$

I protoni formati attraversano la membrana polimerica arrivando al catodo, mentre gli elettroni attraversano il circuito esterno a cui è collegato un carico, generando potenza utile. Viene fornito al catodo un flusso continuo di ossigeno; qui avviene la reazione di ossidazione, dove le molecole d'ossigeno reagiscono con gli elettroni e gli ioni H^+ per realizzare la semireazione di ossidazione, generando come prodotto finale molecole d'acqua:

$$O_{2(g)} + 4H^+_{(aq)} + 4e^- \rightarrow 2H_2O_{(aq)}$$

La reazione complessiva della cella è proprio la (1.6):

$$2H_{2(g)} + O_{2(g)} \rightleftharpoons 2H_2O_{(aq)}$$

La maggior parte delle PEMFC utilizza come catalizzatori delle particelle di platino Pt supportate da carbonio poroso (Pt/C), riuscendo ad ottenere buoni risultati ma con costi elevati. Per migliorare il rendimento dei catalizzatori è possibile ottimizzare forma e dimensione delle particelle di Pt: a parità di platino utilizzato, riducendo le dimensioni delle particelle aumenta l'area totale di contatto del catalizzatore con il gas.
Esistono anche altri metodi per aumentare l'attività catalitica del catalizzatore, e questo si può ottenere utilizzando del platino in leghe con altri metalli, quali in Nickel.

Un'altra caratteristica dei catalizzatori di platino è quella di essere *avvelenati*, cioè di ridurre la loro attività, in presenza di impurezze presenti nel combustibile, e, in particolare, in presenza di quantità anche molto piccole di monossido di carbonio (CO); idrogeno ottenuto per *reforming con vapore* da idrocarburi leggeri (metanolo, etanolo, metano, ecc.) presenta quindi gradi di impurità potenzialmente nocivi per la membrana: essendo l'*avvelenamento da CO* un fattore che influenza molto la resa e la durata di una cella a combustibile, esistono molti studi che mirano ad ottenere idrogeno riformato più puro e membrane più resistenti al monossido di carbonio[4].

Le PEMFC prendono il loro nome dalla membrana polimerica che separa gli elettrodi: William Grubbs nel 1959 scoprì che la membrana riusciva ad essere permeabile agli ioni sebbene non utilizzasse acidi forti, come l'acido solforico presente nelle celle a combustibile fino a quel momento. Vari studi portarono all'attuale stato della tecnologia, dove le membrane sono prevalentemente basate su fluoropolimeri costituiti da tetrafluoroetilene sulfonato (PFSA), cui venne attribuito il nome commerciale di Nafion®. Esistono anche altri tipi di membrane, ma attualmente il Nafion® rappresenta un punto di riferimento: esso presenta una eccellente stabilità chimica, alta conduttività ionica e buona resistenza meccanica. Deve però essere mantenuto costantemente idratato per garantire la permeabilità dei protoni, e per questo la temperatura rappresenta un limite superiore: se viene utilizzato a temperature superiori al punto di fusione dell'acqua (100°C), per mantenere una corretta idratazione, la gestione del sistema che garantisce la ventilazione dell'acqua (o meglio, del vapore acqueo) è critica[3].

1.4 Serenergy® Serenus 166 Air Cooled

Il *Serenus 166 Air Cooled* della Serenergy® è il modello di cella a combustibile a membrana polimerica di cui è fornito il laboratorio di ricerca dell'Università di Padova. Il modulo consiste di uno stack contenente 66 celle elementari poste in serie[9], gestito da un sistema di controllo interno che garantisce, tra le varie cose, la corretta idratazione della membrana e l'appropriata procedura di accensione e

[9]La tensione totale in uscita dalla cella è la somma delle tensioni delle singole celle.

(a) (b)

Figura 1.4: Serenus 166 Air Cooled.

spegnimento della cella; permette inoltre di gestire il modulo tramite applicazioni sviluppate in ambiente NI LabVIEW.

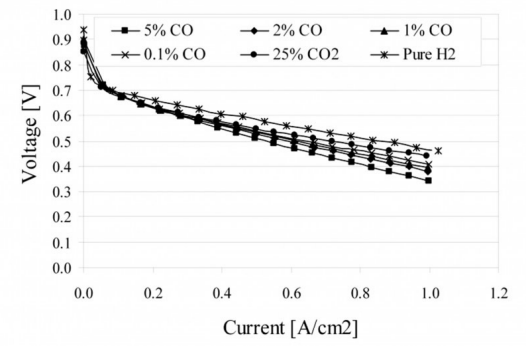


Figura 1.5: Grafico dei punti di lavoro del modulo Serenus 166 AirC per i vari tipi di combustibile

La cella Serenus 166 è un prodotto basato sulla tecnologia HT-PEMFC (High Temperature-PEMFC) sviluppata dalla Serenergy®: il modulo può operare a temperature elevate (150-170°C), fornendo una potenza di 960 W per 5000 ore di utilizzo[10]. Il modulo è rappresentato in Figura 1.3, e può essere alimentato da vari tipi di combustibile, come si evince dalla Figura 1.5.

[10]Valori nominali.

Capitolo 2

Descrizione dell'hardware utilizzato

Come visto nella sezione 1.4, la cella Serenus 166 AirC utilizza idrogeno come materiale combustibile. Essendo l'idrogeno un gas altamente esplosivo è necessaria un'attenta manipolazione del recipiente del combustibile e della cella stessa. Inoltre, dal manuale di manutenzione ed utilizzo del modulo Serenus 166 AirC si è constatata la necessità di mantenere la cella ad una umidità relativa inferiore al 20% RH: valori superiori potrebbero compromettere l'integrità della membrana polimerica, generando il rischio di pericolose perdite di combustibile ed indesiderate esplosioni. Per questi motivi si è ritenuto indispensabile provvedere all'accurato monitoraggio dei parametri ambientali (umidità e temperatura), in modo tale da rendere possibile un tempestivo intervento in caso di eccesso dei valori limite.

Vi è quindi la necessità di una trasduzione dei parametri ambientali in segnali elettrici in modo tale da poterli manipolare, mantenendoli all'interno di valori limite e, possibilmente, di renderli usufruibili in un secondo momento per un eventuale studio o rielaborazione dei dati acquisiti. Nasce perciò l'esigenza di un dispositivo capace di campionare i parametri ambientali e mantenerli in una memoria dedicata, con la possibilità di trasferirli su PC.

La soluzione prevista per questo progetto consiste nella realizzazione di uno ***shield***[1] ***per Arduino*** dove viene implementata una interfaccia utente capace di fornire tutte le caratteristiche necessarie al modulo datalogger.

L'hardware che verrà presentato è l'insieme dei componenti utilizzati nella realizzazione del modulo che provvede a soddisfare le esigenze sopracitate; lo schema a blocchi è rappresentato in Figura 2.1.

[1]Letteralmente: guscio. La vasta diffusione della scheda Arduino fu aiutata anche dalla sua struttura *modulare*. Sono infatti disponibili in commercio numerose schede che integrano hardware per svariati utilizzi, aventi le stesse dimensioni di una scheda Arduino, ed è possibile collegarli semplicemente "impilandoli" l'uno con l'altro.

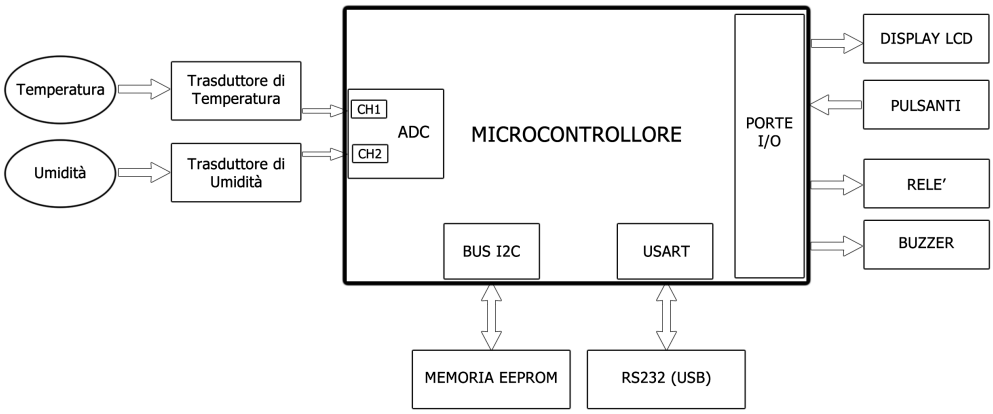
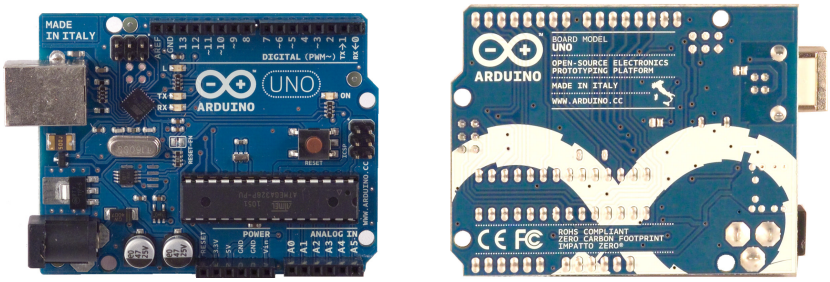


Figura 2.1: Schema a blocchi del modulo datalogger.

2.1 Arduino UNO

Arduino è una piattaforma open-source basata su risorse hardware e software flessibili e di facile utilizzo. É pensata per gli utenti con poca esperienza nel mondo dei microcontrollori, infatti, grazie alla ricca documentazione, alla comunità molto attiva e alle numerose librerie permette di prendere confidenza con questi strumenti con estrema facilità; il tutto è reso ancora più agevole dal costo contenuto e dalla licenza open-source che permette di rendere pubblici tutti i dettagli dell'implementazione fisica.
Per gli utenti più avanzati invece Arduino si rileva una buona piattaforma di prototipazione: infatti è possibile caricare il codice, fare i collegamenti necessari (spesso possibili tramite dei jumper su di una breadboard) per ottenere una rapida anteprima del risultato finale.



