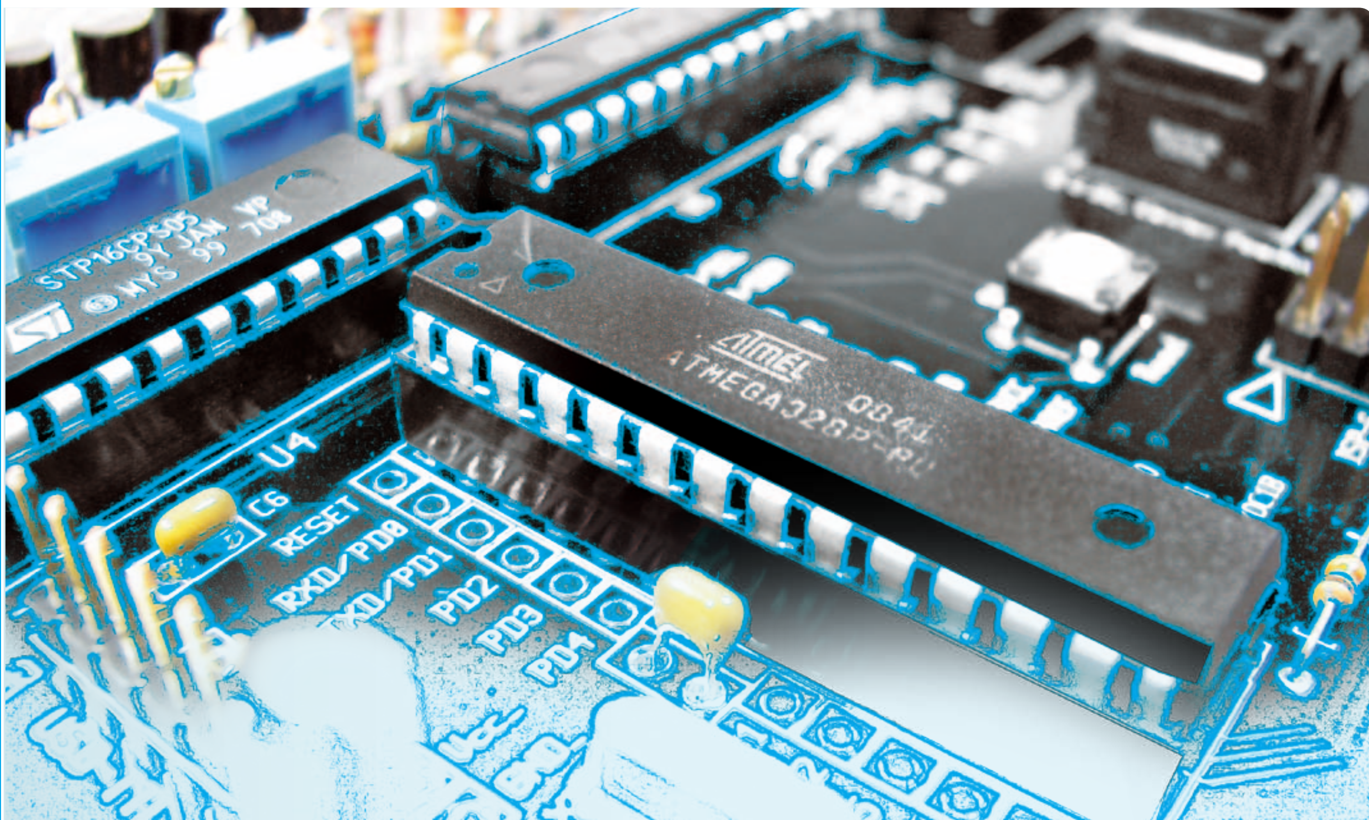


ATMEL OPEN SOURCE

di OSVALDO SANDOLETTI

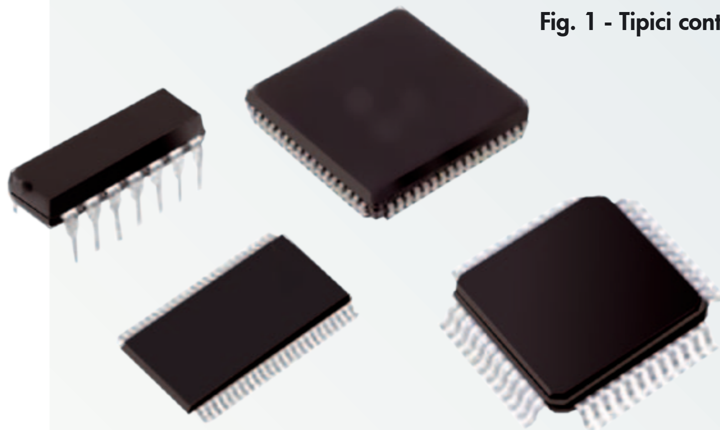
Impariamo ad utilizzare i microcontrollori Atmel e il loro ambiente di sviluppo, partendo da nozioni propedeutiche all'apprendimento della programmazione e dell'uso dell'ATmega16. Prima puntata.

Iniziamo la stesura di una serie di articoli il cui scopo è iniziare i lettori interessati alla conoscenza e programmazione dei componenti elettronici più interessanti e versatili -i microcontrollori- che grazie alle loro caratteristiche e soprattutto al costo in continuo calo, stanno diventando l'elemento principe di un numero sempre maggiore di applicazioni. Premettiamo che tale corso non si focalizzerà su complicati aspetti teorici legati all'architettura dei dispositivi o ad avanzate tecniche di programmazione, di cui esiste abbondanza di documentazione sia cartacea che on-line: l'obiettivo principale è mettervi in condizione, entro le prime cinque puntate, di poter lavorare praticamente con i microcontrollori per poterne apprezzare le potenzialità e verificare se può "valere la pena" di approfondire i numerosi argomenti che comunque tratteremo. Passiamo pertanto ad una breve descrizione necessaria a "rompere il ghiaccio".

CHE COS'È UN MICROCONTROLLORE

Con il termine microcontrollore (anche detto *single-chip microcomputer* e abbreviato in **µC** oppure, informalmente, "micro") si intende

Fig. 1 - Tipici contenitori dei microcontrollori.



un circuito integrato (che esternamente si presenta in uno dei contenitori o *packages* -in inglese- mostrati in **Fig. 1**) che ha la peculiarità di poter essere predisposto, dopo essere stato *istruito* opportunamente, ad eseguire molteplici funzioni.
Per rendere meglio l'idea, possiamo paragonare il microcontrollore alla cucina di un ristorante: in essa è presente un cuoco che già possiede le capacità e gli strumenti per creare diversi tipi di pietanze, ma che non può mettersi al lavoro fin quando non gli viene presentata la ricetta dei piatti scelti dai clienti. Allo stesso modo, il microcontrollore necessita di ricevere opportune istruzioni che gli comandino passo per passo le operazioni da svolgere; tali istruzioni, ovviamente, non sono le fasi di esecuzione della ricetta, ma sequenze di numeri binari e sono contenute non in un libro, ma in un apposito dispositivo detto *memoria di programma*, in quanto l'insieme delle istruzioni è identificato -appunto- con il nome di *programma* (o *firmware*). Da qui deriva la versatilità del microcontrollore: andando a cambiare il contenuto della memoria, possiamo fare eseguire allo stesso dispositivo "ricette" diverse senza cambiare o modificare i componenti della scheda (il cosiddetto *hardware*), microcontrollore compreso.
A questo punto dovrebbe esservi abbastanza chiaro che se si vuole sostituire il microcontrollore guasto di una scheda non è sufficiente procurarsene uno uguale, ma bisogna anche scrivervi nella memoria lo stesso programma di quello "vecchio".
Passiamo ora ad analizzare meglio l'operato del nostro μC, dicendo che per ogni istruzione depositata in memoria esso deve compiere una serie di azioni per eseguire materialmente ciò che l'istruzione stessa chiede.
Un esempio chiarirà meglio il concetto: supponiamo che l'istruzione ricevuta sia relativa ad un'operazione di somma; allora il microcontrollore deve leggere i due operandi da un ingresso, elaborare i numeri per produrre la somma e mandare il risultato ad un'uscita per renderlo disponibile all'utilizzatore. In seguito, verrà letta la prossima istruzione (che può essere relativa ad una qualsiasi altra operazione) ed il ciclo si ripete.
Questa sequenza di operazioni (lettura istruzione, ingresso dati, elaborazione, uscita risultato) è simile quasi per tutti i tipi di istruzione

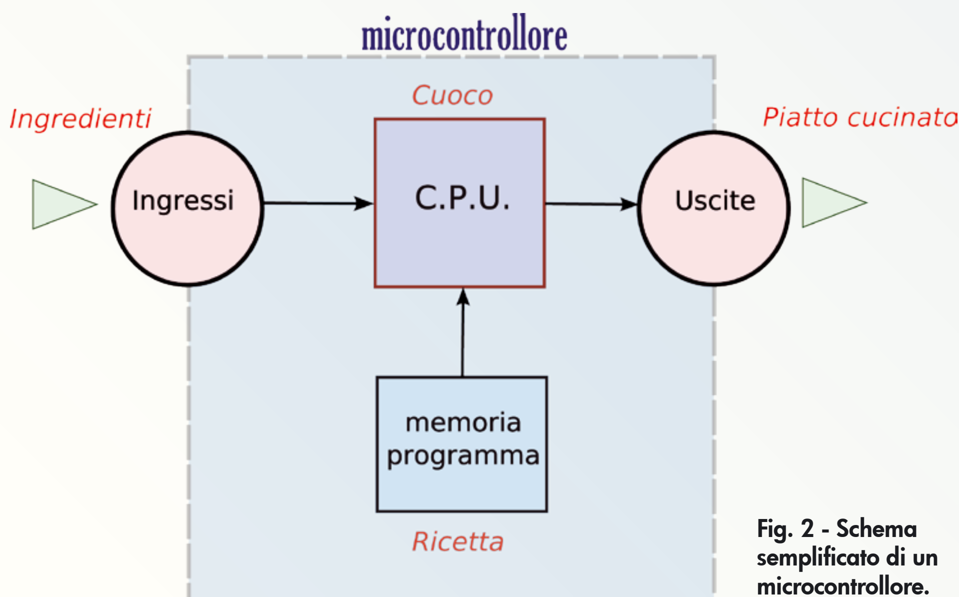
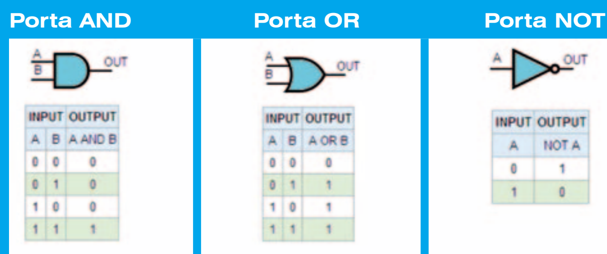


Fig. 2 - Schema semplificato di un microcontrollore.

I numeri **binari**

C'era una volta l'elettronica analogica... Un tempo "lontano" (fino a quarant'anni fa) gli unici circuiti in grado di eseguire calcoli logici e aritmetici erano basati su componenti discreti (transistori) o integrati (amplificatori operazionali) che lavoravano direttamente sui livelli di tensione: per fare un esempio, per sommare 3 + 5 al circuito sommatore bisognava applicare sui due ingressi le tensioni di 3 V e 5 V e si andava a misurare l'uscita per conoscere il risultato (8 V). Successivamente sono state introdotte le *porte logiche*, cioè circuiti in grado di riconoscere ai propri ingressi unicamente due livelli di tensione per discriminare altrettanti *livelli logici: zero* e *uno*; queste porte erano in grado di eseguire solo operazioni elementari in logica binaria (AND, OR, NOT, ecc.). Nella Tabella 2 sono riportate le porte logiche (e le relative "tabelline della verità") corrispondenti alle tre operazioni logiche fondamentali (AND, OR e NOT). Tutte le altre si possono derivare da esse.