(a) (b)

Figura 2.2: Scheda Arduino UNO.

La scheda Arduino UNO (rappresentata in Figura 2.2) monta un microcontrollore (MCU - MicroController Unit) Atmel® ATmega328P[2], e rende accessibili tutti i pin del micro su degli header laterali. La scheda è fornita di un regolatore

[2]Vedi sezione 2.2.

di tensione a 5 Volt che fornisce l'alimentazione adeguata alla mcu, ed è riportata ad uno degli header. É presente inoltre un secondo microcontrollore (un Atmel® ATmega8) che gestisce la conversione seriale[3]-USB: grazie a questo chip è possibile collegare la presa USB della scheda ad un PC, e, dopo aver installato un driver apposito, Arduino viene visto come un dispositivo seriale con cui è possibile comunicare tramite protocollo RS-232, nativo su ogni PC (porta COM).

All'hardware presente sulla scheda viene affiancato un ambiente di sviluppo integrato (IDE - Integrated Development Envinronment) multipiattaforma, che rende possibile la scrittura di programmi (più correttamente *firmware*) in maniera facilitata grazie all'uso di una vasta collezione di librerie. Il linguaggio di programmazione per questo ambiente di sviluppo viene chiamato *Wiring*, ed è una derivazione dei più famosi C e C++.

La programmazione del micro avviene attraverso l'interfaccia USB e alla GUI (Graphical User Interface) sviluppata appositamente; l'interfaccia grafica permette inoltre di scrivere sul microcontrollore un programma[4], il *bootloader*, tramite il quale avviene l'upload del firmware sul micro.

2.2 Microcontrollore Atmel® ATmega328P

Un microcontrollore è un sistema di elaborazione completo ottimizzato per il controllo dell'hardware, e incapsula nello stesso pezzo di silicio un intero processore, la memoria necessaria e tutte le periferiche di I/O. Essendo i componenti presenti nello stesso chip, le prestazioni sono più alte di una configurazione che preveda gli stessi componenti su singoli IC; questo perché le periferiche di I/O necessitano di tempi minori in lettura e scrittura[10].

Il microcontrollore ATmega328P è il core di Arduino. Appartiene all'architettura AVR, famiglia di microprocessori RISC (Reduced Instruction Set Computing) a 8 bit, derivante dall'architettura Harvard e sviluppata dalla Atmel® nel 1996. Questa famiglia fu una delle prime a memorizzare il programma da eseguire sulla memoria FLASH interna al chip, contrariamente a quello che avveniva negli altri microcontrollori, dove il codice veniva salvato in una memoria esterna.

L'ATmega328P è un microcontrollore in tecnologia CMOS a 8 bit, che raggiunge un throughput di 20 MIPS (Million of Instructions Per Second) ad una velocità di clock di 20 MHz; è capace di eseguire 131 istruzioni, e conta 32 registri a 8 bit. Incorpora nello stesso chip le memorie necessarie all'esecuzione: una FLASH da 32 KB utilizzata per mantenere il codice, una EEPROM da 1 KB e 2 KB di memoria SRAM. Il suo *pinout* è rappresentato in Figura 2.3.

L'architettura AVR fornisce svariate periferiche, tra le quali:

[3]La comunicazione seriale avviene con la periferica USART dell'ATmega328p.

[4]Questa operazione deve avvenire tramite un programmatore esterno.

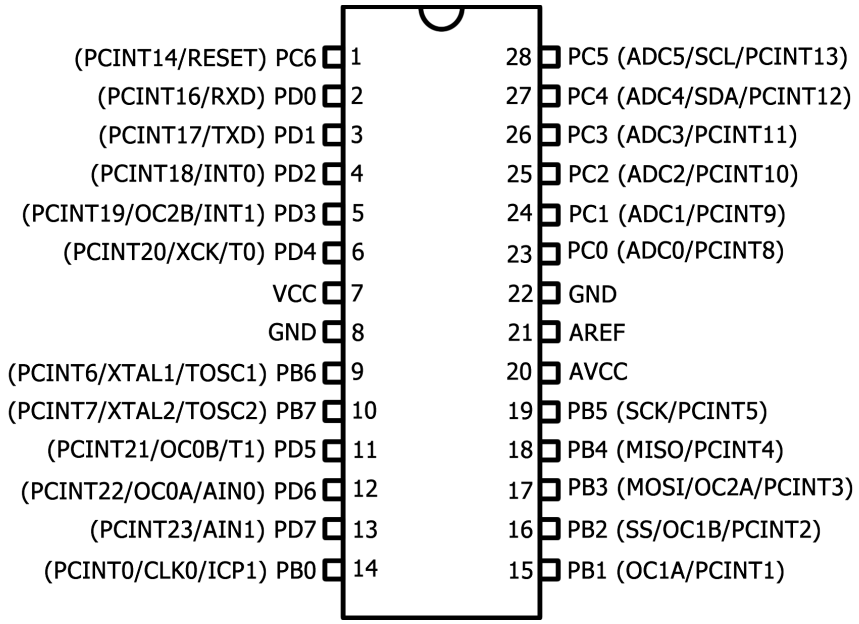


Figura 2.3: Pinout dell'ATmega328P, versione P-DIP

- Numerose porte di I/O (Input/Output), con resistori di pull-up programmabili.
- Oscillatori interni, tra cui un oscillatore RC capace di generare un segnale di clock senza alcun componente esterno.
- Memoria interna *self-programmable* attraverso interfaccia JTAG proprietaria.
- Supporto all'On-Chip Debugging attraverso JTAG[5].
- Memoria FLASH, EEPROM e SRAM interne al chip stesso.
- Timer a 8 e a 16 bit: forniscono uscite PWM[6] e vengono usati per ottenere la durata di intervalli di tempo.
- Amplificatore differenziale usato come comparatore analogico: la variazione della sua uscita può far partire un evento di interrupt.
- Convertitori Analogico-Digitale (ADC) a 10 o 12 bit, con la possibilità di *multiplexare* l'ingresso su vari canali (fino a 16).
- Interfacce seriali come

[5]Possibilità di fare il debug del dispositivo durante l'esecuzione, ovvero di far eseguire al micro una istruzione alla volta e la possibilità di cambiare il contenuto dei registri e il valore dei pin.

[6]Pulse Width Modulation: tecnica di modulazione che modifica la larghezza degli impulsi in base all'informazione da trasmettere.

  - I^2C^{TM} compatibile con TWI (Two Wire Interface).
  - Synchronous/Asynchronous serial peripheral (USART/UART), dialoga con protocolli RS-232, RS-485 e altri.
  - Serial Peripheral Interface (SPI)

- Rilevazione di cadute di tensione nell'alimentazione (Brownout detection).
- Watchdog timer: contatore utilizzato per rilevare malfunzionamenti nel programma o nel micro.
- Numerose modalità di risparmio energetico e *sleep-modes*.

Questo tipo di architettura è provvista inoltre di un **sistema di interruzioni**, caratteristica molto utile nel mondo *embedded* dove si cerca di mantenere minimo il numero di componenti, costringendo così un microcontrollore a gestire simultaneamente più periferiche e dispositivi.
Il sistema è di tipo a interrupt vettorizzati (*vectored*), dove il riconoscimento dell'interruzione (ovvero capire *quale* evento ha lanciato l'interruzione) viene eseguito direttamente dall'hardware e non dal software[6]. Nello specifico, al verificarsi di un'interruzione ogni dispositivo (se interno alla mcu) provvede ad *alzare il flag* corrispondente; il micro, una volta stabilito quale dispositivo ha generato la chiamata di interruzione, colloca l'indirizzo presente nell'*interrupt vector* adeguato nel PC (Program Counter): tale operazione garantisce che la prima istruzione eseguita a seguito della chiamata sia proprio il gestore dell'interrupt che si è verificata. Tutto questo avviene però se l'*Interrupt Enable Bit* è abilitato; se così non fosse, l'evento di interrupt verrebbe ignorato.

2.3 Microchip® 24AA1025 1 Mbit EEPROM

Memoria EEPROM (Electrically Erasable Programmable Read-Only Memory) seriale dotata di interfaccia I^2C^{TM}. Fabbricata in tecnologia CMOS, garantisce bassi consumi e clock fino a 400 KHz; contiene 128 x 8 (1024 K) bit, ovvero 8 *pagine* da 128 bit ciascuna. Permette la scrittura a byte singolo oppure di una pagina intera; è possibile eseguire letture casuali (una qualsiasi locazione di memoria) o sequenziali, queste ultime però limitate agli intervalli `0000h-FFFFh` e `10000h-1FFFFh`.

É possibile collegare fino a quattro dispositivi simili sullo stesso bus I^2C^{TM}, arrivando ad una memoria totale di 4 Mbit: per fare questo è necessario impostare un *indirizzo di bus* specifico per ogni dispositivo, agendo sui pin A0 e A1[7]. Differenti combinazioni dei pin corrispondono a differenti indirizzi: all'inizio di

[7] Fisicamente, questi vengono collegati a massa o alla tensione di alimentazione.

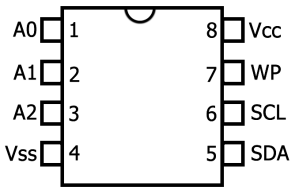


Figura 2.4: Pinout del dispositivo 24AA1025

ogni comunicazione il *master* (il dispositivo che inizia la trasmissione) invia nel byte di controllo una sequenza comprendente i bit di indirizzo, e tutti i dispositivi presenti sul bus (gli *slave*) leggono questo byte; se l'indirizzo comunicato dal master corrisponde a quello impostato fisicamente nei pin A0 e A1 di un determinato dispositivo, questo risponde al master e prosegue la comunicazione. Se il master comunica un indirizzo al quale non è riferito nessun dispositivo, il messaggio viene ignorato da tutti gli slave presenti sul bus. Un esempio di byte di controllo inviato sul bus I^2C^{TM} è rappresentato in Figura 2.5.

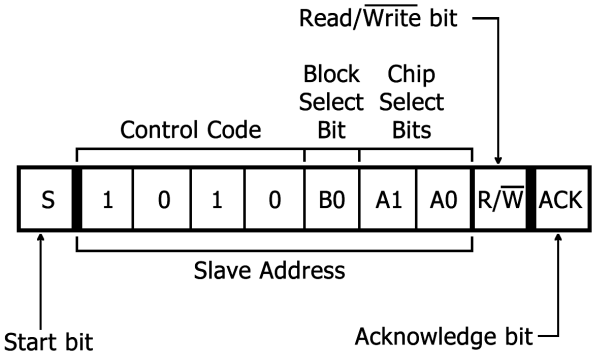


Figura 2.5: Formato del byte di controllo: si noti come i bit di indirizzamento influiscono su questo byte.

2.4 Honeywell HIH-4000-001

Sensore di umidità integrato: fornisce in uscita una tensione che è funzione dell'umidità dell'ambiente in cui si trova. La tensione non è lineare nel parametro umidità, e per ottenere il valore desiderato è necessario fare un *fit* dei valori di uscita; la formula usata per l'interpolazione è

$$V_{out} = V_{supply} * (0.0062 * RH_{sensor} + 0.16), \tag{2.1}$$

per un valore di temperatura di 25°C. Da questa formula ricaviamo quindi il valore di umidità in %RH:

$$RH_{sensor} = \frac{V_{out}}{V_{supply}} - 0.16, \tag{2.2}$$

che equivale ad una costante di proporzionalità di 31.483 mV/RH.

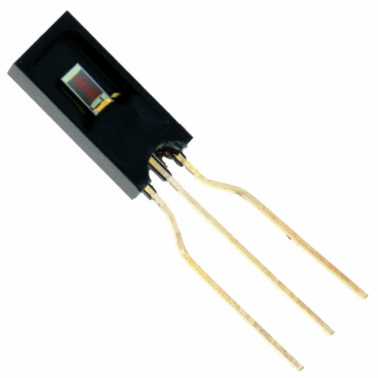


Figura 2.6: Immagine del sensore HIH-4000-001.

Nel caso in cui la temperatura si discosti molto da quella standard di 25°C viene fornita una formula per la compensazione:

$$RH_{true} = \frac{RH_{sensor}}{1.0546 - 0.00216 * T}, \tag{2.3}$$

con T valore della temperatura in °C.

Il dispositivo presenta una precisione del ± 3.5 % RH allo stato standard di utilizzo di 25°C e V_{supply}=5 Volt. Il tempo di assestamento è di 70 ms e la risposta in aria quasi stazionaria è di 5 secondi. Richiede per funzionare una corrente di circa 200 μA, e può essere utilizzato in un range di valori di temperatura che spazia dai -40°C a 85°C.

É molto sensibile alle scariche elettrostatiche, e il circuito di utilizzo prevede una resistenza di almeno 80 KΩ connessa dal piedino di uscita a quello di massa.

2.5 Texas Instruments LM35CAZ

Sensore di temperatura integrato, a differenza dell'HIH-4000-001 fornisce una tensione di uscita direttamente proporzionale alla temperatura (in °C) dell'ambiente in cui è immerso, con costante di proporzionalità di 10mV/°C. Non richiede nessuna calibrazione esterna per fornire la precisione di $\pm 1/4$ LSB a temperatura ambiente, e di $\pm 3/4$ LSB nell'intervallo da -50 a 150 °C.

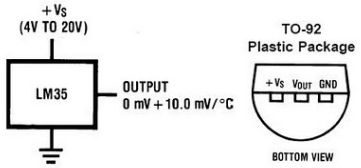


Figura 2.7: Pinout del sensore LM35.

L'assorbimento di corrente è di circa 60 μA, garantendo così un *auto-riscaldamento* molto basso, minore di 0.1 °C in aria ferma o quasi-stazionaria.

Capitolo 3

Sviluppo del software di controllo

Il software sviluppato per la gestione del modulo datalogger permette il controllo del dispositivo tramite l'interfaccia visibile sul display LCD, dentro la quale è possibile "navigare" tramite i tre pulsanti presenti sullo shield. Da questo si possono impostare ora, data, soglia di accensione del deumidificatore e soglia di allarme; è possibile gestire il collegamento con l'applicazione su PC, visualizzare lo stato di utilizzo della memoria ed effettuarne la cancellazione.

L'interfaccia grafica si differenzia in vari "livelli":

- Livello 0: Funzionamento *idle*[1]. Vengono stampate data, ora, e i valori istantanei di temperatura e umidità.
- Livello 1: Menu. In questo livello è possibile scorrere le varie opzioni implementate nell'applicazione.
- Livello 2: Opzione scelta nel menu del livello 1.

É possibile passare dal funzionamento *idle* al menu premendo il tasto **C** per almeno un secondo (pressione prolungata); si scorrono le varie voci del menu con i pulsanti **A** e **B**, e si entra nell'opzione scelta tramite la pressione del tasto **C**. All'interno delle varie opzioni è poi possibile confermare la propria scelta tramite il relativo *dialogo di conferma*.

In seguito verranno descritte in maggior dettaglio le modalità di implementazione dell'interfaccia e le librerie necessarie per realizzarla.

[1] Letteralmente: inattivo, *disoccupato*.

3.1 Gestione del microcontrollore

3.1.1 Multitasking con prelazione

La creazione di una UI è resa complessa dal fatto che l'utente è *molto più lento* del microcontrollore: per questo è impossibile pensare che il micro debba aspettare l'intervento dell'utilizzatore per l'avanzamento del programma caricato. É stato quindi necessario prevedere un flusso di controllo che restasse consistente anche indipendentemente dall'input dell'utente; ovvero, le funzionalità del micro non devono essere bloccate dalla *lentezza* dell'utilizzatore, e deve essere possibile gestire eventi a lui esterni in maniera trasparente a quest'ultimo. In particolare, se l'utente sta navigando nel menu, il micro non deve essere costretto ad aspettare che il suo intervento sia concluso per poter leggere i dati relativi ai sensori; se così fosse, potrebbe essere saltata una scansione che avrebbe fatto accendere il deumidificatore o, peggio, l'allarme.

Per questo motivo, e perché la programmazione di un microcontrollore è solitamente di questa forma, si è deciso per una gestione del software *ad interrupt*: quando un evento esterno (o anche interno, vedi il timeout di un timer o la necessità di inviare un byte alla usart) lo richiede, il micro blocca l'esecuzione del programma principale per passare il controllo alla ***ISR*** (Interrupt Service Routine) chiamante, la quale farà eseguire le istruzioni necessarie per gestire l'evento scatenante l'interruzione[2].

Quando il controllo è passato ad una routine, di norma l'esecuzione è resa *non interrompibile*, cosicché il controllo venga restituito al programma originale solo quando la ISR ha terminato il blocco di istruzioni a lei relativo. Questo tipo di politica è detta *multitasking con prelazione*, ovvero ***cooperative multitasking***. La politica opposta stabilisce che una routine debba passare il controllo ad una seconda routine chiamante, se quest'ultima ha *priorità più alta*; quest'ultimo tipo di programmazione è detto *multitasking senza prelazione*.

3.1.2 Flusso di controllo

La programmazione di un microcontrollore come cuore di un progetto il cui scopo è quello di eseguire indefinitamente il compito prefissato, si traduce nell'esistenza di un ciclo infinito, e nella ripetizione continua di determinate istruzioni. Ciò che cambia nelle diverse fasi dell'esecuzione è lo ***stato*** del sistema: per tenerne traccia si utilizza una variabile interna (`bState`[3]), la quale assumerà valori diversi in funzione dello stato. La funzione `main()` guarda il valore di `bState` e decide in quale ramo del programma l'esecuzione dovrà continuare: stampare le *quote*[4], visualizzare il menu o far partire l'ADC sono solo alcune delle possibilità. Lo stato

[2] Vedi sezione 2.2 per una descrizione sul tipo di interrupt presenti sull'ATmega328P.

[3] Viene usata la convenzione secondo la quale al nome delle variabili deve essere anteposto il *tipo*: in questo caso, la variabile `bState` che indica lo stato del sistema è di tipo *byte*.

[4] Data, ora, temperatura ed umidità.

viene cambiato durante l'esecuzione: dopo essere entrati in un *branch* (ramo) potrebbe esservi la necessità di eseguirne completamente un altro, senza passare alla modalità *idle*, dove tutto ciò che il micro fa è ciclare senza sosta aspettando l'interazione dell'utente.

```
int main(void){

    _init_AVR();                // Inizializzazione del dispositivo

    while(1) {                  // Ciclo infinito

        switch( bState ){
            case STATE_IDLE:
              ...
            case STATE_MENU:
              ...
            case STATE_EDIT_DATE:
              ...
            case STATE_EDIT_TIME:
              ...
            case STATE_EDIT_HUM_ON_TH:
              ...
            case STATE_EDIT_HUM_AL_TH:
              ...
            case STATE_LOG_DATA:
              ...
            case STATE_ALARM:
              ...
            case STATE_COMM:
              ...
            case STATE_PRINT_N_DAYS_LOGGED:
              ...
            case STATE_PRINT_EEPROM_USAGE:
              ...
            case STATE_EEPROM_ERASE:
              ...
            default:
                break;
        }
    }
}
```

Listing 3.1: Possibili *branch* in cui il programma può entrare durante la sua esecuzione.

Ognuno di questi rami poi valuta se l'utente ha inserito dell'input, nel qual caso esso va gestito opportunamente (input da parte dell'utente corrisponde alla pressione breve o prolungata di uno dei tasti di controllo). Si tiene traccia dell'immissione dell'input grazie alla variabile `bBtn`, che assume il valore `NO_BTN` quando non è stato premuto nessun tasto; per una pressione *corta* assume i valori `BTN_X`, mentre per una pressione *lunga* i valori `BTN_X_LONG`, con X=A, B o C.