Porta AND **Porta OR** **Porta NOT**



Queste operazioni sono dette di logica *binaria*: leggono ingressi con soli due stati possibili e il risultato può avere ancora due soli stati.
Combinando in modo opportuno queste tre funzioni logiche fondamentali, si è in grado di eseguire virtualmente qualsiasi operazione aritmetica e logica: in questo senso si possono paragonare le porte logiche ai mattoni, che pur essendo "tutti uguali" permettono di costruire edifici di qualsiasi forma.
Inoltre il fatto di lavorare con circuiti che accettano solo livelli logici binari (i *bit*) non è limitante: se con un bit rappresentiamo solo due valori, con due bit abbiamo la possibilità di rappresentarne quattro: 00, 01, 10, 11. Con tre bit arriviamo ad 8 valori differenti e così via. La regola generale è che il numero di combinazioni rappresentabili vale **N = 2^B** dove B è il numero di bit. Quindi l'aumento del numero di bit porta ad avere un aumento esponenziale del numero di combinazioni rappresentabili; nei circuiti a microprocessore si lavora soprattutto con numeri formati da 8 bit (256 combinazioni e quindi rappresentano, in decimale, i numeri da 0 a 255), ma si arriva anche a 32 bit (oltre

Decimale (base 10)	Esadecimale (base 16)	Binario (base 2)	Decimale (base 10)	Esadecimale (base 16)	Binario (base 2)
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

Tabella 1

4 miliardi di combinazioni). A seconda del numero di bit che li compongono, si sono stabiliti dei nomi per contraddistinguere i gruppi: *nibble* (4 bit), *byte* (8 bit), *word* (16 bit) e *double word* (32 bit).
Esistono regole precise per fare le conversioni tra i numeri binari (formati dai bit, 0 e 1) e i numeri decimali (le cui cifre vanno da 0 a 9, ai quali noi siamo abituati a pensare), che si possono rapidamente trovare in Internet. Purtroppo, però, la corrispondenza binario-decimale non è intuitiva: difficile capire "ad occhio" che il numero binario 1001101 corrisponde al numero decimale 77. Pertanto per esprimere rapidamente un numero binario è stata introdotta la numerazione esadecimale, che assegna ad ogni combinazione che può assumere un gruppo di 4 bit (il nibble) un simbolo diverso: poiché nel nibble i valori variano da 0000 a 1111 (16 combinazioni), utilizziamo, per rappresentarli, i numeri da 0 a 9 e le lettere da A a F come mostrato nella Tabella 1.
Quindi possiamo suddividere un numero binario arbitrariamente grande in gruppi da 4 bit e sostituire ad essi i simboli in accordo con la tabella. Ad esempio, trasformiamo il numero a 16 bit (word) '1101001000110100' in un numero esadecimale:

1101001000110100 ▯ 1101 | 0010 | 0011 | 0100 ▯ D234

Questa notazione è largamente utilizzata per esprimere numeri binari nella stesura dei programmi, in quanto è più semplice scrivere D234 piuttosto di 1101001000110100, e genera meno confusione ed errori anche di semplice battitura (è facile sbagliarsi e scrivere 1101010000110100 invece di 1101001000110100, mentre è più difficile scrivere D434 invece di D234). Un'ultima osservazione: poiché i simboli della notazione esadecimale comprendono anche i numeri, per non confondere un numero decimale con uno esadecimale (sono diversi) si possono far seguire questi ultimi da una 'H', oppure (in stile del linguaggio C) precederli con '0x'. Ad esempio 16 decimale vale 10 esadecimale e lo indichiamo come 10h o 10_H o 0x10; al numero decimale invece non si fa in genere seguire alcun suffisso e raramente si usa la lettera 'D' ($16_D = 10_H$).

che esso può eseguire. Lo schema semplificato di un microcontrollore può pertanto essere rappresentato come in **Fig. 2**, dove sono state anche riportate, in rosso, le voci dell'analogia culinaria.
Il micro è essenzialmente composto da blocchi di ingresso e uscita e da un'unità centrale di elaborazione (*CPU*, da *Central Process Unit*), detta anche *processore* oppure *core*, in grado di interpretare ed eseguire le istruzioni che va a leggere da un'apposita memoria di programma, come vedremo meglio nel seguito.

LA CPU

È l'unità di elaborazione del microcontrollore: essa preleva le istruzioni dalla memoria di programma, le decodifica (per capire qual è l'azione da compiere) e le esegue, prelevando

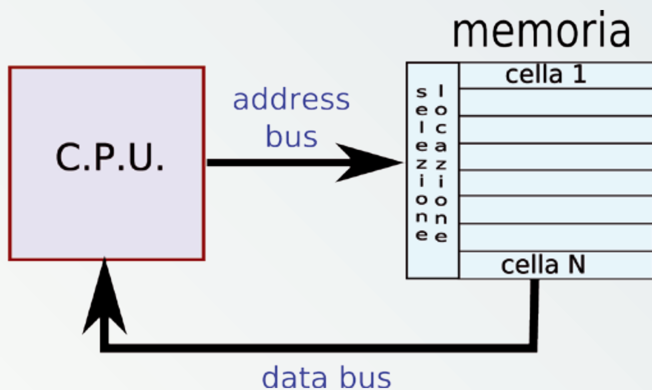


Fig. 3 - Collegamento tra CPU e memoria di programma.

(eventualmente) gli operandi dal blocco di ingresso e trasmettendo, infine, i risultati al blocco di uscita.
A questo punto chiariamo meglio in cosa consistono i dati (operandi ed istruzioni) con cui ha a che fare la CPU; questa è una **macchina digitale** in grado di leggere, interpretare ed elaborare sequenze di **numeri binari** (per chi avesse qualche perplessità sul significato del termine "binario" consigliamo di leggere il riquadro I **numeri binari**, in queste pagine) pertanto le uniche due lettere del suo alfabeto sono i valori "0" e "1" che possono assumere i bit costituenti le parole (siano esse *byte*, *word*, *double word*, …) dei dati e delle istruzioni.
Normalmente le CPU lavorano con parole il cui numero di bit è una potenza di due, da 4 in su (quindi 4, 8, 16, 32 bit), anche se esistono eccezioni (alcuni µC della famiglia *PIC* di Microchip hanno istruzioni nel formato di 12 e 14 bit, ma i dati sono sempre "larghi" 8 bit). La CPU della stragrande maggioranza dei microcontrollori in commercio (tra cui il dispositivo ATMEGA16 di Atmel che utilizzeremo in questo corso) lavora con parole da 8 bit, sia per le istruzioni che per i dati.

LE ISTRUZIONI E LA MEMORIA DI PROGRAMMA

Come già accennato, la *memoria di programma* (esiste anche la memoria dati, ovvero la RAM, che però analizzeremo più avanti) è il contenitore in cui depositare tutte le istruzioni che compongono il programma da far eseguire al microcontrollore.
Possiamo paragonare questa memoria ad un mobile con tanti cassetti, identificati con una numerazione crescente, contenenti ciascuno un numero binario corrispondente ad un'istruzione del programma. Durante il funzionamento, la CPU del microcontrollore va ad aprire questi cassetti (uno alla volta), ne estrae il numero binario che identifica una particolare istruzione, la esegue e quindi passa ad aprire il cassetto seguente per la lettura del numero binario (e quindi dell'istruzione) successivo. Come nelle fasi di preparazione di una ricetta di cucina, anche durante la normale esecuzione del programma le istruzioni devono essere eseguite in una precisa e rigorosa successione: pertanto la CPU si cura di aprire i cassetti in sequenza, ovvero dopo il cassetto 1 passa al 2, poi al 3 e così via.
Sperando che il concetto sia stato chiarito, formalizziamo alcune definizioni: parlando di memorie elettroniche, i cassetti si chiamano *celle* o *locazioni* e la loro selezione si chiama *indirizzamento*.
Nella **Fig. 3** è schematizzata la connessione tra CPU e memoria di programma.
L'address bus (bus degli indirizzi) è formato da un certo insieme di fili che portano alla memoria il numero binario specificante la cella da selezionare: ad esempio mettendo 0101 (5 in decimale) si seleziona la quinta cella. Il valore contenuto nella cella viene quindi "riversato" sulle linee del *data bus* (bus dati).
In definitiva, in fase di lettura dell'istruzione la CPU imposta sui fili dell'*address bus* l'indirizzo della cella interessata e va poi a leggerne il valore sul data bus.
Una delle principali caratteristiche della memoria programma è la non volatilità, ovvero la capacità di mantenere i dati anche in caso di mancanza di alimentazione del dispositivo, per cui non è necessario riscriverla (programmarla) ad ogni accensione del sistema. Inoltre tale memoria è di sola lettura (*Read Only Memory*, o ROM) da parte della CPU, in quanto deve solo leggervi le istruzioni; può invece essere eventualmente riscritta (se del tipo EEPROM o Flash) dal programmatore (quindi da noi, tra poche lezioni…) per potervi caricare un nuovo programma.

IL SEGNALE DI RESET

In ogni microcontrollore esiste un ingresso di *reset* (che significa azzeramento), il cui scopo è quello di riportare la CPU ed i registri interni del micro nelle condizioni iniziali, indipendentemente dallo stato in cui si trovavano nell'istante di attivazione del reset stesso. Per *condizione iniziale* della CPU si intende, tra le

altre cose, che essa deve ripartire ad eseguire il codice dalla prima istruzione del programma, mentre i registri (dei quali spiegheremo più avanti le loro funzioni) vengono riportati ai valori definiti come "Initial value" nel datasheet del microcontrollore, quindi in uno stato predefinito.
È dunque importante che il reset sia attivato almeno all'accensione del sistema, in modo da fare partire correttamente l'esecuzione del programma dalla prima istruzione; in caso contrario, sarebbe indefinito il punto in cui la CPU inizierebbe ad eseguirlo! Come tanti altri dispositivi, l'ATMEGA16 dispone di un meccanismo detto "brown-out" (talvolta è chiamato anche *smart reset*) che rileva quando la tensione d'alimentazione del micro è inferiore al livello di funzionamento normale (ad esempio perché sta ancora salendo dallo zero dopo l'accensione del sistema): in tal caso tiene attivato il reset fin quando il valore non è arrivato al livello stabilito, allorché lo rilascia e la CPU parte ed esegue regolarmente il programma dall'inizio.
Nel nostro micro il segnale di reset è comunque accessibile da un piedino esterno, in modo da dare all'utente la possibilità di attivarlo anche manualmente; potrebbe, ad esempio, essere utile per fare ripartire un programma che si è bloccato.
Nulla di strano: anche tutti i PC desktop hanno il famoso tastino di reset, che ben conoscono gli utilizzatori di alcuni sistemi operativi...

LE PERIFERICHE DI INGRESSO E USCITA

I blocchi di ingresso e uscita (detti *periferiche*) sono le porte che collegano il microcontrollore al mondo esterno: da esse vengono letti i dati e trasmessi i risultati di un'operazione. Le periferiche più semplici (dette porte parallele o *PIO, Parallel Input-Output*) collegano direttamente i piedini fisici (pin) del dispositivo ai bus (dati e indirizzi) della CPU, che può dunque leggerne lo stato (interpretato come bit: assenza di tensione -> legge 0, presenza di tensione positiva -> legge 1), oppure impostarlo (a 0 oppure a 1). Attraverso apposite istruzioni si può configurare un certo piedino per comportarsi arbitrariamente come ingresso o come uscita; da qui il termine di Input/Output (I/O pin).
Questi piedini vengono raggruppati per

Microprocessori e microcontrollori

La rivoluzione digitale che ha radicalmente cambiato in questi anni il nostro modo di vivere e comunicare inizia ufficialmente con la nascita del microprocessore, nel lontano (un'*era* fa, elettronicamente parlando) 1971. Con esso si apre la strada all'elettronica *programmata*, ovvero ai circuiti in grado di leggere programmi "pre-caricati" per eseguire calcoli anche notevolmente complessi e soprattutto in maniera flessibile, in quanto è il programma a decidere l'operazione da eseguire in ogni momento. Non ci volle molto ad intuire le potenzialità di questo componente, sul quale si incentrarono le risorse di diverse Case produttrici di semiconduttori (ed altre nacquero appositamente), ed iniziò subito la corsa al miglioramento continuo e all'abbattimento costi.
Nel corso del tempo, questo sviluppo si divise in due strade: da una parte la ricerca di un continuo aumento prestazionale (che seguono i microprocessori destinati per esempio ai Personal Computer), e dall'altra una riduzione dei componenti necessari per realizzare un sistema completo, e quindi l'ottimizzazione dei costi e dello spazio.
Un sistema richiede infatti, oltre al microprocessore stesso, una memoria di programma ed una memoria per i dati temporanei, oltre ad un insieme (variabile a seconda dell'applicazione) di periferiche di ingresso e uscita (I/O). Inoltre bisogna tenere in considerazione che questi componenti vanno interconnessi tra di loro con un certo numero di linee (che formano il *bus di sistema*), con la conseguente necessità di adottare circuiti stampati di ragguardevole dimensione e multistrato. Il miglioramento delle tecniche microelettroniche nello sviluppo dei chip di silicio ha però consentito di inglobare all'interno di un *unico* dispositivo, oltre al microprocessore, anche le memorie ed alcuni dispositivi di I/O, dando origine al microcontrollore: un singolo circuito integrato contenente al suo interno un sistema a microprocessore completo. Esso, per funzionare richiede solamente di essere alimentato e naturalmente di essere programmato (o meglio bisogna programmare la sua memoria di programma interna, alla quale si può accedere con opportuni circuiti e segnali "dedicati").
A questo punto, la strada verso la miniaturizzazione e la diffusione di massa è stata spianata: oggigiorno esistono numerose Case produttrici di microcontrollori (a differenza dei microprocessori, i quali sono oramai in un regime di oligopolio che guida il mercato degli elaboratori, tipo i PC) ciascuna delle quali propone una o più "famiglie" di dispositivi (come i *PIC* di Microchip e gli *ATMEGA* di Atmel) che coprono potenzialmente ogni esigenza di prezzo e prestazioni.
Di fatto, oggi possiamo facilmente trovare un microcontrollore dentro ad una sveglia elettronica, come pure nei bigliettini di auguri musicali o nei termostati del riscaldamento; l'adozione di questi componenti richiede un costo minore di quello necessario per realizzare i circuiti corrispondenti con transistori o integrati dedicati (industrialmente, un piccolo microcontrollore "da termostato" costa poche decine di centesimi, ma anche comprandolo al dettaglio non si superano i 3 €).

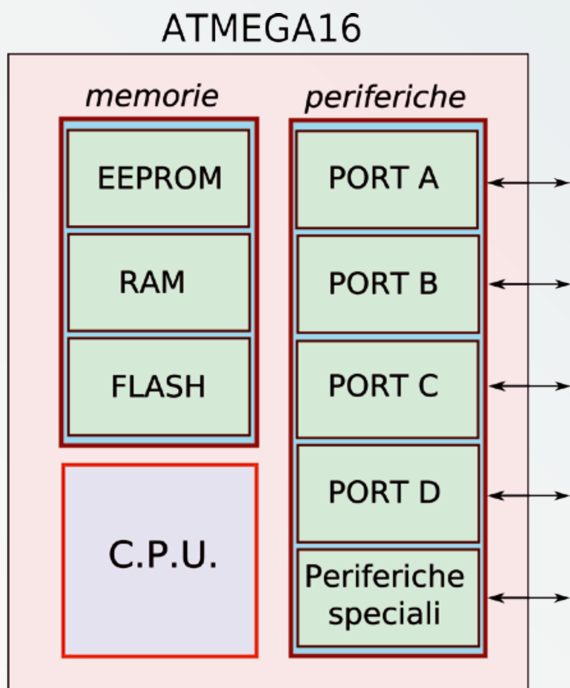


Fig. 4 - Schema a blocchi dell' Atmega16.

formare i cosiddetti I/O PORT (raramente tradotto in italiano in *PORTI*), in modo che il processore possa accedere contemporaneamente a tutti i piedini in esso contenuti (in modo, appunto, parallelo). Ad esempio, una CPU che lavora con parole da 8 bit (un byte) può leggere o scrivere in un "colpo solo" un Port formato da otto I/O pin. Avremo comunque modo di chiarire meglio questo concetto con gli esempi pratici delle prossime puntate. Come per la memoria, anche le periferiche sono indirizzate dalla CPU attraverso un numero binario sull'*address bus*, allo scopo di selezionare esattamente di volta in volta con quale si vuole operare.
Esistono anche periferiche più complesse, come la USART (Universal Synchronous Asynchronous Receiver Transmitter) o la I²C (Inter Integrated Circuit communication), per l'implementazione di protocolli seriali (i byte in arrivo dalla CPU vengono trasmessi all'esterno un bit per volta utilizzando un unico filo). Alcune periferiche possono anche non avere un collegamento diretto con i piedini esterni del microcontrollore (come l'oscillatore che fornisce le temporizzazioni alla CPU).