Il caso `bState == STATE_IDLE && bBtn = NO_BTN` (vedi listato 3.2, primo caso) corrisponde quindi al funzionamento idle, dove non ci sono azioni da eseguire. Se invece `bBtn` assume valori diversi da `NO_BTN`, il micro intraprenderà l'azione adeguata: se è avvenuta una pressione corta di un qualsiasi tasto, provvederà ad

accendere la retroilluminazione del display; se `bBtn == BTN_C_LONG` il micro cambia il valore della variabile di stato eseguendo l'istruzione `bState = STATE_MENU`, portando il processore nel branch della visualizzazione del menu. Dopodiché provvede a resettare il valore della variabile di input con `bBtn = NO_BTN`; se così non fosse, al successivo ciclo il micro penserebbe erroneamente che dell'altro input dovrebbe essere gestito.

```
case STATE_IDLE:

    switch( bBtn ){
        case NO_BTN:
            if( bTimeColonToToggle ){
                toggleTimeColon();
                bTimeColonToToggle=0;
            }

            refreshQuote();     // aggiorna i valori sul display
            bPrintQuotes=1;
            break;

        case BTN_A:
        case BTN_B:
        case BTN_C:
        case BTN_A_LONG:
        case BTN_B_LONG:
            RESTART_BACKLIGHT();    // avvia la retroilluminazione
            bBtn=NO_BTN;            // resetta l'input
            break;

        case BTN_C_LONG:
            bState = STATE_MENU;    // cambia lo stato
            STOP_BACKLIGHT();
            BACKLIGHT_ON();         // accende il display
            bBacklightActive=1;
            bBtn=NO_BTN;
            break;

        default:
            break;
    }
break;
```

Listing 3.2: Listato del *branch* relativo a `bState==STATE_IDLE`.

Si prende ora in esame il caso in cui `bState==STATE_MENU` (vedi listato 3.3). In questa situazione il micro ha il compito di stampare il menu, e di gestire la navigazione delle opzioni tramite la pressione dei tasti **A** e **B**: nel primo caso, viene incrementata la variabile relativa all'opzione da stampare; nel secondo la si diminuisce. Se invece viene premuto il tasto **C**, l'utente ha scelto un'opzione del menu, ed è necessario portare il sistema nel branch relativo alla scelta fatta.

Nel listato 3.3 è possibile vedere quali sono le opzioni che l'utente ha a disposizione:

- Modifica della data.
- Modifica dell'ora.
- Modifica della soglia di attivazione del deumidificatore.
- Modifica della soglia di attivazione dell'allarme.
- Inizio comunicazione seriale con l'applicazione su PC.
- Stampa del numero di giorni *loggati*, ovvero salvati in memoria.
- Visualizzazione della percentuale di memoria EEPROM utilizzata.
- Cancellazione della memoria EEPROM.

```
case STATE_MENU:
    switch( bBtn ){
        case NO_BTN:
            if( bSelectionMenuChanged || bPrintQuotes ){
                bPrintQuotes=0;
                bSelectionMenuChanged=0;
                lcd_putsXY(0,0,"-");
                lcd_putsXY(1,0, options[bSelectionMenu]);
                lcd_putsXY(0,1,"␣");
                lcd_putsXY(1,1, options[bSelectionMenu+1]);
            }
            break;

        case BTN_A:
            bSelectionMenu++;
            bSelectionMenu %= NUMBER_OF_OPTIONS;
            bSelectionMenuChanged=1;
            bBtn=NO_BTN;
            break;

        case BTN_B:
            if( bSelectionMenu>0 ) bSelectionMenu--;
            else bSelectionMenu=(NUMBER_OF_OPTIONS-1);
            bSelectionMenu %= NUMBER_OF_OPTIONS;
            bSelectionMenuChanged=1;
            bBtn=NO_BTN;
            break;

        case BTN_C:
            switch( bSelectionMenu ){
                case SEL_DATE:
                    bState = STATE_EDIT_DATE;
                    break;
```

```
                case SEL_TIME:
                    bState = STATE_EDIT_TIME;
                    break;

                case SEL_HUM_TH_1:
                    bState = STATE_EDIT_HUM_ON_TH;
                    break;

                case SEL_HUM_TH_2:
                    bState = STATE_EDIT_HUM_AL_TH;
                    break;

                case SEL_COMM:
                    bState = STATE_COMM;
                    bCOMState = COM_WAIT;
                    break;

                case SEL_PRINT_N_DAYS:
                    bState = STATE_PRINT_N_DAYS_LOGGED;
                    break;
                case SEL_PRINT_EEPROM_USAGE:
                    bState = STATE_PRINT_EEPROM_USAGE;
                    break;

                case SEL_EEPROM_ERASE:
                    bState = STATE_EEPROM_ERASE;
                    break;

                default:
                    break;
            }
            bBtn = NO_BTN;
            bPrintQuotes=1;
            break;

        case BTN_C_LONG:
            bState = STATE_IDLE;
            LCD_RESET();

            bSelectionMenu=0;

            // Appena rientra in idle stampa le quote
            bDateChanged=1;
            bTimeChanged=1;
            bTempChanged=1;
            bHumChanged=1;
            bPrintQuotes=1;

            bBacklightActive=0;
            BACKLIGHT_OFF();
            START_BACKLIGHT();
            bBtn=NO_BTN;
            break;

        default:
```

```
            break;
    }
break;
```

Listing 3.3: Listato del *branch* relativo al caso `bState==STATE_MENU`.

Come descritto nel paragrafo 3.1.1, eventi esterni o periferiche interne possono generare delle chiamate alle relative ISR in qualsiasi momento; quando il sistema si trova all'interno della funzione `main()`, questo è ***sempre interrompibile***[5]: ciò significa che le ISR devono essere scritte in maniera tale da non causare involontari *stalli di sistema*, ma, una volta concluse, devono poter rendere possibile al micro il suo ritorno alla normale esecuzione. Per facilitare questo compito vengono utilizzate numerose variabili di tipo `volatile`, variabili il cui spazio di visibilità comprende sia la normale esecuzione che quella all'interno delle ISR: grazie a queste il micro riesce a *ricordarsi da dove era venuto*, rendendo così il sistema più affidabile.

3.2 Interrupt Service Routine principali

3.2.1 TIMER0

Il *timer* di un AVR è una periferica che, come dice il nome, *scandisce il tempo*: ad ogni ciclo di clock incrementa (o decrementa) un contatore, e quando arriva ad un determinato valore (oppure a zero, partendo da un valore preimpostato) genera un'interruzione di tipo *timer overflow* [6]. Il clock non è quello principale del micro, ma una sua versione scalata[7], e il valore limite è contenuto in un registro (nell'ATmega328P questo registro è chiamato `TCNT0`).

Nel datalogger, il ***timer0*** è impostato per generare un'interruzione ogni 10 millisecondi: tenendo conto di quante interrupt sono avvenute (incrementando cioè un contatore ogni qualvolta si verifica un'interrupt generata dall'*overflow* del timer) è possibile tenere traccia del tempo trascorso durante l'esecuzione del programma. Questa capacità è utilizzata per effettuare il *debounce dei tasti*: quando un pulsante viene premuto, quella che si crede una singola pressione è in realtà una sequenza più o meno variabile di *commutazioni spurie* del tasto, che risultano in numerose pressioni, prima di stabilizzarsi nello stato finale, come si può vedere dalla Figura 3.1. Per ignorare queste false pressioni è necessario filtrare il segnale d'ingresso proveniente dal tasto. Il filtraggio può essere fatto tramite hardware (filtri RC passa-basso) o tramite software.

[5] A meno che le interrupt non siano state disabilitate con l'istruzione `cli()`, *clear interrupts*.

[6] Un timer può, alternativamente, cambiare lo stato di un pin: in questa modalità fornisce la funzione di *generatore di forme d'onda PWM*.

[7] Il clock del timer è prelevato da quello principale, scalato di 1, 8, 64, 256 o 1024 volte.

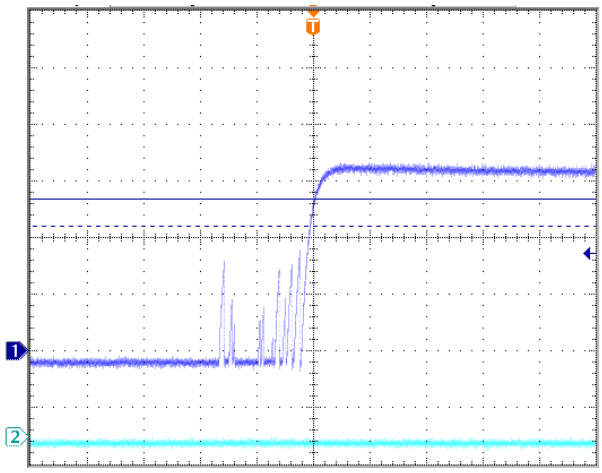


Figura 3.1: Commutazioni spurie dovute alla singola pressione di un tasto.

Il filtro di *debounce* realizzato in questo progetto fa si che un tasto venga considerato ***premuto*** solo quando il valore di tensione risulta alto (e quindi stabile) per un tempo maggiore di 80 ms[8].

3.2.2 TIMER1

Questa routine è impostata per generare un'interruzione ogni secondo[9], ed è utilizzata per mantenere il calendario interno al dispositivo. Il codice per ottenere ciò è il seguente:

```
if( tTime.bSec<59 ){
    tTime.bSec++;

    // il separatore ora/minuti sul display lampeggia ogni secondo
    bTimeColonToToggle=1;
}else{
    tTime.bSec=0;
    if( tTime.bMin<59 ){
        tTime.bMin++;
    }else{
        tTime.bMin=0;
        if( tTime.bHour<23 ) tTime.bHour++;
        else {
            tTime.bHour=0;
            if(tTime.bDay<(baDays[tTime.bMonth-1])){
                tTime.bDay++;
                if(tTime.bDay==29 && tTime.bMonth==2 &&
                                && (!isLeapYear(tTime.bYear))){
                    tTime.bDay=1;
```

[8] Valore ottenuto sperimentalmente.