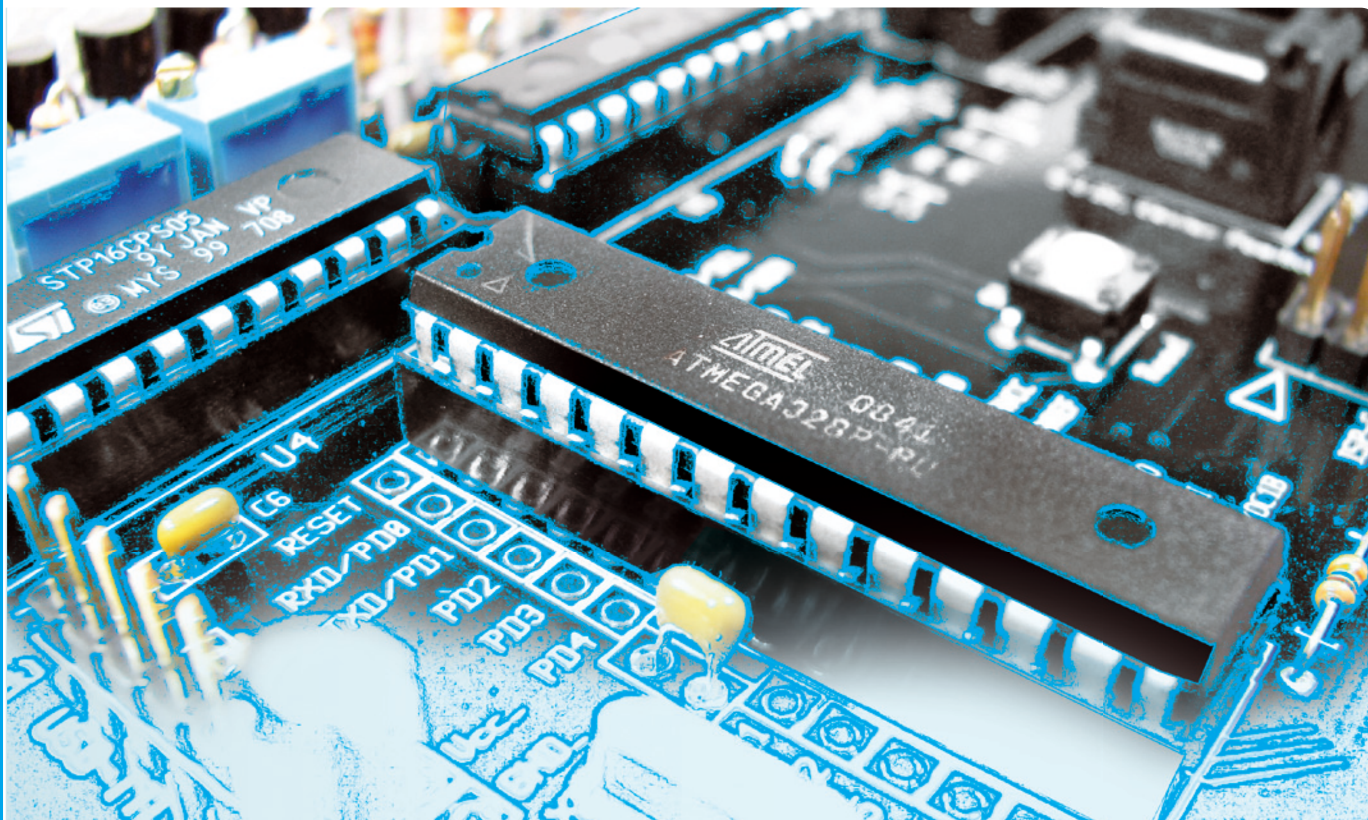
IL MICROCONTROLLORE ATMEGA16

Dopo una sommaria panoramica sui microcontrollori, possiamo focalizzare la nostra attenzione sul dispositivo prescelto per lo sviluppo di questo corso: il chip **ATmega16** della Atmel. La sua scelta è stata dovuta principalmente ai seguenti fattori:

- fa parte di una "famiglia" di microcontrollori molto estesa e diffusa (la famiglia ATMEGA, della quale fanno parte anche i dispositivi utilizzati, tra l'altro, sulle schede *Arduino*), i cui membri si distinguono tra loro fondamentalmente per la diversa capacità di memoria, la presenza o meno di particolari periferiche ed il numero di piedini di I/O disponibili;
- ampia memoria programma (di tipo Flash e quindi riscrivibile) e numerose periferiche interne;
- ottima disponibilità di programmi e schede applicative in Internet;
- compilatore C e ambiente di sviluppo disponibili gratuitamente;
- sistemi di programmazione e *debug* a basso costo;
- disponibile in contenitore dual-in-line per facilità di montaggio e sostituzione anche sui prototipi.

Detto questo, proponiamo uno schema a blocchi semplificato ed una breve descrizione delle sue componenti; per approfondimenti è consigliabile scaricarne il data-sheet completo dal sito Atmel (*www.atmel.com/dyn/resources/prod_documents/doc2466.pdf*). In ogni caso, il funzionamento e l'utilizzo delle periferiche sarà descritto nel dettaglio durante le future applicazioni teoriche e pratiche.

CPU: è il motore del sistema in quanto legge ed esegue le istruzioni; essa è chiamata da Atmel "AVR-CPU" in quanto utilizzata nei microcontrollori che l'Atmel ha battezzato **AVR** (di cui fa parte la famiglia ATmega).
Flash: è la memoria di programma, può contenere fino a 16.384 Bytes, ed è cancellabile e riscrivibile (in fase di programmazione) fino a 10.000 volte.
RAM: è una *memoria dati*, ovvero un "serbatoio" che la CPU utilizza per riporre temporaneamente variabili e risultati delle operazioni. Pertanto, a differenza della memoria di programma è scrivibile dalla CPU durante l'esecuzione del programma stesso. Un'altra



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

Dopo avervi introdotti nel mondo dei microcontrollori, affrontiamo insieme le problematiche riguardanti la programmazione di questi utilissimi componenti. Seconda puntata.

Il microcontrollore è una *macchina digitale* estremamente flessibile in quanto programmabile: lo abbiamo spiegato nella prima puntata di questo corso, evidenziando come sia in grado di eseguire operazioni diverse, stabilite dal progettista, in base alle funzionalità che deve realizzare nel sistema che lo ospita. Questa flessibilità si paga col fatto che il componente dev'essere quindi preventivamente istruito (o meglio *programmato*) sui compiti che dovrà eseguire quando accenderemo il sistema che lo ospita; infatti quando ne acquistiamo uno nuovo, esso non contiene alcun programma (in questo stato viene definito "vergine") e quindi non può eseguire alcun compito. Insomma, no fa nulla.
Dato che il programma è una componente essenziale per il funzionamento dei microcontrollori, passiamo a vedere, partendo dalle funzionalità richieste, come si stila un programma.

FONDAMENTI DI PROGRAMMAZIONE

Ogni tipo di microcontrollore può riconoscere ed eseguire un certo insieme (*set*) di istruzioni elementari: il "nostro" ATmega16 ne preve-

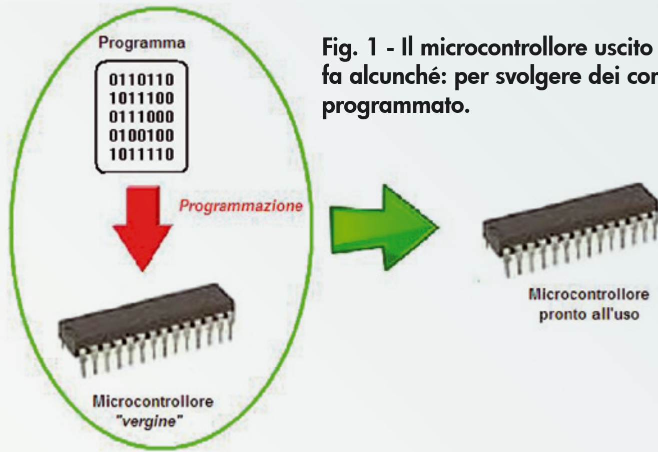


Fig. 1 - Il microcontrollore uscito di fabbrica non fa alcunché: per svolgere dei compiti dev'essere programmato.

de ben 131, mentre alcuni dispositivi della famiglia PIC16 di Microchip ne hanno solo 35 (si parla di micro basati su RISC, cioè a set di istruzioni ridotto). Va comunque detto che un processore che accetti meno istruzioni di un altro può eseguire le stesse operazioni, anche se dovrà farlo con più passaggi.
Per spiegare ciò supponiamo di avere due sistemi a microcontrollore, uno dei quali è equipaggiato con l'ATmega16 e l'altro con un microcontrollore della famiglia PIC16, e di volere far eseguire loro l'operazione aritmetica 7x4: purtroppo, a differenza dell'ATmega16, il PIC16 non prevede l'istruzione di moltiplicazione, ma solo quella di somma, tuttavia questo non è un problema, in quanto moltiplicare un numero per N significa fare N somme del numero, ovvero il risultato di 7x4 può essere ottenuto anche come 7+7+7+7. La differenza a livello di realizzazione sul microcontrollore è che se utilizziamo l'ATmega possiamo usare una sola istruzione (la moltiplicazione), mentre con il PIC16 ce ne vogliono necessariamente tre (di somma).
Questo esempio mostra una cosa importante: il principale problema della programmazione è far eseguire al microcontrollore le funzioni che desideriamo, ma utilizzando le sole istruzioni che il dispositivo (o il linguaggio di programmazione, come vedremo più avanti) mette a disposizione.

IL CODICE OGGETTO ED IL LINGUAGGIO ASSEMBLY

Facciamo un primo esempio di programmazione: supponiamo ci venga chiesto di fare in modo che il nostro microcontrollore ATmega16 legga un numero binario dal port A, gli cambi il segno (cambiare il segno di un numero binario intero significa invertire il valore di tutti i bit e sommare '1': l'operazione è detta *complemento a due*), e scriva il risultato sul port B. La **Fig. 2** mostra i soggetti coinvolti in queste operazioni (esclusa la memoria di programma, che ormai consideriamo sottintesa in quanto deve contenere le necessarie istruzioni).
Sulla base di queste informazioni (*dati di ingresso* del problema) bisogna ricavare la sequenza di operazioni da svolgere (l'*algoritmo*) e quindi tradurla in operazioni eseguibili dal microcontrollore. La sequenza delle operazioni potrebbe essere la seguente:

1. *lettura dato su PINA e memorizzazione su variabile interna R0;*
2. *complemento a 2 di R0;*
3. *scrittura di R0 su PORTB.*

Questo è in pratica l'algoritmo del programma, ovvero la descrizione formale dei passi necessari per risolvere il problema. PINA è un *registro* (ovvero una cella di memoria RAM con funzioni particolari) che contiene il valore dei livelli logici presenti fisicamente sui piedini del PORT A. La variabile interna (R0) è necessaria in quanto ci consente di fare una copia del valore del PORT A per poterlo manipolare; R0 è il primo dei 32 registri interni della CPU nell'ATmega16 che possono essere utilizzati per questa operazione.
La fase successiva è la traduzione (detta anche *implementazione*) dell'algoritmo nelle operazioni eseguibili dal microcontrollore; come già sappiamo, questo significa scrivere codici binari interpretabili dalla CPU: prendendo il manuale di programmazione dell'ATmega16 liberamente scaricabile dal sito Atmel [1], andiamo dunque a cercare i codici corrispondenti a ciascuna delle operazioni che ci

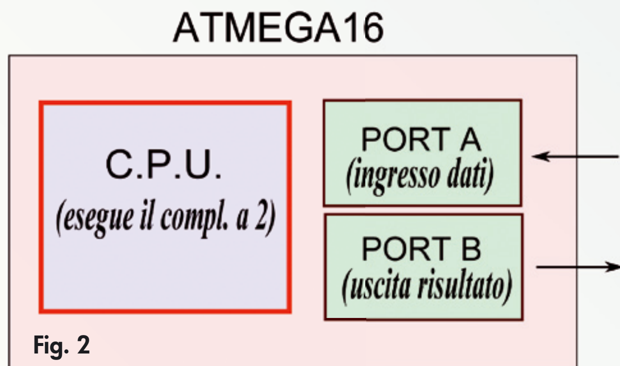


Fig. 2

Struttura dei programmi in **linguaggio C**

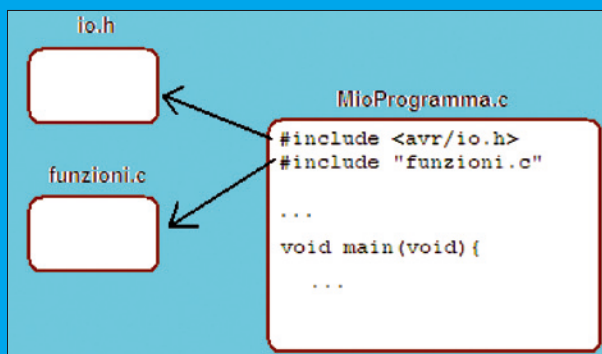
Un programma in C è costituito da uno o più documenti di testo semplice, creati con un qualsiasi editor (tipo il "blocco note" di Windows), che costituiscono *i sorgenti* (notare l'articolo maschile: in informatica si dice *il sorgente* e non la sorgente) del programma. Spesso i programmi sono suddivisi in più documenti, per non avere un unico file di dimensioni "eccessive" ma anche (e soprattutto) per separare sezioni di codice con funzioni precise a tutto vantaggio di leggibilità e manutenibilità (che ricordiamo essere caratteristiche molto importanti di un "buon" sorgente). In C esistono fondamentalmente due tipi di documenti: gli header (che hanno estensione ".h") e i file di codice (che hanno estensione ".c"). Questi ultimi contengono appunto il codice sorgente vero e proprio, mentre gli header contengono le dichiarazioni dei tipi di dati e funzioni (come il file "io.h" contenente le associazioni nome-indirizzo dei registri del microcontrollore). Il contenuto degli header quindi non serve direttamente a creare codice per il microcontrollore, ma ad istruire il compilatore sul significato di particolari definizioni che possiamo adottare nel programma; ad esempio, come ben sa chi ha già scritto qualche programma in C per elaboratori "tradizionali" (quali i PC), non si può usare la funzione di stampa a schermo "printf()" senza aver prima incluso il file di header "stdio.h" che ne contiene la definizione.
Ogni programma C dev'essere composto da almeno un file di codice (.c) cosiddetto principale, in quanto contiene la funzione principale `main()`. In funzione della lunghezza e delle esigenze del programma, ad esso possono essere agganciati, con direttive "`#include`", un numero arbitrario di altri file di codice o di header. Normalmente i file aggiuntivi possono essere situati in qualsiasi locazione, in quanto nella dichiarazione bisogna sempre specificare il percorso completo o relativo rispetto al file principale:

```
"#include "C:\pippo.h" /* include il file pippo.h che
si trova in C:\ */
"#include "pippo.h" /* include un header contenuto nel-
la stessa cartella del file principale */
```

Se il file da includere fa parte degli header di dichiarazione delle librerie di sistema, il compilatore è già istruito sui percorsi della loro installazione e dunque inseriamo il nome tra freccette anziché doppie virgolette: la seguente espressione

```
#include <avr/io.h>
```

carica il file *io.h* che si trova nella cartella *avr*, il cui percorso è già "noto" al compilatore avr-gcc in quanto gli viene passato tra i parametri di compilazione (per una spiegazione esauriente di questi argomenti rimandiamo al manuale del gcc).
In definitiva la struttura completa di un *programma sorgente* può essere come quella dell'esempio mostrato in figura, composta da tre file, due di codice ("MioProgramma.c" e "funzioni.c" contenuti nella stessa cartella) ed un header ("io.h"); "MioProgramma.c" è il file sorgente principale, che sarà infine fatto leggere al compilatore per la creazione del codice eseguibile del microcontrollore.



Programma sorgente completo.

interessano; per comodità sono riportati (in esadecimale) nella **Tabella 1**.
Pertanto il codice completo del nostro programma è il seguente:

```
B209
9401
BA08
```

Questo codice (detto *codice oggetto* o *linguaggio macchina*) andrà infine scritto nella memoria programma del microcontrollore, in modo da farlo leggere ed eseguire alla CPU quando accendiamo il sistema. L'insieme delle istruzioni caricate in memoria forma il programma completo e viene talvolta chiamato *immagine eseguibile*.
Ovviamente questo è un programma molto semplice, composto di appena tre istruzioni; nella realtà è facile che le righe di codice siano qualche centinaio anche solo per il programma di gestione di un termostato; è pertanto intuibile che la scrittura di una lunga lista di sterili bit diventi molto complicata, oltre ad essere completamente illeggibile (a meno di non conoscere a memoria il codice di ogni istruzione).
Di conseguenza, con la diffusione dei primi microprocessori si è subito cercato il modo di poter scrivere le istruzioni in maniera umanamente "comprensibile" associando a ciascuna una sigla che, in modo più o meno immediato, ricordi il significato dell'istruzione stessa: è così nato un codice cosiddetto mnemonico, che è universalmente conosciuto come codice (o linguaggio) *Assembly*.

Descrizione operazione	Linguaggio macchina corrispondente
Lettura dato da PINA in R0	B209
Complemento a 2 di R0	9401
Uscita R0 su PORT B	BA08

Tabella 1

Tabella 2

Descrizione operazione	Linguaggio macchina corrispondente	Codice Assembly
Lettura dato da PINA in R0	B209	IN R0, PINA
Complemento a 2 di R0	9401	NEG R0
Uscita R0 su PORT B	BA08	OUT PORTB, R0

Pertanto alla **Tabella 1** possiamo aggiungere una colonna, in cui scriviamo il codice assembly corrispondente alle rispettive istruzioni; otteniamo così la **Tabella 2**.
Si può notare che il codice Assembly "richiama" subito il tipo di operazione svolta: IN sta per input (ingresso dato), NEG per negazione (dunque complemento a 2), OUT per output (uscita dati); il codice scritto in questo modo acquisisce subito maggiore chiarezza e leggibilità, oltre ad essere più facile ed immediato da scrivere.
Naturalmente il codice Assembly non è direttamente utilizzabile, in quanto il microcontrollore è in grado di comprendere il solo linguaggio macchina; pertanto, i programmi in Assembly si scrivono normalmente su PC con un qualsiasi foglio di testo, che viene successivamente "dato in pasto" ad un apposito programma assemblatore (detto Assembler, funzionante sempre su PC) il quale produce in uscita un file contenente il linguaggio macchina corrispondente, i cui bit vengono infine caricati nella memoria di programma del microcontrollore.
Una cosa molto utile della scrittura in Assembly è la possibilità di aggiungere commenti testuali a piacimento, in quanto per ogni riga l'assemblatore ignora il carattere '**;**' (punto e virgola) e tutto ciò che si trova alla sua destra.
Il nostro programma potrebbe dunque assumere il seguente "aspetto":

```
; **** Programmino in codice assembly con commenti****
IN R0, PINA      ; Lettura dato da PORT A (registro 'PINA') su R0
NEG R0           ; Complemento a 2 di R0
OUT PORTB, R0    ; Trasferimento R0 su PORT B
; **** Fine programma ****
```

Scartando i commenti, il codice macchina prodotto dall'assemblatore rimane sempre lo stesso, ma il programma Assembly diventa ancora più chiaro a chi lo legge. Questo aspetto è molto importante in quanto è legato, oltre alla chiarezza immediata, anche alla *manutenibilità* del software: specialmente in ambito professionale, infatti, può esservi necessità di "rimetterci le mani" dopo diverso tempo anche da parte di persone diverse dall'autore; è quindi essenziale che i programmi siano il più documentati possibile (anche attraverso i commenti inseriti nel programma stesso).
Nonostante i suoi vantaggi, l'Assembly è un linguaggio cosiddetto *di basso livello*, in quanto vi è una corrispondenza "uno a uno" con le istruzioni della CPU del microcontrollore: questo significa la scrittura del programma sarà comunque condizionata al tipo di istruzioni disponibili per il dispositivo utilizzato, che il programmatore deve conoscere. In altre parole, ogni tipo di microcontrollore, avendo un proprio set di istruzioni, ha anche un proprio linguaggio Assembly. Per rendervi chiaro il concetto, riscriviamo il nostro programmino d'esempio per un microcontrollore della famiglia HC08 (Freescale):

```
; **** Programmino in codice assembly per uC della famiglia HC08 ****
LDA PORTA          ; Lettura dato da PORT A sul registro A ('accumulatore')
NEGA               ; Complemento a 2 di A
STA PORTB          ; Trasferimento di A su PORT B
; **** Fine programma ****
```

Si nota che, oltre al codice "mnemonico" delle istruzioni, è cambiato anche il nome della variabile che usiamo per memorizzare temporaneamente il dato, in quanto nei processori HC08 non c'è il registro R0 ma un registro accumulatore (A). Le cose si complicherebbero ulteriormente se si dovesse scrivere il codice per i controllori della famiglia PIC16, i quali non prevedono neppure l'istruzione di complemento a 2.

LINGUAGGI DI *ALTO LIVELLO*

Fortunatamente, oggigiorno tante famiglie di microcontrollori dispongono di compilatori per linguaggi di alto livello (primo fra tutti il **C**), che permettono la scrittura di codice complesso senza preoccuparsi troppo di qual è il processore su cui verrà fatto "girare"; con l'eccezione di particolari definizioni (che vedremo più avanti), il codice è trasportabile da un processore all'altro, in quanto deve aderire allo standard del linguaggio utilizzato. Così la scrittura del nostro programma utilizzando il linguaggio C diventa la seguente:

```
/**** Programmino in codice C ****/
char MiaVariabile; /* dichiaro una variabile per
dato temporaneo */
MiaVariabile = PINA;        /* Lettura valore da
PORT A (reg. PINA) */
MiaVariabile = MiaVariabile*(-1);/* Moltiplico
per -1: inversione segno*/
PORTB = MiaVariabile; /* scrittura risultato su
PORTB */
```

Naturalmente anche in questo caso bisogna conoscere il linguaggio che si intende utilizzare, ma il grande vantaggio è che il codice così scritto è pressoché indipendente dal tipo di microcontrollore che si utilizzerà, in quanto sarà il compilatore a creare il codice oggetto adatto al micro su cui desideriamo far girare il programma. Ad esempio, non bisognerà più preoccuparsi di verificare se il processore supporta l'operazione di complemento a 2 oppure no, perché il compilatore genererà in ogni caso il codice necessario per svolgere il compito richiesto. Bisogna però conoscere sempre il nome dei registri legati alle periferiche; il semplice programma in C visto sopra, legge il dato dal registro "PINA":

```
MiaVariabile = PINA;
```

Dunque, va bene per i micro ATmega, mentre se volessimo utilizzare un micro HC08 dovremmo scrivere "PORTA", in quanto è da tale registro che si leggono i dati:

```
MiaVariabile = PORTA;
```

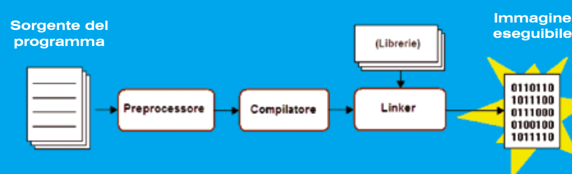
Come per l'assemblatore, anche per i linguaggi ad alto livello bisogna procurarsi il programma specifico (il compilatore, appunto...) che legge il file di testo contenente il "sorgente" del programma e genera il codice oggetto per il microcontrollore voluto. Normalmente questi compilatori "girano" su un comune PC; poiché non generano codice adatto al processore del PC (famiglie 80x86, Pentium o altri) bensì ad un determinato microcontrollore, sono anche detti *cross-compilatori*.
Il problema è che a differenza dei programmi assemblatori, che i costruttori di microcontrollori forniscono sempre gratuitamente, i compilatori sono spesso a pagamento e talvolta si pagano a caro prezzo. La scelta di un micro della famiglia ATmega per le nostre esercitazioni è stata guidata anche dal fatto che l'ATMEL rende disponibile il compilatore in linguaggio C/C++ **avr-gcc** [3] che, essendo una variante per i micro della Atmel del noto compilatore *open-source* **gcc**, è completamente gratuito.

Dal sorgente all'eseguibile: le fasi della creazione dei programmi

Chi volesse analizzare più nel dettaglio il procedimento necessario per ottenere, a partire dal sorgente nel linguaggio C (quindi composto da documenti di codice ed headers), il codice binario da caricare nella memoria di programma del microcontrollore (la cosiddetta "immagine eseguibile"), può osservarne le fasi schematizzate nella figura in fondo a questo riquadro.
Il *preprocessore* esegue una prima elaborazione sui sorgenti, in quanto il suo compito principale è l'esecuzione delle direttive ad esso rivolte (ovvero quelle che iniziano con il carattere '#', come "`#include`" o "`#define`"). Con tali direttive, il programmatore può tra le altre cose decidere quali file devono essere coinvolti nel processo di compilazione: se nel sorgente principale vi è una riga con la direttiva

```
#include "pippo.c"
```

Il preprocessore sostituirà tale riga con l'intero contenuto del file pippo.c (che ovviamente deve esistere nella stessa cartella del sorgente principale). Il codice risultante viene poi ripulito dai commenti ed inviato al *compilatore*, che lo trasforma effettivamente nel codice oggetto (binario) composto dalle istruzioni nel linguaggio macchina del microcontrollore.
Opzionalmente, il codice così creato può ancora dover essere trattato dal *linker* se il programma deve utilizzare particolari funzioni, il cui codice è contenuto in appositi file detti "librerie". Infatti il compito del linker è inserire nel modo opportuno il codice oggetto di tali funzioni in quello, creato dal compilatore, del nostro programma. Il file di codice così creato (*l'immagine eseguibile*) è pronto per essere "scaricato" nella memoria del microcontrollore.
Anche se abbiamo parlato di tre fasi distinte di elaborazione, esse fanno normalmente parte di un unico pacchetto: quando ci si riferisce ad un programma compilatore è implicito che esso sia in grado di eseguire, oltre alla compilazione vera e propria, i compiti del preprocessore e del linker. Pertanto nel corso utilizzeremo il termine "compilazione" per definire il processo complessivo di creazione dell'eseguibile a partire dai sorgenti.



```
void main(void) {

    char MiaVariabile;

    MiaVariabile = PINA;
    MiaVariabile = MiaVariabile * (-1);
    PORTB = MiaVariabile;
}
```


Fig. 3

PERCHÉ IL LINGUAGGIO C?