[9] In realtà ottenere un'interruzione ogni 1000ms si è rivelato pressochè impossibile: il miglior risultato ottenuto è un discostamento di 1 secondo ogni 4 ore, in riferimento all'orologio di un PC.

```
                    tTime.bMonth=3;
                }
            }else{
                tTime.bDay=1;
                    if(tTime.bMonth<12) tTime.bMonth++;
                    else{
                        tTime.bMonth=1;
                        tTime.bYear++;
                    }                           // mesi
            }                                   // giorni
            bDateChanged=1;
            }                                   // ore
    }                                           // minuti
    bTimeChanged=1;     // refresh del display ogni minuto

}                                               // secondi
```

Listing 3.4: Codice per il calendario interno.

La struct `tTime` serve per mantenere le variabili temporali, ed è così definita:

```
typedef struct{
    word wMilli;
    byte bSec;
    byte bMin;
    byte bHour;
    byte bDay;
    byte bMonth;
    byte bYear;
} time_date;

volatile time_date tTime;
```

Listing 3.5: Definizione della struct tTime.

3.2.3 Analog to Digital Converter

La periferica ADC ha varie modalità di funzionamento: *Free Running Mode* e *Single Conversion Mode*. Nella prima modalità, l'ADC viene fatto partire e campiona ripetutamente l'ingresso selezionato, generando un segnale di interrupt alla fine di ogni conversione; nella seconda modalità, invece, dopo aver fatto la campionatura voluta la periferica si arresta ed attende che le venga nuovamente dato il "via" (settando ad 1 il bit `ADSC` nel registro `ADCSRA`). É questa seconda modalità quella usata nel progetto; inoltre, è buona pratica scartare la prima conversione perché imprecisa[13].

L'ATmega328P è fornito di un ADC a 6 canali: ciò significa che è possibile campionare il valore di tensione su 6 pin diversi, gestendo ovviamente il *channel switching* tramite software.

Il codice per questa routine è il seguente:

```
if(bFirstConversion){
        bFirstConversion=0;
        ADCSRA |= 1<<ADSC;
        return;                 // scarta il primo campione
    }


    float fTemperatureOld;
    float fHumidityOld;

    switch(bChannel){
        case ADC_TEMPERATURE_CHANNEL:
            fTemperatureOld = fTemperature;

            // chiama la funzione per calcolare la temperatura
            fTemperature = getTemperature();

            if(fTemperatureOld != fTemperature) bTempChanged=1;

            // da il via alla conversione nel secondo canale
            ADCSRA |= 1<<ADSC;

            // alla prima campionatura il canale selezionato è quello
            // dell'umidità
            ADC_SET_HUMIDITY_CHANNEL();


            bChannel = ADC_HUMIDITY_CHANNEL;
            bFirstConversion=1;
            break;

        case ADC_HUMIDITY_CHANNEL:
            fHumidityOld = fHumidity;
```

```
            // chiama la funzione per il calcolo dell'umidità
            fHumidity = getHumidity(fTemperature);

            if(fHumidityOld != fHumidity) bHumChanged=1;

            ADC_SET_TEMPERATURE_CHANNEL();
            bChannel = ADC_TEMPERATURE_CHANNEL;
            bFirstConversion=1;


            // manda il sistema nel branch di memorizzazione del dato
            bState = STATE_LOG_DATA;

            break;

        default: break;

    }
```

Listing 3.6: ADC Interrupt Service Routine.

```
#define TEMP_SENSOR_GAIN    0.01    // dal datasheet del LM35
#define VREF                5.0     // tensione di riferimento

float getTemperature(){
    float temp;
    float fVadc;

    // ADC: valore ottenuto dal campionamento
    fVadc = ADC * VREF/1024;

    temp = fVadc / TEMP_SENSOR_GAIN;


    return temp;
}
```

Listing 3.7: Listato per `getTemperature()`.

```
float getHumidity(float temperature){
    float fVadc0;
    float fRH;
    float fRH_comp;

    fVadc0 = ADC * VREF/1024;
    fRH = (fVadc0/VREF - 0.16) / 0.0062;
    fRH_comp = fRH/(1.0546-0.00216*temperature);

    return fRH_comp;
}
```

Listing 3.8: Listato per `getHumidity(float temperature)`. Le formule sono ricavate dal datasheet del HIH-4000-001: vedi sezione 2.4.

3.2.4 USART0_RX

La routine viene eseguita quando il modulo di ricezione della periferica USART ha ricevuto un byte. Questa situazione si verifica quando il modulo datalogger viene collegato tramite USB al computer, e viene avviata la sequenza di trasferimento dati: dopo aver messo il modulo in modalità *connessione* e dopo aver avviato l'interfaccia su PC (SENSE-GUI), alla pressione del pulsante ***trasferimento dati*** l'interfaccia in Windows provvede ad inviare un byte di controllo al microcontrollore; se la connessione è stabilita correttamente il micro dovrà rispondere con lo stesso byte ricevuto. Per questo, l'unico compito che questa routine deve svolgere è ritrasmettere il byte appena ricevuto:

```
wRx = UDR0;
UDR0 = wRx;
```

Listing 3.9: ISR relativa alla ricezione di un byte da parte del modulo USART.

`UDR0` è il registro che contiene il byte ricevuto; nel microcontrollore AVR per inviare un byte tramite interfaccia seriale è sufficiente scriverlo nello stesso registro. Queste due righe quindi effettuano la ritrasmissione del byte appena ricevuto. Sarà il software di controllo (fuori dalle routine) a gestire correttamente la connessione per quanto riguarda il microcontrollore: infatti, quando la connessione tra le due interfacce è stabilita, l'applicazione Windows può inviare la richiesta di trasferimento dati sotto forma di un altro byte di controllo; se il byte ricevuto è quello che l'applicazione nel microcontrollore si aspetta, allora procederà ad inviare il contenuto della memoria, altrimenti il comando verrà reputato come sconosciuto, ed ignorato.

3.3 Struttura della memoria di log

Per il mantenimento dei campioni in memoria si è deciso per una struttura dove ogni giorno (***daily_log***) viene salvato in memoria come una struct così definita:

```
// Misurazione ogni 30 minuti
#define MINS_BETWEEN_SAMPLES        30

// Numero di campioni (log) giornalieri
#define NUMBER_OF_LOGS_PER_DAY      24*60/MINS_BETWEEN_SAMPLES

typedef struct{
    byte bDay;
    byte bMonth;
    byte bYear;
} date;

typedef struct{
    date dDate;
    float fHumValues[NUMBER_OF_LOGS_PER_DAY];
    float fTempValues[NUMBER_OF_LOGS_PER_DAY];
}daily_log;
```

Così facendo, per ogni giorno campionato si salvano in memoria un *header* contenente giorno, mese e anno del log, e successivamente una serie di campioni relativi ad umidità e temperatura.

Per salvare questo tipo di struttura in memoria, però, è necessario avere tutti i campioni dello stesso giorno, cosa impossibile da ottenere per ogni campione; infatti, alle 10 di mattina (per esempio) non si conoscono i valori dei campioni delle ore successive. Questo comporta l'impossibilità di salvare i valori in memoria fino al completamento dell'intera struct: per questo motivo tale struttura è rimasta solo concettuale, e si è deciso di utilizzarne una alternativa, anche se molto simile.

Questa nuova struttura mantiene l'*header* come la precedente, ma salva ogni dato campionato non appena la conversione è avvenuta: si riesce così ad ottenere un salvataggio in memoria sempre aggiornato, e non vi è il rischio di perdere importanti misurazioni.

La struttura di salvataggio dei campioni prevede anche la presenza di alcuni byte di controllo, posti nelle prime locazioni della memoria esterna: in questi byte vengono mantenute le informazioni aggiornate riguardanti la data dell'ultimo giorno salvato in memoria, il numero di campioni da salvare prima di "chiudere" la struct ***daily_log***, il valore delle soglie impostate, il numero di giorni completati e l'indirizzo dell'ultimo campione salvato in memoria. Questi dati servono per mantenere sincronizzato il microcontrollore con la memoria, per avere un riferimento immediato in caso di necessità e, cosa molto importante, per permettere il ripristino del sistema in caso di blackout. Infatti, grazie a questa struttura e ad un'oculata programmazione, in caso di stallo il sistema provvede a riportarsi *all'ora esatta dell'ultimo campione salvato in memoria*: in questo modo si evita di perdere i campioni rilevati dopo il blackout, e lo scarto massimo tra ora effettiva e del sistema è al massimo quella che intercorre tra due campioni consecutivi[10].

I dati in memoria sono salvati come `float`, ed occupano 4 byte per ogni campione: così, una struct ***daily_log*** contenente 48 campioni di umidità, 48 di temperatura e 3 byte di header, contiene un totale di 3+48*2*4 = 387 byte.
La memoria 24AA1025 (vedi sezione 2.3) contiene 1 Mbit = 131072 byte, e può contenere fino a 338 giorni di campioni rilevando i due parametri ogni mezz'ora.

```
case STATE_LOG_DATA:     // si arriva in questo branch
                         // solo dalla ISR ADC_vect

    _CLI(sreg); // rende il processo non interrompibile

    // bLogData: indica quando la misura effettuata deve essere
    // salvata in EEPROM; utile nel caso si volesse rilevare i dati
    // ambientali MA NON SALVARLI!
    if( bLogData ){

        if(++bTodayLogs > NUMBER_OF_LOGS_PER_DAY){
            // raggiunto il numero massimo di log giornalieri
            // (completata la struct daily_log)
```

[10] Questo valore è deciso in fase di progetto, ed è impostato a 30 minuti.

```
            bTodayLogs=1;
            wLoggedDays++;

            EEPROM_writeByte(EEPROM_DAY_ADD, tTime.bDay);
            EEPROM_writeByte(EEPROM_MONTH_ADD, tTime.bMonth);
            EEPROM_writeByte(EEPROM_YEAR_ADD, tTime.bYear);

            EEPROM_writeData(EEPROM_LOGGED_DAYS_ADD,
                (byte*)&wLoggedDays, sizeof(word));
        }

        if(bTodayLogs == 1){
            EEPROM_writeByte(lLastIndex++, tTime.bDay);
            EEPROM_writeByte(lLastIndex++, tTime.bMonth);
            EEPROM_writeByte(lLastIndex++, tTime.bYear);
        }
        EEPROM_writeData(lLastIndex, (byte*)&fHumidity,
                                    SIZE_OF_LOG);
        lLastIndex += SIZE_OF_LOG;
        EEPROM_writeData(lLastIndex, (byte*)&fTemperature,
                                    SIZE_OF_LOG);
        lLastIndex += SIZE_OF_LOG;



        // Aggiorno todayLogs, lLastIndex e ora ad ogni campionamento.
        EEPROM_writeByte(EEPROM_TODAY_LOGS_ADD, bTodayLogs);
        EEPROM_writeData(EEPROM_LAST_INDEX_ADD,
            (byte*)&lLastIndex, sizeof(long));

        EEPROM_writeByte(EEPROM_MIN_ADD, tTime.bMin);
        EEPROM_writeByte(EEPROM_HOUR_ADD, tTime.bHour);

        bLogData = 0;       // azzera la variabile spia
    }
}
```

3.4 Librerie esterne

Con progetti di simili dimensioni l'utilizzo di librerie esterne diventa indispensabile; la comunicazione tramite bus I^2C^{TM} e la gestione del display LCD poi sono operazioni che vengono svolte molto spesso durante l'esecuzione del programma. Per questo, la presenza di librerie ben organizzate permette di snellire il codice e rendere molto più agevole il processo di programmazione.

3.4.1 Libreria `i2c.h`

L'insieme di funzioni presenti in questa libreria permette il dialogo tra dispositivi secondo il protocollo I^2C^{TM}, fornendo le basi per la comunicazione tra master (microcontrollore) e slave (memoria EEPROM esterna). La libreria è stata sviluppata appositamente per questo progetto seguendo le definizioni del protocollo e delle operazioni da seguire contenute nei vari datasheet [14], [15].

```
/**
 * \file i2c.h
 * \brief I2C communication lib, header file.
 *
 * \date 16/08/2011
 * \author Stefano Cillo <stefano.cillo25@gmail.com>
 * \version v0.1
 *
 */

#ifndef I2C_H_
#define I2C_H_

#define  START              0x08
#define  REPEAT_START       0x10

// Master Transmit Mode
#define  MT_SLA_ACK         0x18
#define  MT_SLA_NACK        0x20
#define  MT_DATA_ACK        0x28
#define  MT_DATA_NACK       0x30
// Master Receive Mode
#define  MR_SLA_ACK         0x40
#define  MR_SLA_NACK        0x48
#define  MR_DATA_ACK        0x50
#define  MR_DATA_NACK       0x58

#define  ARB_LOST           0x38

#define  ERROR_CODE         0xD7

#define TW_STATUS   TWSR
#define TW_CONTROL  TWCR

#define RX_ACK      NACK

unsigned char i2c_start(void);
unsigned char i2c_start_address(unsigned char);
unsigned char i2c_repeatStart(void);
unsigned char i2c_sendAddress_ACK(unsigned char);
unsigned char i2c_sendAddress_NACK(unsigned char);
unsigned char i2c_sendData_ACK(unsigned char);
unsigned char i2c_sendData_NACK(unsigned char);
unsigned char i2c_receiveData_ACK(void);
unsigned char i2c_receiveData_NACK(void);
```

```
void i2c_stop(void);
```

Listing 3.10: Contenuto del file `i2c.h`.

3.4.2 Libreria `EEPROM.h`

Come la precedente, questa libreria è stata sviluppata seguendo accuratamente il datasheet [15] e raccogliendo alcuni risultati dalla rete. Il risultato è una libreria che fornisce delle funzioni che permettono di *dialogare* direttamente con la memoria EEPROM, collegata tramite bus I^2C^{TM} con il microcontrollore.

```
/**
 * \file EEPROM.h
 * \brief EEPROM chip communication interface, header.
 *
 * \date 08/11/2011
 * \author Stefano Cillo <cillino.25@gmail.com>
 * \version v0.1
 *
 *
 */

#ifndef EEPROM_H_
#define EEPROM_H_

#define MICROCHIP   1
#define ATMEL       2

#define EEPROM_BRAND    MICROCHIP

// 1KB eeprom = 1048576 bit = 131072 byte
#define EEPROM_SIZE_B           131072UL
#define EEPROM_EXTENDED_SIZE    1

#define EEPROM_PAGESIZE 128

#define EEPROM_PAGE_NUMBER  EEPROM_SIZE_B / EEPROM_PAGESIZE

#define EEPROM_WRITE_TIME_MS        5


// Slave address
// EEPROM will be used with A1=A0=0 (GND)
#define SLA         0xa0

// Page number
#define PAGE_0      0x0

#if EEPROM_BRAND==MICROCHIP
  #define PAGE_1        8
#elif EEPROM_BRAND==ATMEL
  #define PAGE_1        1
```

```
#endif

#define  W          0x0
#define  R          0x1



/***************** EEPROM MiddleWare ****************/
// at24cXX termina il RANDOM READ con un NACK!!
#define AT24_RR_ACK_TYPE        0

// at24c usa un ACK per operazioni BYTE WRITE
#define AT24_BW_ACK_TYPE        1




/*************************************************/
/*************************************************/


uint8_t EEPROM_open (void);
uint8_t EEPROM_readByte( uint32_t address );
uint8_t EEPROM_writeByte( uint32_t address, uint8_t data);
uint8_t EEPROM_writeData( uint32_t address, uint8_t * bpData,
                                    uint8_t length);
uint8_t EEPROM_readData( uint32_t address, uint8_t * bpDest,
                                    uint8_t lenght);
uint8_t EEPROM_readPage( uint32_t pageNumber, uint8_t * dest );
uint8_t EEPROM_writePage( uint32_t pageNumber, uint8_t * src );
uint8_t EEPROM_sequentialRead( uint32_t address, uint32_t numOfBytes,
                                    uint8_t * dest);
uint8_t EEPROM_sequentialWrite( uint32_t address, uint32_t numOfBytes,
                                    uint8_t * src);
uint32_t EEPROM_erase( uint32_t sizeKbit );

#endif // EEPROM_H_
```

La scrittura di un campione di temperatura (per i campioni di umidità la cosa è identica) in memoria è realizzata dall'istruzione
`EEPROM_writeData(lLastIndex, (byte*)&fTemperature, SIZE_OF_LOG);`,
dove `lLastIndex` è una variabile di tipo `long` contenente l'indirizzo della prima cella di memoria in cui scrivere il dato[11], `fTemperature` è il puntatore al valore da scrivere in memoria, `SIZE_OF_LOG` è la dimensione del dato.

3.4.3 Libreria `lcd_fleury.h`

Questa libreria è utilizzata così com'è stata messa in rete dall'autore; la programmazione del display LCD si è rivelata particolarmente difficoltosa, mentre

[11] Si ricordi che un `float` corrisponde a 4 byte.

la seguente libreria gestisce in maniera efficiente tutte le fasi di inizializzazione dell'LCD, fornisce funzioni complete ed è stato possibile integrarla velocemente con il progetto.

```
#ifndef LCD_H
#define LCD_H
/************************************************************************
 Title  :   C include file for the HD44780U LCD library (lcd.c)
 Author:    Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
 File:      $Id: lcd.h,v 1.13.2.2 2006/01/30 19:51:33 peter Exp $
 Software:  AVR-GCC 3.3
 Hardware:  any AVR device, memory mapped mode only for AT90/8515/Mega
************************************************************************/
/**
 @defgroup pfleury_lcd LCD library
 @code #include <lcd.h> @endcode

 @brief Basic routines for interfacing a HD44780U-based LCD display

 Originally based on Volker Oth's LCD library,
 changed lcd_init(), added additional constants for lcd_command(),
 added 4-bit I/O mode, improved and optimized code.

 Library can be operated in memory mapped mode (LCD_IO_MODE=0) or in
 4-bit IO port mode (LCD_IO_MODE=1). 8-bit IO port mode not supported.

 Memory mapped mode compatible with Kanda STK200, but supports also
 generation of R/W signal through A8 address line.

 @author Peter Fleury pfleury@gmx.ch http://jump.to/fleury


*/

extern void lcd_init(uint8_t dispAttr);
extern void lcd_clear(void);
extern void lcd_home(void);
extern void lcd_gotoxy(uint8_t x, uint8_t y);
extern void lcd_putc(char c);
extern void lcd_puts(const char *s);
extern void lcd_putsXY(uint8_t x, uint8_t y, const char *s);
extern void lcd_puts_p(const char *progmem_s);
extern void lcd_command(uint8_t cmd);
extern void lcd_data(uint8_t data);

#define lcd_puts_P(__s)         lcd_puts_p(PSTR(__s))

#endif //LCD_H
```

3.5 Applicazione per il trasferimento dati su PC (SENSE-GUI)

Questa applicazion permette di trasferire i campioni memorizzati nella EEPROM al PC, in un file di tipo ***csv*** (Comma Separated Values); utilizza la porta COM per la comunicazione seriale con il microcontrollore del modulo ed è stata scritta interamente in C# in ambiente Visual Studio. La comunicazione è possibile grazie al driver integrato nella scheda Arduino che effettua il *trasporto* della comunicazione seriale su un supporto fisico diverso, in questo caso sul protocollo USB.

L'interfaccia grafica richiede che vengano impostati i parametri di connessione con il microcontrollore prima di effettuare la connessione stessa; quando quest'ultima è stabilita, è possibile inviare dei comandi al modulo tramite la pressione di appositi pulsanti presenti sul *form* dell'applicazione.

Per eseguire il trasferimento dei dati è sufficiente, una volta stabilita la connessione, cliccare sul pulsante ***Trasferimento Dati***, e successivamente su ***Crea File*** (vedi Figura 3.2): dopo la pressione del secondo pulsante verrà aperta una finestra di dialogo che permetterà di inserire il nome del file di destinazione e la sua posizione. Una volta inserite queste informazioni, verrà creato un file in formato `csv` contenente tutti i valori campionati fino a quel momento, formattati in maniera tale da poter essere visualizzati agevolmente tramite un foglio di calcolo. Grazie alla struttura del file creato è possibile creare un grafico dei valori, per meglio visualizzare gli andamenti dei parametri campionati.