Tra i linguaggi ad alto livello, nel mondo dei microcontrollori il C "la fa da padrone assoluto"; i motivi principali sono:

- conoscenza e diffusione; negli ambienti tecnici il C è il linguaggio più conosciuto e utilizzato, anche su PC e Workstation, quindi per i programmatori il passaggio al mondo dei microcontrollori è meno "traumatico" rispetto alla scrittura di codice Assembly;
- portabilità; il C esiste per numerose piattaforme (microprocessori e microcontrollori), per cui lo stesso programma può essere ricompilato per funzionare su sistemi diversi con differenze minime (almeno in teoria);
- efficienza; quasi tutti i compilatori C possono essere impostati per creare codice oggetto ottimizzato all'applicazione, nel senso che è possibile scegliere, a parità di codice sorgente, se generare il minor numero possibile di istruzioni in linguaggio macchina (ottimizzazione in spazio) oppure scegliere le istruzioni che possono essere eseguite il più veloce possibile (ottimizzazione in velocità).

Inoltre, essendo un linguaggio di alto livello (e dunque con istruzioni più potenti e "vicine" al modo di esprimersi delle persone), permette di accorciare notevolmente il tempo di sviluppo rispetto al linguaggio Assembly, il quale viene oramai utilizzato raramente.
Alla luce di queste considerazioni, nel corso impareremo a sviluppare programmi per l'ATmega16 esclusivamente in C, approfittando anche del fatto che il relativo compilatore è completamente gratuito e scaricabile da Internet [3]. Nella prossima puntata ci preoccuperemo nel dettaglio della sua installazione, insieme all'ambiente di sviluppo necessario ad iniziare le prove pratiche.
Nel seguito di questa puntata vedremo, invece, alcuni fondamenti di programmazione in tale linguaggio, evidenziandone soprattutto le differenze rispetto al C standard (ANSI C), del quale in Internet si può trovare innumerevole documentazione.

STRUTTURA DEL PROGRAMMA

La struttura "minima" di un programma scritto in C è la seguente (ricordiamo che tutto il testo scritto tra i caratteri "/*" e "*/" viene trattato dal compilatore come un commento, e dunque non genera codice macchina):

```
void main(void) { /* inizio funzione principale 'main' */

/* mettere qui la dichiarazione variabili locali */
...
/* scrivere qui il codice della funzione */
...

} /* fine funzione 'main' */
```

La dichiarazione iniziale "`void main(void)`" non è un'istruzione, ma indica al compilatore che il codice che segue tra le parentesi graffe è la *funzione principale* (chiamata appunto *main*), ovvero quella che il microcontrollore inizia subito ad eseguire dopo aver rilasciato il reset (che viene generalmente attivato almeno all'accensione del sistema, quindi si parla di power-on reset) e dunque dev'essere sempre presente in un programma scritto in C. Talvolta, in programmi molto semplici, è anche l'unica.
Questa funzione (come peraltro tutte le funzioni dei programmi C) è suddivisa nelle dichiarazioni delle eventuali variabili locali (che servono a gestire tutti i dati "temporanei" dell'applicazione e devono essere dichiarate prima che vengano utilizzate) e nel codice relativo alla funzione vera e propria.
Le voci "`void`" indicano che questa funzione non necessita di parametri supplementari quando viene "chiamata" e non ritorna nulla quando "finisce"; ciò costituisce una differenza importante rispetto ad un programma scritto per un calcolatore tipo PC, il quale viene mandato in esecuzione dal sistema operativo della macchina e può prevedere alcuni parametri aggiuntivi. In un sistema a microcontrollore, invece, la funzione principale main() del programma viene avviata

direttamente e automaticamente all'accensione del sistema (o meglio, dopo il reset del microcontrollore) per cui nessuno può specificare parametri aggiuntivi da passargli.
Il programmino di esempio che avevamo già visto può pertanto diventare il programma principale del nostro microcontrollore, a patto che sia scritto all'interno della funzione main(), come riportato nella **Fig. 3**, dove sono state evidenziate le sezioni di dichiarazione variabili e di codice.
In tal modo questo codice va direttamente in esecuzione quando il sistema viene acceso: il valore letto sul PORT A (registro chiamato PINA) viene memorizzato in `MiaVariabile`, quindi il segno viene invertito ed il risultato è riportato in uscita sul PORT B.
E dopo, cosa succede? Avevamo visto nella scorsa puntata che la CPU del microcontrollore legge ed esegue le istruzioni in modo continuativo, ovvero terminata l'istruzione 'N' passa alla 'N+1', poi alla 'N+2' e così via, senza mai fermarsi; pertanto dopo aver scritto il valore di `MiaVariabile` nel PORT B, la CPU va a cercare l'istruzione successiva, che però non esiste in quanto il programma è terminato. Se il programma girasse su un PC, il problema non sussisterebbe in quanto il controllo tornerebbe al sistema operativo, mentre in un microcontrollore bisogna fare in modo che l'esecuzione della funzione principale *non termini mai*, a meno che il sistema non venga fisicamente spento. In pratica ciò significa che durante l'esecuzione del programma la CPU non deve mai "arrivare" alla parentesi graffa '}'di chiusura della funzione main(). In teoria si potrebbe inserire in fondo al programma, prima di chiudere il main, un'istruzione di STOP che di fatto blocca la CPU indefinitamente. L'ATmega prevede una istruzione di questo tipo (detta *SLEEP*): la CPU va in uno stato "dormiente" e può essere risvegliata solo da particolari eventi, come l'asserzione del segnale di *reset*, che la farebbe ripartire dall'inizio.
Una soluzione più pratica è quella di far tornare automaticamente il processore alla prima istruzione, dopo che ha eseguito l'ultima, creando così un ciclo perpetuo in cui viene chiuso l'intero programma. Nel linguaggio C vi sono diverse espressioni per creare cicli di esecuzione; una di queste è il comando while, che ha la seguente struttura:

```
while (condizione)  {
        ... /* corpo del while */
}
```

Il funzionamento è semplice: il codice racchiuso tra le parentesi graffe (il corpo del while) viene continuamente eseguito finché la condizione (che può essere un numero o una variabile) è VERA (in C significa che è diversa da zero). Quando il valore di condizione diventa zero (condizione FALSA), l'esecuzione prosegue con le istruzioni successive al ciclo.
Riscriviamo dunque il nostro esempio, racchiudendo il codice in un ciclo while:

```
void main(void) {
  char MiaVariabile;
  while (1) { /* inizio ciclo 'while' */
MiaVariabile = PINA;
MiaVariabile = MiaVariabile * (-1);
PORTB = MiaVariabile;
  } /* fine ciclo 'while': il processore torna
al suo inizio */
}
```

Poiché la condizione del ciclo while è il numero "1", essa è sempre vera, dunque il processore non uscirà mai dal ciclo e quindi neppure dal programma. Operativamente questo vuol dire che, dopo aver letto un valore dal PORT A, avergli cambiato segno e mandato il risultato sul PORT B, il processore torna alla lettura del PORT A e tutto si ripete, sino a quando non spegniamo il sistema.
Il programma così creato in realtà non funziona ancora correttamente, in quanto necessita di un'altra sezione, fondamentale quando si lavora con i microcontrollori: l'*inizializzazione*. Questa fase consiste nella scrittura di determinati valori in alcuni registri e periferiche interni del microcontrollore e va eseguita immediatamente dopo l'accensione (o il reset) del sistema del sistema, per due motivi:
- quando il µC viene alimentato, la RAM interna che contiene le variabili del programma possono essere in uno stato "sconosciuto" (bit a zero oppure a uno in modo casuale);
- altri registri vengono forzati (per effetto del reset) con tutti i bit a zero, ma per funzio-

Fig. 4 - Registri DDRA e DDRB.

DDRA – Port A Data Direction Register

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRB – Port B Data Direction Register

Bit	7	6	5	4	3	2	1	0	
	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDxn	I/O
0	Input
1	Output

nare secondo le esigenze del programma bisogna imporre loro un altro valore.

Nel nostro programma, ad esempio, abbiamo assunto che il PORT A è un ingresso dati ed il PORT B un'uscita. Se però consultiamo il datasheet dell'ATmega16 (che potete liberamente scaricare dal link indicato al punto [2] dei riferimenti) nella sezione "I/O Ports" scopriamo che sia il PORT A sia il PORT B sono *bidirezionali* (cosa normale nei microcontrollori) e vengono impostati come ingresso o come uscita sulla base del valore dei bit di appositi registri di direzione dati (Data Direction Register, DDR): ogni port ha il proprio DDR, ed ogni bit del DDR stabilisce la direzione del relativo bit del port (bit del DDR = 0 → bit del port come ingresso; bit del DDR = 1 → bit del port come uscita). In **Fig. 4** sono riportati i due registri DDRA e DDRB ed una tabella che mostra la relazione tra il valore dei loro bit e la direzione dei bit rispettivi dei port.
Le righe "Initial value" mostrano lo stato in cui vengono posti automaticamente i bit dei due registri in seguito all'accensione (o comunque un reset) del sistema; come si nota, sono messi tutti a 0, il che corrisponde ad impostare tutte le linee dei port A e B come ingressi. Appena partirà, il programma dovrà andare subito a modificare questi valori, per impostare la direzione delle linee nel modo richiesto.
Ad esempio, se in DDRA scriviamo il valore 0x0F (= 00001111 in binario) poniamo come uscite i primi quattro bit del PORT A (cui fanno capo i piedini PA0...PA3 del micro) e come ingressi gli altri (PA4...PA7). Attenzione: nel seguito indicheremo gruppi di bit indicandone solo gli estremi, racchiusi tra freccette e separati dal carattere ':'; ad esempio, per indicare il gruppo di quattro bit PA0...PA3 scriveremo PA<3:0>.
Pertanto anche nel nostro programma dobbiamo preoccuparci di impostare correttamente la direzione dei piedini dei port prima di utilizzarli: l'inizializzazione va quindi fatta, sempre nella funzione main, subito dopo la dichiarazione delle variabili locali.

```
void main(void) {
  char MiaVariabile;
/* Inizializzazione */
  DDRA = 0x00;  /* tutti i bit di DDRA = 0:
PA<7:0> sono 8 ingressi */
  DDRB = 0xFF;  /* tutti i bit di DDRB = 1:
PB<7:0> sono 8 uscite */
/* Fine inizializzazione */
  while (1) {
MiaVariabile = PINA;
MiaVariabile = MiaVariabile * (-1);
PORTB = MiaVariabile;
  }
}
```

Da notare che l'inizializzazione si trova all'esterno del ciclo while e pertanto viene eseguita una volta sola dopo l'accensione, ma è sufficiente in quanto, una volta impostati, i registri DDRA e DDRB non vengono più modificati.
Nota: all'accensione del sistema tutti gli I/O port del micro vengono impostati come ingressi; tuttavia conviene impostare quelli non utilizzati come delle uscite. La regola è valida per tutti i microcontrollori e nasce da una considerazione: avere degli ingressi non utilizzati e quindi scollegati (flottanti) può introdurre disturbi tali da alterare l'esecuzione del programma.
Siccome l'ATmega16 ha anche i PORT C e D, dovremo aggiungere nell'inizializzazione le due istruzioni seguenti:

```
DDRC = 0xFF;  /* PC<7:0> non utilizzati: confi-
guro come uscite */
DDRD = 0xFF;  /* PD<7:0> non utilizzati: confi-
guro come uscite */
```

Bene, abbiamo completato il nostro programma dal punto di vista funzionale, ma se lo dessimo in pasto al compilatore esso ci riporterebbe un errore, dicendoci che non sa cosa sono le parole DDRA, DDRB, PINA, PORTB; in effetti il compilatore si aspetta di trovare al loro posto un numero binario che specifica l'indirizzo "fisico" del registro all'interno del microcontrollore.
Per esempio, il registro PINA si trova all'indirizzo 0x19 (nel datasheet dell'ATmega16 trovate l'elenco degli indirizzi di tutti i registri nella sezione "Register Summary"), per cui nel programma dovremmo scrivere 0x19 al posto di PINA, 0x1A al posto di DDRA e così via. Alternativamente si può inserire, ad inizio programma (cioè prima del main), la seguente riga:

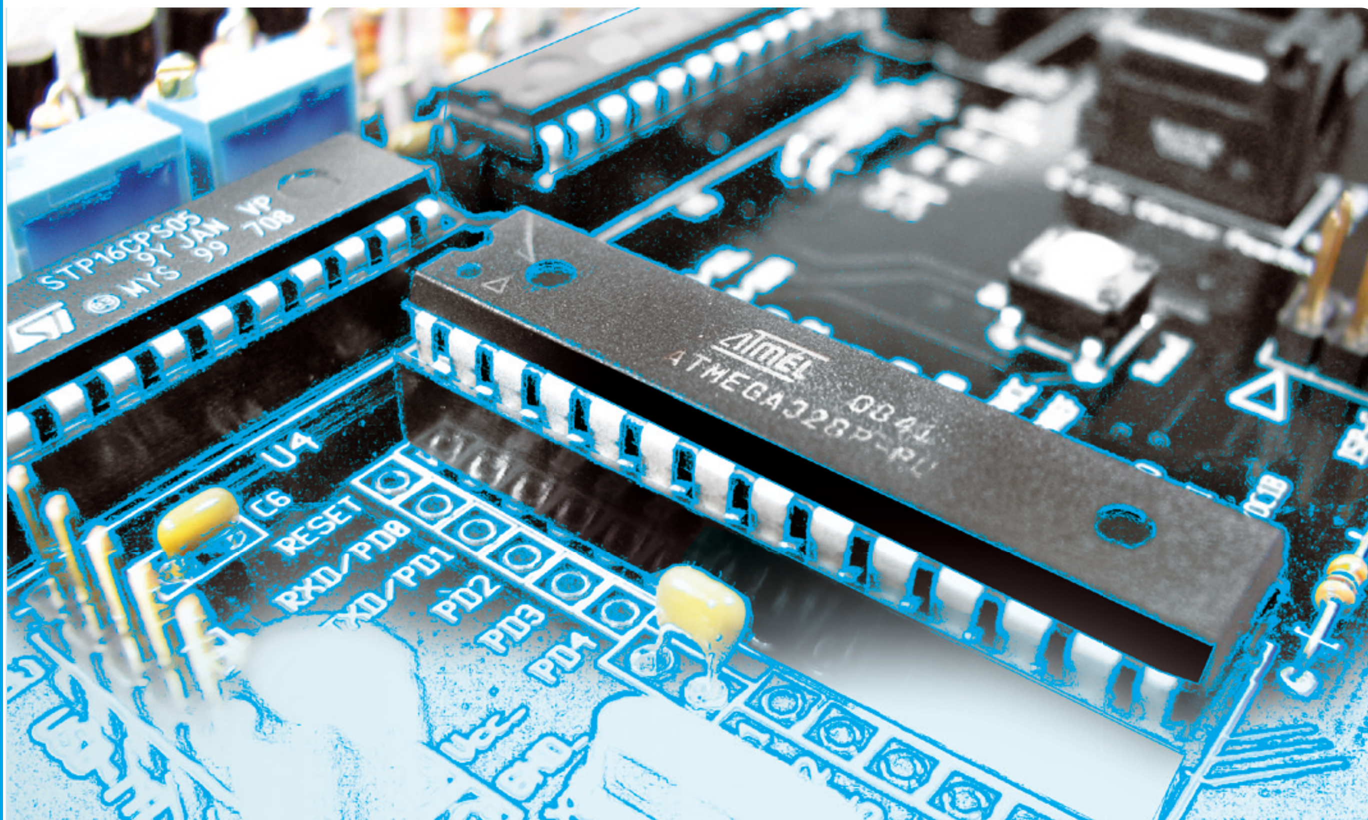
```
#include <avr/io.h>
```

Essa è una *direttiva del preprocessore* (si veda il riquadro "Dal sorgente all'eseguibile") che, quando lanciamo la compilazione, fa inserire automaticamente nel nostro programma il file "io.h" il quale contiene le associazioni *nome* ↔ *indirizzo* di tutti i registri dei micro della famiglia AVR (quindi anche gli ATmega).
Abbiamo così scritto un semplice ma completo programmino in C; nella prossima puntata affronteremo l'installazione del compilatore avr-gcc e dell'ambiente di sviluppo *AVR Studio*, per essere in grado di scrivere e compilare i nostri programmi direttamente sul PC.

Riferimenti

[1] ***http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf***
[2] ***www.atmel.com/dyn/resources/prod_documents/doc2466.pdf***
[3] ***http://winavr.sourceforge.net/***

DENSION
WiRC
RADIOCOMANDO SU RETE Wi-Fi
cod. WIRC
€132,00
Utilizza il tuo iPhone o iPad per pilotare da remoto qualsiasi modello sfruttando la telecamera montata a bordo.
Prezzi IVA inclusa.
FUTURA ELETTRONICA
Via Adige, 11
21013 Gallarate (VA)
Tel. 0331/799775 • Fax. 0331/792287
Maggiori informazioni su questo prodotto e su altri dispositivi per robotica sono disponibili sul sito *www.futurashop.it* tramite il quale è anche possibile effettuare acquisti on-line.



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

Procuriamoci ed impariamo ad utilizzare gli strumenti software che ci consentono di compilare, caricare ed emulare il programma scritto per l'ATmega16, che è il processore scelto per le nostre prove. Terza puntata.

Dopo aver definito, nella prima puntata, che cosa sono microprocessori e microcontrollori (i componenti che più c'interessano per questa trattazione) e nella seconda i fondamenti della programmazione e lo scopo che essa ha, in queste pagine passiamo a spiegarvi come reperire ed installare gli strumenti software indispensabili per sviluppare e compilare un programma in C per microcontrollori; in particolare, concentreremo l'attenzione sugli strumenti occorrenti a lavorare con l'ATmega16, che è il micro da noi scelto per la trattazione. Con questi strumenti mostreremo anche come sia possibile simulare un programma per verificarne il funzionamento prima ancora di scaricarlo nel microcontrollore. Per spiegarvi come procedere, partiamo dal presupposto che già abbiate a disposizione un Personal Computer (un desktop, un tower o anche un notebook) nel quale sia installato un sistema operativo di tipo Microsoft Windows e che ovviamente ne conosciate l'uso.

I "TOOLS" DI SVILUPPO: IL PACCHETTO *WINAVR*

Stabilito ciò, diciamo che gli strumenti strettamente necessari per sviluppare codice per i microcontrollori sono, in linea generale, un *editor* di testo ed un compilatore; il primo serve a scrivere il programma "sorgente"

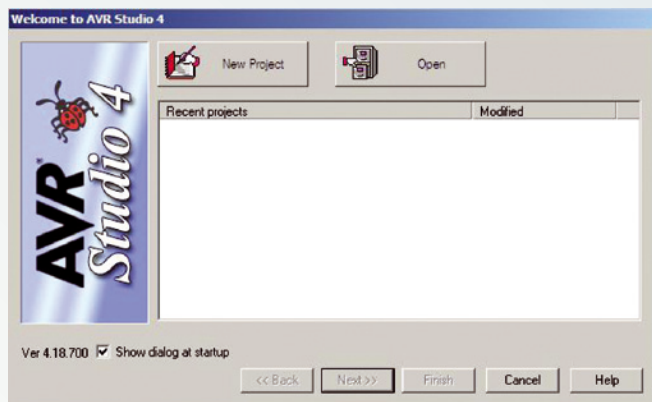


Fig. 1

ed il secondo a creare il file compilato interpretando le istruzioni scritte ed adattandole al nostro micro. Come già anticipato, per i microcontrollori con CPU della famiglia "AVR" della Atmel (dei quali fa parte l'ATmega16) viene fornito gratuitamente il compilatore **avr-gcc**, il quale fa parte di un "pacchetto" (detto *suite*) completo di programmi (detti *strumenti* o -in inglese- *tools*) che servono allo sviluppo su PC di programmi per microcontrollori; questo pacchetto si chiama ***WinAVR*** ed è scaricabile dal sito Internet indicato al punto [1] dei Riferimenti, sotto forma di singolo file di installazione di Windows. Nella stessa pagina web si trovano diverse versioni di WinAVR, ordinate per data di rilascio; consigliamo di scaricare la versione più recente, in quanto dovrebbe contenere sempre migliorie rispetto a quelle precedenti (ottimizzazioni, correzioni di eventuali problemi riscontrati, ecc.). In seguito, si può fare una verifica annuale del rilascio di nuove versioni e decidere se procedere o meno all'aggiornamento, tuttavia l'operazione non è necessaria se non si riscontrano particolari problemi sulla versione in uso.
L'ultima versione rilasciata durante la stesura di questo corso è datata 20/01/2010 ed è quella cui faremo riferimento nel seguito. Dal sito sopra citato clicchiamo dunque sulla cartella "20100110" (notare che le prime quattro cifre si riferiscono all'anno e le due seguenti al mese di rilascio), per accedere al file di installazione "WinAVR-20100110-install.exe". Cliccandovi sopra si viene indirizzati alla pagina del download, che dovrebbe far comparire nel vostro browser la richiesta di salvataggio del file sul vostro PC. Dopo un tempo più o meno breve (dipendente dalla vostra connessione Internet) vi troverete il file scaricato; mandatelo in esecuzione, senza curarvi dei messaggi di "attenzione" di Windows circa l'affidabilità del software. Accettate la licenza d'uso, specificate una cartella di destinazione dei programmi (consigliamo di lasciare quella preimpostata), ed installate il tutto lasciando selezionati tutti gli articoli proposti (lo spazio occorrente su disco è meno di 300 MB). Dopo l'installazione, per la quale sono richiesti circa due minuti, appare il messaggio che ne conferma il completamento; a questo punto cliccate su "Fine" e il WinAVR è pronto all'uso.
Ora nel vostro browser dovrebbe automaticamente aprirsi una finestra che mostra una pagina web (che avete già in locale in quanto è stata installata con il pacchetto) contenente il manuale utente del WinAVR (in inglese), ovvero la spiegazione di tutte le caratteristiche ed i programmi facenti parte il pacchetto e dove sono stati installati; vi si può dare una veloce lettura a livello di curiosità, ma non è certo indispensabile.
Con la suite WinAVR abbiamo quindi installato nel nostro PC il compilatore C/C++ chiamato *avr-gcc*, l'assemblatore chiamato *avr-as* (utile a chi fosse interessato a scrivere qualche programma in Assembly) e numerosi altri programmi di varia utilità. Va rimarcato che queste applicazioni sono *a riga di comando*, come pure il compilatore avr-gcc stesso; per compilare un file sorgente chiamato "MioProgramma.c" bisogna aprire una console di Windows (*Start→Programmi→Accessori→ Prompt dei comandi*), spostarsi all'interno della cartella che contiene il file, digitare: "avr-gcc MioProgramma.c" e premere Invio.
Fortunatamente per chi è ormai abituato a lavorare solo con mouse, menu a tendina e pulsanti, esiste anche un ambiente di sviluppo che dà una veste grafica al nostro compilatore: l'*Avr Studio di Atmel*.

AVR STUDIO

Per i suoi microcontrollori, Atmel fornisce un *ambiente di sviluppo integrato* (detto IDE, dall'inglese *Integrated Development Environment*) chiamato "AVR Studio", che permette di gestire tutte le operazioni di scrittura e compilazione del programma (e, come vedremo, anche di emulazione) all'interno di un unico ambiente in maniera molto intuitiva. L'unica cosa che "gli manca" (come la stragrande maggioranza degli IDE per microcontrollore gratuiti) è la possibilità di compilare programmi scritti in C; tuttavia

può interfacciarsi con alcune applicazioni di WinAVR (fra le quali il compilatore avr-gcc), per cui potremo, all'interno di questo ambiente, scrivere i programmi sorgenti (anche in C) e compilarli semplicemente cliccando su un tastino.
Passiamo alla sua installazione: AVR Studio è scaricabile (gratuitamente) dal sito della Atmel (punto [2] dei Riferimenti); in alternativa scrivete "Avr studio" in un qualsiasi motore di ricerca e sicuramente vi verrà proposta la stessa pagina. Nella sezione "software" bisogna cliccare, nella riga che riporta "AVR Studio 4.18", sull'icona del disco con scritto "Register", in quanto bisogna fare una breve registrazione sul sito per poter scaricare il programma; chi non vuole farlo, deve cercare di reperirlo da qualche altro sito non ufficiale. Dopo aver scaricato il file di installazione (circa 120 MB), lo si esegue, preferibilmente lasciando tutte le impostazioni predefinite. Conclusa l'installazione di AVR Studio è consigliato applicarvi subito l'ultimo "Service pack" disponibile, che si scarica dalla stessa pagina dell'AVR Studio, sotto la voce "AVR Studio 4.18 SPx" dove 'x' è il numero di versione. Eseguita anche quest'ultima operazione, siamo finalmente operativi!

FAMILIARIZZIAMO CON L'AMBIENTE DI SVILUPPO: I PROGETTI

Dal menu *Start* di Windows, selezioniamo *Programmi→Atmel AVR Tools→AVR Studio4* per avviare l'IDE; subito si apre la schermata di avvio del programma (finestra di **Fig. 1**). Questa finestra permette di creare nuovi progetti o aprirne uno già esistente; inoltre mostra l'elenco dei progetti già creati. Notate che dopo una nuova installazione del programma, è normale che la lista sia vuota.
Ma che cos'è un progetto? È la raccolta delle informazioni che servono ad AVR Studio per gestire i programmi che andremo a creare. Ad esempio, nel progetto vi sono le indicazioni sui file sorgente che devono essere compilati, le direttive da dare al compilatore, il microcontrollore che verrà utilizzato ed altro ancora. Per chiarire meglio il concetto, creiamo subito un nuovo progetto: dalla finestra di **Fig. 1** clicchiamo su "New Project", cosicché apparirà la finestra di **Fig. 2**. In essa diamo un nome ed assegniamo un percorso al nostro nuovo progetto. Per fare ciò bene seguire i seguenti passi:

Fig. 2

- in "Project type", cliccare su "AVR GCC" per selezionarlo; in tal modo scegliamo avr-gcc come compilatore;
- in "Project name", immettere un nome a piacere che identifichi il progetto; tale nome viene automaticamente riportato sulla casella sottostante "Initial file", che diverrà il file sorgente principale;
- assicurarsi di spuntare (selezionare) entrambe le caselline "Create initial file" e "Create folder";
- assegnare nella casella "Location" (cliccando sul tastino coi puntini) una cartella sul disco del vostro PC, che conterrà tutte le sottocartelle relative ai progetti che creeremo; con le impostazioni di **Fig. 2**, verrà creata la sottocartella di progetto relativa a "PrimoProgetto" sotto la cartella C:\AVR, creata precedentemente.

Infine clicchiamo su "Finish" (non su "Next"): abbiamo creato ed aperto un nuovo progetto, chiamato "PrimoProgetto", sul quale possiamo finalmente lavorare andandoci a scrivere i nostri sorgenti. Il progetto si presenta con la videata di **Fig. 3**.
La finestra in basso, contrassegnata in figura come "Finestra messaggi", permette in realtà di mostrare diversi tipi di informazioni sulla base della sottostante linguetta selezionata:
- *Build*: mostra i risultati della compilazione;
- *Message*: riporta segnalazioni di AVR Studio;
- *Find in Files*: mostra i risultati delle ricerche di parole all'interno dei documenti;
- *Breakpoints and Tracepoints*: visualizza l'elenco dei punti omonimi, che si possono applicare ad un programma durante il debugging (ritorneremo su questo più avanti).