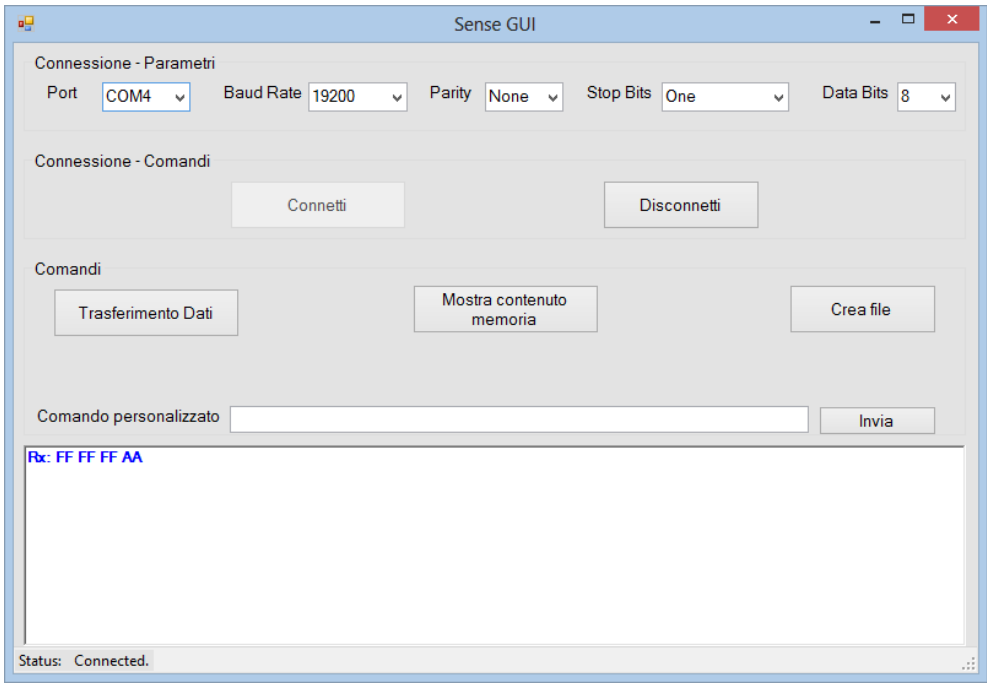


Figura 3.2: Interfaccia SENSE-GUI a connessione avvenuta.

Capitolo 4

Risultati e Conclusioni

La scrittura dell'intero programma ha richiesto una notevole quantità di tempo, per svariate ragioni: prima fra tutte, l'affacciarsi a un nuovo mondo, quello dei microcontrollori. Lo stile di programmazione è molto particolare, e non è stato facile capire come meglio gestire l'interazione tra l'esecuzione normale del programma e le varie interrupt. Alcune volte anche la fase di debug sembrava inutile, perché a causa di eventi di interrupt non controllati il comportamento del micro sembrava assolutamente casuale. Per questo, l'utilizzo di software di prototipazione che permettessero di conoscere il valore delle variabili interne in qualsiasi momento, bloccare il micro ed eseguire un'istruzione alla volta si è rivelata di importanza fondamentale.

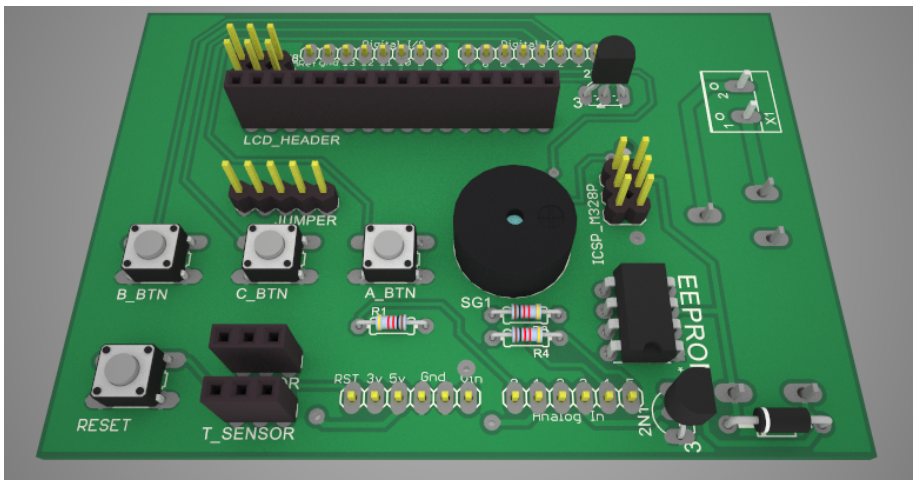
Nonostante i numerosi problemi, una volta entrati nell'ottica corretta e avendo un'adeguata visione d'insieme si riesce a cogliere anche il più piccolo dettaglio, e si riesce a capire fino a fondo l'esecuzione complessiva del sistema. Il mondo dei microcontrollori è l'interfaccia tra il mondo fisico e la realtà software, una volta capite le sue potenzialità si riescono a trovare grandi soddisfazioni.

4.1 Realizzazione sperimentale

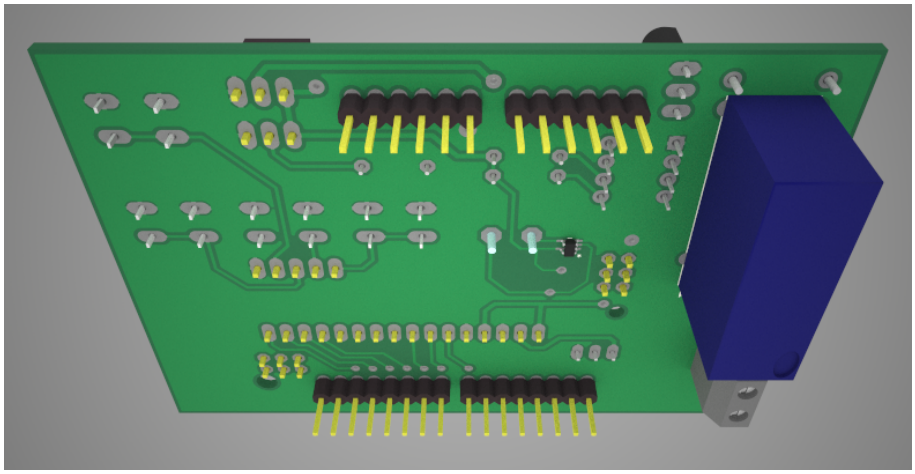
Lo shield è stato realizzato fisicamente su una basetta ramata a doppia faccia, utilizzando la tecnica della fotoincisione con bromografo a LED ultra-violetti. Lo strumento è stato realizzato in casa, all'interno di una scatola di legno secondo un progetto personale. I risultati di fabbricazione sono davvero ottimi, si sono realizzate con successo piste della larghezza di circa 12 mil (cioè 0.3 mm) sebbene l'incisione non sia veloce come nei normali bromografi a neon UV (7-8 minuti dei LED contro i 2 minuti del neon). La cosa che tutt'ora comporta il maggior dispendio di tempo nella fabbricazione di PCB *do-it-yourself (o DIY)* è la foratura della scheda e l'applicazione dei *via*, fori passanti che collegano piste del lato superiore con quelle del lato opposto.

Lo schema elettrico dello shield (rappresentato in Figura 4.2) è sostanzialmente la realizzazione fisica del render di Figura 4.1. Sono presenti tutti gli

elementi aggiuntivi necessari per l'implementazione del progetto, e viene sfruttata la modularità di Arduino: infatti gli header `ANALOG_IN`, `POWER`, `DIGITAL_0` e `DIGITAL_1` si collegano direttamente alla piattaforma sottostante (Arduino appunto), e derivano le sue connessioni ai pin del microcontrollore per collegare i componenti presenti. Per una maggiore chiarezza viene riportato anche lo schema elettrico di Arduino, rappresentato in Figura 4.3



(a)



(b)

Figura 4.1: Render dello shield. (a): top, (b): bottom.
Immagine creata utilizzando SketchUp e il plugin Maxwell.