L'area sulla sinistra dell'ambiente AVR Studio

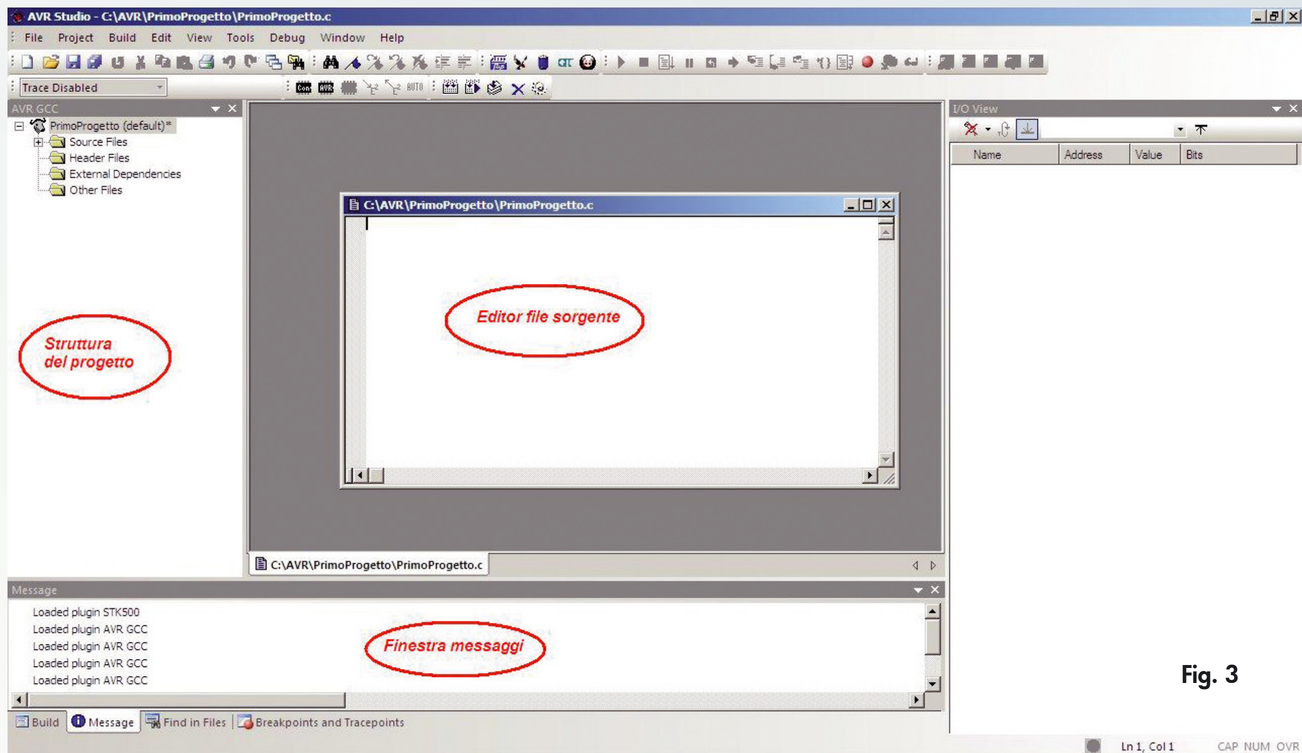
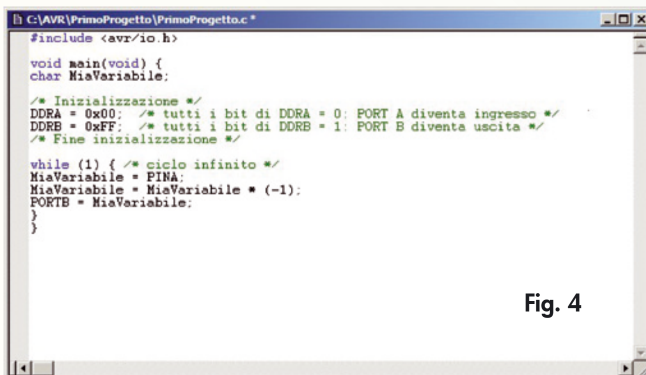


Fig. 3

permette di visualizzare tutti i file che compongono il progetto e di selezionarli per aprirli con l'editor al centro: cliccando sul segno '+' della cartella "Source file" compare il nome del nostro primo file, "PrimoProgetto.c", che è stato creato con il nuovo progetto. Cliccandovi due volte sopra, l'editor, di cui appare la finestra al centro dello schermo, lo apre; chiaramente la finestra è vuota perché ancora nel progetto non abbiamo scritto nulla!
Andiamo dunque a copiarvi il programmino di esempio della scorsa puntata, come riportato in **Fig. 4**.
Ricordate che nella stesura di programmi ad alto livello come il C, è buona norma indentare le righe (ovvero aumentarne il rientro, verso destra) all'interno del corpo delle funzioni (come la *main()*) e degli operatori (come il *while*). In seguito useremo sempre questa regola, che pur non cambiando il comportamento dei programmi li rende molto più leggibili, specie se complessi e con molte funzioni "nidificate" (cioè inserite una dentro l'altra). In **Fig. 5** è stata applicata l'indentazione al nostro programma, usando la spaziatura fornita dal tasto "tab" (corrispondente a quattro caratteri); fare ciò è più comodo che premere quattro volte consecutive la barra spazio! Per cercare di chiarire meglio il concetto, nella **Fig. 5** figura è stata disegnata una freccetta rossa in corrispondenza dell'applicazione di ogni "tab" (le frecce, chiaramente, non appaiono nell'editor del programma). A proposito dell'editor, avrete certamente notato che esso colora automaticamente le parole o le frasi che vi si inseriscono, a tutto vantaggio della comprensione. Ad esempio, i commenti (tutto il testo tra i caratteri "/*" e "*/") sono in ver-


```
#include <avr/io.h>

void main(void) {
char MiaVariabile;

/* Inizializzazione */
DDRA = 0x00;  /* tutti i bit di DDRA = 0: PORT A diventa ingresso */
DDRB = 0xFF;  /* tutti i bit di DDRB = 1: PORT B diventa uscita */
/* Fine inizializzazione */

while (1) { /* ciclo infinito */
MiaVariabile = PINA;
MiaVariabile = MiaVariabile * (-1);
PORTB = MiaVariabile;
}
}
```
Fig. 4

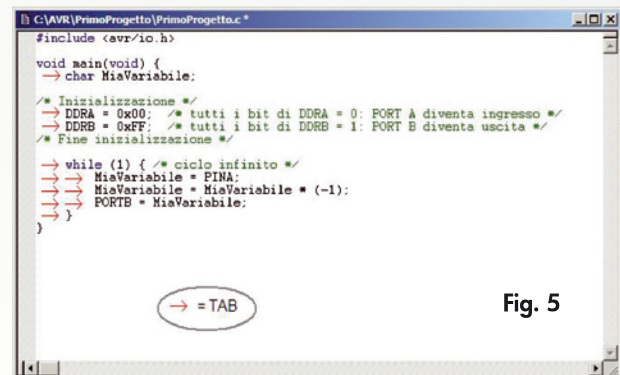
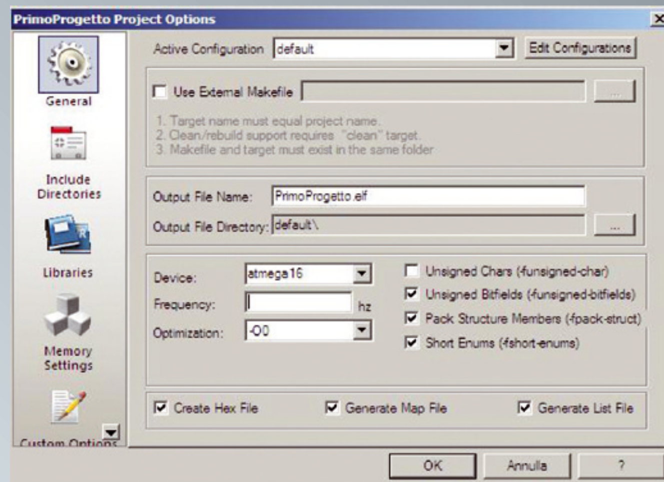

```
#include <avr/io.h>

void main(void) {
    char MiaVariabile;

/* Inizializzazione */
    DDRA = 0x00;  /* tutti i bit di DDRA = 0: PORT A diventa ingresso */
    DDRB = 0xFF;  /* tutti i bit di DDRB = 1: PORT B diventa uscita */
/* Fine inizializzazione */

    while (1) { /* ciclo infinito */
        MiaVariabile = PINA;
        MiaVariabile = MiaVariabile * (-1);
        PORTB = MiaVariabile;
    }
}
```

Fig. 5

Fig. 6



de, le parole chiave del C sono in blu (*while*, *void*, *char*, …), mentre tutto il testo non facente parte del linguaggio (come il nome delle variabili, che è inventato da noi...) viene lasciato in nero. Come si farebbe con qualsiasi editor di testo, bisogna infine preoccuparsi di salvare il contenuto del file, cliccando sull'icona del dischetto nella barra del menu; la scomparsa dell'asterisco a fianco del nome del file nella cornice superiore dell'editor conferma l'avvenuto salvataggio.

COMPILIAMO IL PROGRAMMA

Come già anticipato, l'AVR Studio può essere integrato con il compilatore avr-gcc per la compilazione diretta di programmi scritti in C per il nostro microcontrollore ATMEGA; per abilitare questa funzione nel progetto su cui lavoriamo bisogna solo verificare alcune impostazioni, alle quali accediamo aprendo la tendina della voce "Project" del menu, e selezionando quindi "Configuration options": dovrebbe aprirsi la finestra di **Fig. 6**.
Impostate le voci come mostrato in figura, e in particolare selezionate "atmega16" nella tendina "Device" e "-O0" nella tendina "Optimization". In seguito, cliccate sull'icona in basso a sinistra, che mostra un foglio e una matita, per aprire nella finestra le "Custom options", e verificate che tutte le voci siano impostate come in **Fig. 7**. In particolare, verificate che le voci nel riquadro "External Tools" facciano riferimento alle locazioni in cui avevate installato il pacchetto *WinAVR*. A questo punto potete cliccare sul tasto "OK" per chiudere la finestra. Queste impostazioni vanno verificate ed eventualmente modificate ogni volta che si crea un nuovo progetto, ed eventualmente "in corso d'opera" per variare qualche direttiva del compilatore.
Possiamo finalmente lanciare la prima compilazione vera e propria del nostro programma, selezionando la scelta "Build" nella tendina che si apre cliccando nella voce "Build" da menu, oppure premendo il tasto F7.
La finestra inferiore dell'ambiente (la "Finestra messaggi" in **Fig. 3**) commuta automaticamente sulla voce "Build" per mostrare le fasi del processo di compilazione e soprattutto il loro esito, ovvero informa se il processo è andato a buon fine o meno.
Andiamo, allora, ad analizzare quella che, se

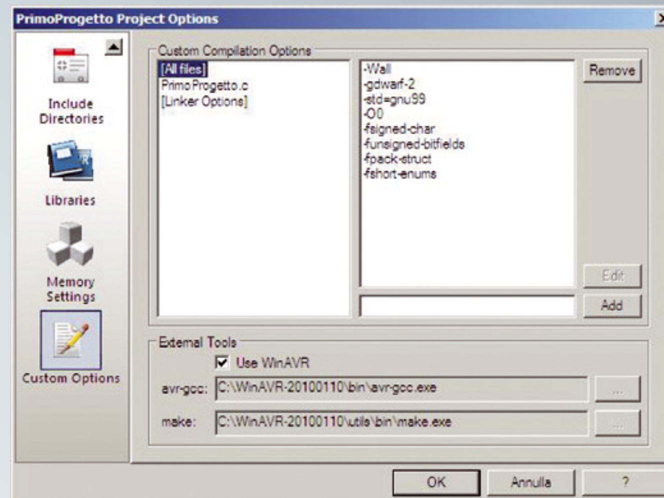


Fig. 7

tutto è andato bene, dovrebbe risultare uguale a quella riportata in **Fig. 8**.
Partendo dall'inizio, si possono notare varie righe, ognuna corrispondente ad uno dei processi di compilazione (oppure ad un messaggio) precedute da un pallino colorato in base all'esito dell'operazione: verde se è andato tutto bene, giallo se c'è qualche messaggio di avvertimento (*warning*), rosso se il processo non può essere concluso a causa di qualche errore.
Analizzando la **Fig. 8**, si nota che il primo processo avviato è (tramite il comando omonimo) il compilatore avr-gcc, seguito da una lista di parametri (facilmente identificabili nella riga in quanto sono preceduti dal segno meno) che derivano dalle impostazioni da noi scelte nella finestra "Configuration options" di **Fig. 6**. Ad esempio, il primo parametro è:

```
-mmcu=atmega16
```

con il quale si informa il compilatore che dev'essere generato codice per l'ATmega16.