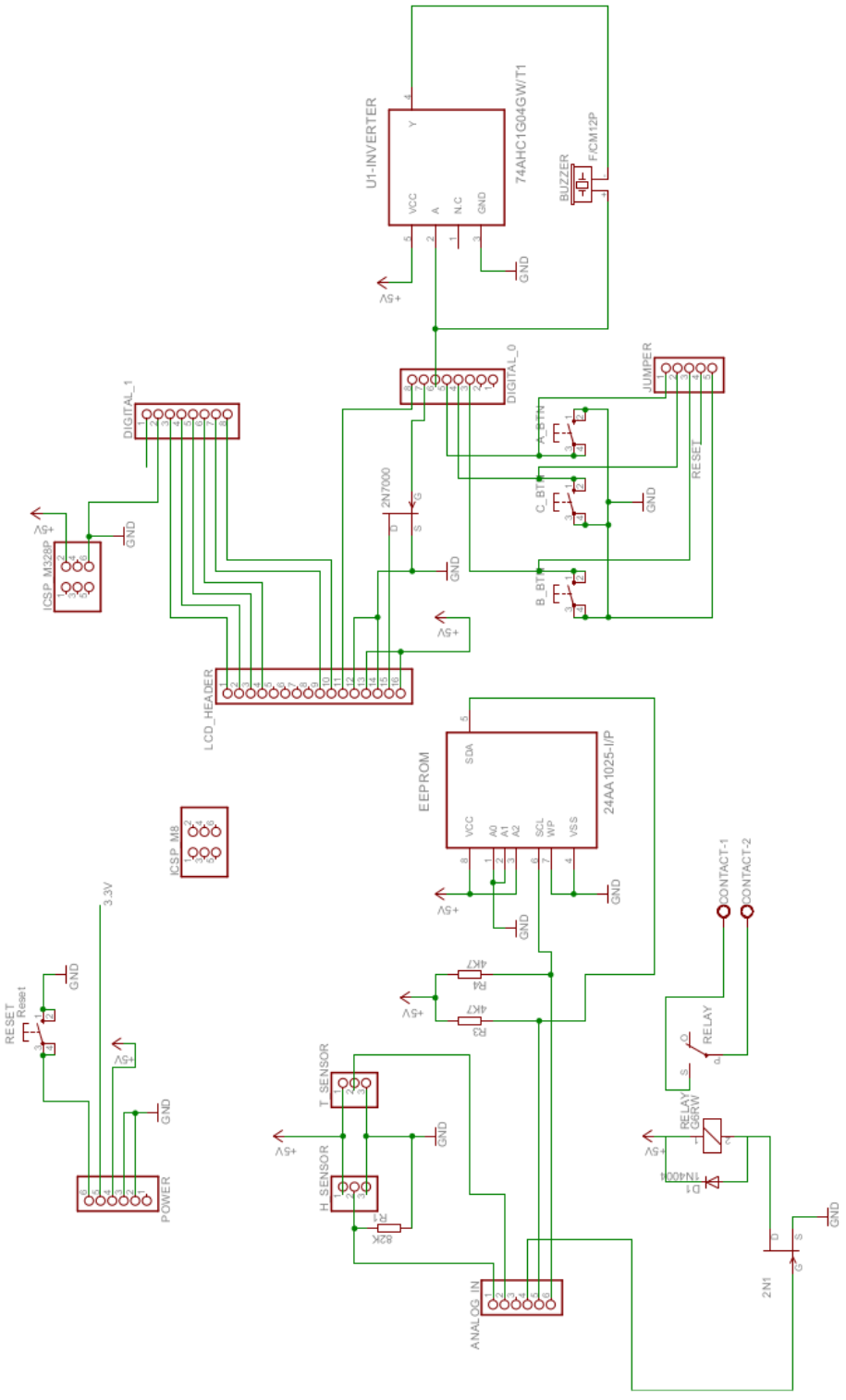


Figura 4.2: Schema circuitale del modulo datalogger.

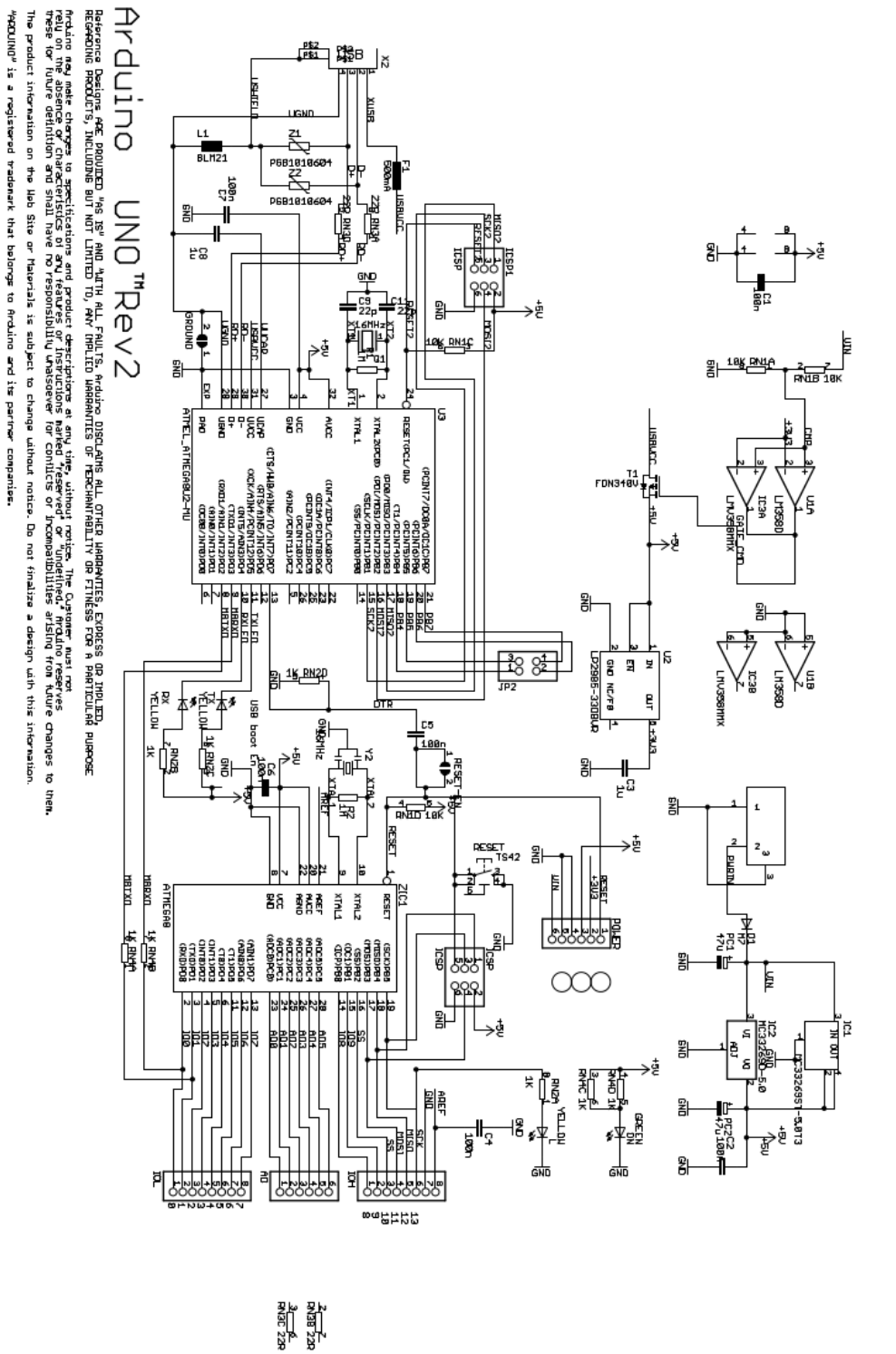


Figura 4.3: Schema circuitale della piattaforma Arduino.

4.2 Conclusioni e Sviluppi futuri

Il modulo datalogger è progettato per acquisire i valori di umidità e temperatura dell'ambiente in cui è immerso. Si serve di sensori per la trasduzione dei parametri in segnali elettrici, effettua la conversione analogico/digitale tramite una periferica integrata nel microcontrollore Atmel® ATmega328P, cuore della scheda Arduino. É in grado di campionare i dati e di salvarli in una memoria EEPROM dedicata, e renderli disponibili in un foglio di calcolo per una successiva rielaborazione, grazie alla collaborazione di una interfaccia utente presente nel modulo e visualizzabile tramite un display LCD 16x2, e un'applicazione Windows sviluppata in C# .

Il dispositivo consiste in uno *shield* sviluppato per sfruttare la modularità con Arduino, nel quale è contenuto tutto l'hardware necessario per la realizzazione del progetto: un display LCD 16x2 su cui è possibile "navigare" tramite tre pulsanti, sensori per umidità e temperatura, una memoria EEPROM su cui vengono salvati i dati campionati, un relè collegato ad un deumidificatore ed un cicalino che segnala il superamento del valore limite di umidità.

Dall'interfaccia utente è possibile impostare due valori di soglia per l'umidità: se viene superato il primo, il modulo provvede a chiudere il relè che accende un deumidificatore; superata la seconda soglia, viene fatto squillare un cicalino che avverte della situazione di allarme i tecnici responsabili della manutenzione della cella.

L'UI (*User Interface*) permette inoltre di impostare la data e l'ora, in modo che i valori salvati in memoria abbiano il corretto riferimento temporale; i campioni possono essere letti con un foglio elettronico una volta trasferiti su PC tramite l'apposita applicazione su PC.

Le possibilità di ampliamento del progetto sono potenzialmente infinite; scegliendo tra le tante quelle a maggior riscontro pratico, si potrebbe pensare di montare un modulo GSM sullo shield, in modo tale che qualora il programma segnalasse una situazione d'allarme dei responsabili verrebbero prontamente avvertiti.

Un altro grosso miglioramento sarebbe di riuscire ad interfacciarsi alla cella a combustibile tramite l'apposita interfaccia CAN: tramite questo protocollo sarebbe possibile inviare e ricevere messaggi di stato alla cella, nonché ricevere tempestive segnalazioni d'allarme.

Ulteriori sviluppi potrebbero prevedere il miglioramento del software, migliorando la fase di acquisizione dati, ad esempio filtrando i valori campionati con una serie di filtri digitali per "ammorbidire" le curve dei parametri rilevati.

Bibliografia

[1] Rino A. Michelin, Andrea Munari, *Fondamenti di Chimica*, II edizione, CEDAM, 2011.

[2] G. Hoogers, *Fuel Cell Technology Handbook*, Boca Raton, FL, CRC Press, 2003.

[3] Je Seung Lee, Nguyen Dinh Quan, Jun Min Hwang et al., *Polymer Electrolyte Membrane for Fuel Cells*, Journal of Industrial and Engineering Chemical Research, vol. 12, no. 2, 2006.

[4] James Larminie and Andrew Dicks, *Fuel Cell systems explained*, John Wiley & Sons, England, 2000.

[5] Serenergy, *Serenus 166/390 Air Cooled User Manual*.

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

[7] Atmel® , *ATmega328P datasheet*, 2009.

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

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

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

[11] Atmel® Site, URL: *http://www.atmel.com*

[12] AVRfreaks Site, URL: *http://www.avrfreaks.net*

[13] Atmel® , *Atmel AVR126: ADC of megaAVR in Single Ended Mode*, 2011, URL: *http://www.atmel.com/Images/doc8444.pdf*

[14] Atmel® , *ATmega48PA/88PA/168PA/328P microcontroller datasheet*, URL: *http://www.atmel.com/Images/doc8161.pdf*

[15] Microchip, *1024K I^2C^{TM} CMOS Serial EEPROM*, URL: *http://ww1.microchip.com/downloads/en/DeviceDoc/21941B.pdf*