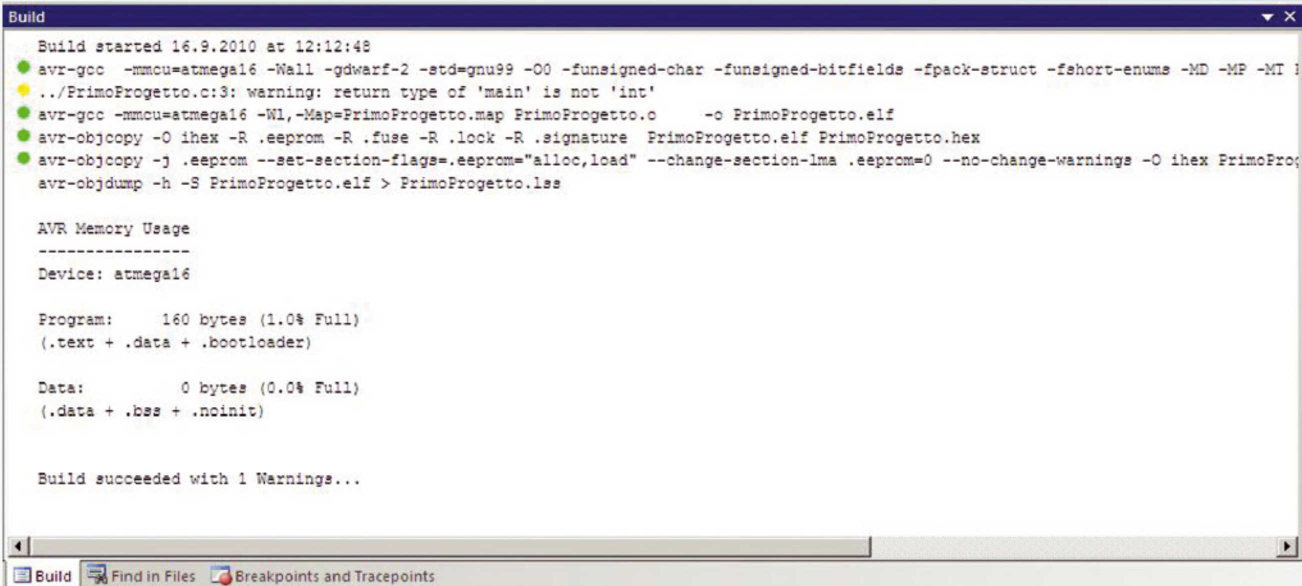
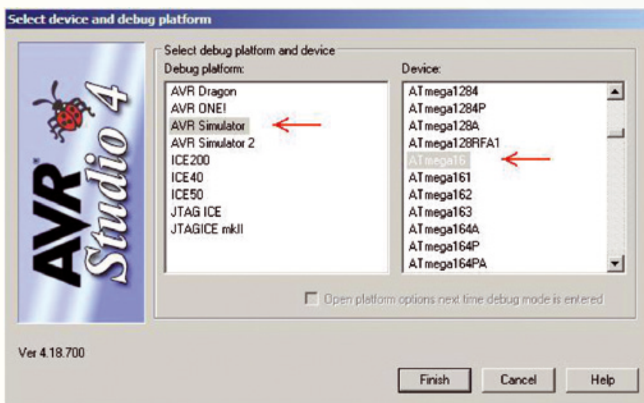


Fig. 8

La riga successiva è preceduta da un pallino giallo, che identifica un messaggio di “warning” generato dal comando precedentemente eseguito, in questo caso dall’avr-gcc (quindi dal processo di compilazione): il compilatore ci informa che nel file “PrimoProgetto.c” alla riga 3, la funzione **main()** non ritorna alcun valore (quando termina) in quanto nella dichiarazione vi è la parola chiave `void` mentre, “per definizione” della funzione **main()** in linguaggio C, il compilatore si aspetterebbe che ritornasse un numero intero (`int`). Per quanto visto la scorsa puntata, questo fatto è ininfluente ai fini del funzionamento del programma, in quanto la funzione **main()**, in programmi per microcontrollori, non deve mai terminare.
Pertanto in questo caso possiamo tranquillamente ignorare l’avvertimento. Alternativamente, potreste verificare che sostituendo `void` con `int` , ovvero scrivendo:

```
int main(void)
```

Fig. 9



e ricompilando il tutto, il messaggio di warning scomparirebbe.
Quando si compilano programmi più complessi è comunque normale che vengano generati diversi warning; sebbene essi non impediscano al compilatore di portare a termine il suo lavoro (in quanto non sono errori veri e propri) il consiglio è quello di controllarli uno ad uno per verificare che non vi sia effettivamente “da ritoccare” qualcosa nel sorgente.
Se invece il compilatore rileva degli errori, essi vanno necessariamente corretti in quanto in caso contrario la fase di compilazione non potrà terminare.
Un tipico errore è quello di digitazione: se al posto di ‘`DDRA`’ scriviamo ‘`DDRAS`’, all’atto della compilazione vedremo comparire nella finestra di Build tre righe di errore:

```
../PrimoProgetto.c:7: error: ‘DDRAS’ undeclared
(first use in this function)
../PrimoProgetto.c:12: error: (Each undeclared
identifier is reported only once
../PrimoProgetto.c:12: error: for each function
it appears in.)
```

Sebbene ad un primo sguardo diano l’impressione che mezzo programma sia sbagliato, esse stanno solo ad indicarci che la variabile ‘`DDRAS`’non esiste; trattandosi di un errore, le righe sono precedute dal bollino rosso ed il codice oggetto non viene ovviamente creato, ed infatti l’ultima riga della finestra di Build riporta l’infausto messaggio:

```
Build failed with 3 errors and 1 warnings.
```

dove i tre errori sono riferiti alle altrettante

righe “a bollino rosso” che appaiono, mentre il warning è sempre quello analizzato prima. Si noti infine che facendo doppio clic sulle righe di warning o di errore della finestra di Build, nella finestra del sorgente il cursore si posiziona sulla riga ‘incriminata’.
Proseguendo nell’analisi della finestra “Build” di **Fig. 8** troviamo una seconda invocazione del programma avr-gcc, ma questa volta con la funzione di *linker* per “mettere insieme” al codice generato dal nostro programma eventuali librerie aggiuntive richieste.
Anche in questa fase potrebbero essere generati messaggi di errore: ad esempio, se le librerie da includere non esistessero.
Seguono due chiamate a “avr-objcopy”, un programma di utilità (fornito nel pacchetto WinAVR) per creare, leggendo il file oggetto creato dalle precedenti fasi di compilazione e linking, altri file con formati specifici per essere utilizzati ad esempio dai programmatori di microcontrollori (uno dei formati più diffusi a tal scopo è il cosiddetto “*Intel Hex*”, i cui file hanno estensione “.hex”).
L’ultimo comando eseguito è “avr-objdump”, che con i parametri impostati crea un file cosiddetto di *listing*, chiamato “PrimoProgetto.lss”; questo documento, che è un normale file di testo, mostra nel dettaglio la traduzione del codice C di ogni riga del programma sorgente nel linguaggio macchina del microcontrollore, e mostra pure i corrispondenti indirizzi di memoria in cui viene caricato.
Tenete presente che generalmente non è necessario consultare tale documento; la consultazione può servire solo per scopi particolari: ad esempio, se bisogna, per qualche motivo, scoprire qual è la locazione fisica di una variabile o di una certa funzione.

SIMULIAMO IL PROGRAMMA

Spesso e specialmente quando si scrivono i primi programmi, è utile avere la possibilità di verificarne “virtualmente” il funzionamento con il PC, ovvero sapere come si comporta senza dover programmare fisicamente un microcontrollore per farlo funzionare su una scheda reale; questa procedura è chiamata *simulazione* del microcontrollore.
AVR Studio possiede un *debugger*, ovvero le funzionalità che consentono (tra le altre cose) di simulare via software i microcontrollori At-

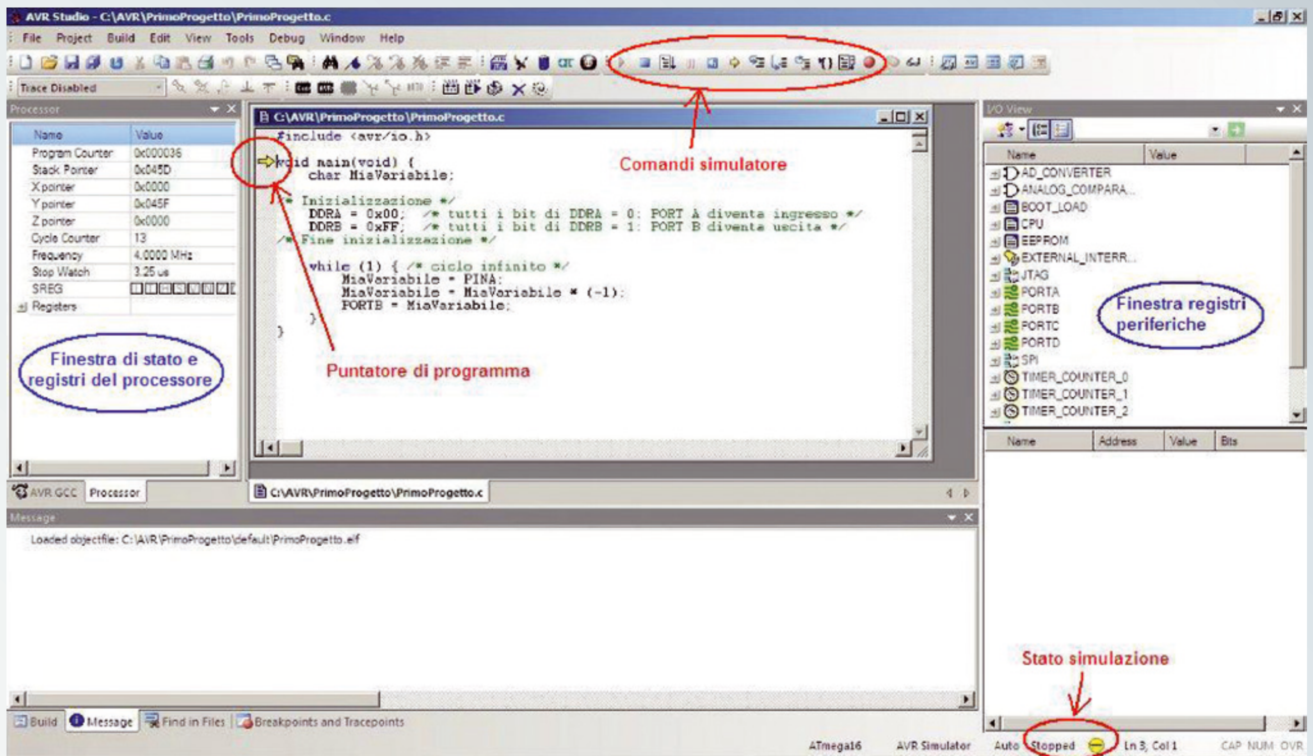


Fig. 10 - AVR Studio nella modalità di debug.

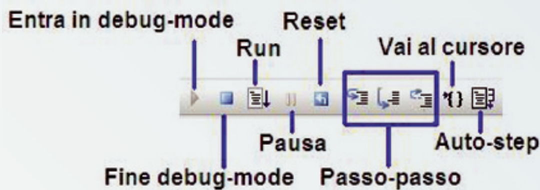


Fig. 11 - Comandi principali del debugger.

mel, pertanto potremo verificare il funzionamento del nostro primo programmino senza dover disporre di un hardware reale.
Per attivare la funzione di simulazione bisogna selezionare da menu le voci *Debug→Select Platform and Device...* ; compare la finestra riportata in **Fig. 9**. Selezionate quindi le voci dei riquadri "Debug platform" e "Device", come suggerito dalle freccette rosse in figura, ed infine cliccate su "Finish" per chiudere la finestra. In tal modo il *debugger* di AVR Studio è impostato per funzionare in modalità simulatore, ovvero senza dispositivi esterni; per avviarlo, cliccate sulla freccia verde nella barra degli strumenti, oppure selezionare da menu: *Debug→Start Debugging*.
Si può notare che nella IDE di AVR Studio compaiono nuove finestre, come mostrato in **Fig. 10**. Innanzitutto si può notare, in basso a destra, l'indicazione "stopped" (accompagnata dal pallino giallo): essa segnala che il simulatore è fermo sull'istruzione indicata dalla freccia gialla osservabile a sinistra della finestra del nostro sorgente, ed indicata come "puntatore di programma" (questo indica qual è la prima istruzione che sarà eseguita quando il simulatore viene avviato). Nel caso della **Fig. 10**, punta alla funzione **main()**, che è l'inizio del programma.
Sulla sinistra vi è un menu a tendina chiamato "*Processor*", in cui si possono vedere ed impostare i registri della CPU. Sulla destra dello schermo troviamo la tendina "*I/O view*", nella quale si accede ai registri delle periferiche (porte di I/O, timer, UART, ecc.), per leggere e/o immettervi un valore. A tale riguardo, si ricordi che i valori dei registri del processore, della memoria e delle periferiche, possono essere modificati solo quando l'emulatore è fermo (stato "stopped"). Per eseguire la simulazione vi sono i due metodi seguenti.

- Funzionamento normale o continuo (*Run*): il processore virtuale entra in funzione e non si arresta sin quando l'utente non interviene sul tasto di *Stop*. Durante il funzionamento, l'indicazione di stato del simulatore è "Running" affiancato da un pallino verde; durante questo stato non si possono modificare manualmente i registri del processore, della memoria o delle periferiche.
- Esecuzione passo-passo (*Step by step*) in cui le istruzioni del programma vengono eseguite una per volta: l'utente deve immettere manualmente il comando di "Step", in risposta al quale il processore esegue una singola istruzione del programma e poi si ferma, rimanendo in attesa di un nuovo comando. Usando questo metodo si possono modificare i valori dei registri.

In AVR Studio, sulla barra degli strumenti, compaiono in modalità debugger dei pulsanti relativi a queste funzioni, mostrati nella **Fig. 11**. I primi due comandi servono ad entrare ed uscire dalla modalità di simulazione (*debug mode*, nella quale sono attivati i comandi di simulazione di seguito descritti); il pulsante "Run" avvia la simulazione in modalità continua, che può essere fermata premendo il tasto di pausa oppure quello di reset: il primo blocca il programma dal punto in cui è arrivato (segnalato dal puntatore di programma), e da lì riparte se in seguito si preme nuovamente il "Run".
Il secondo pulsante (Reset) azzera il processore e lo riporta nella condizione di "accensione"; al prossimo comando di "Run" l'esecuzione ripartirà dall'inizio del programma (ovvero la funzione **main()**).
I successivi tre tasti riguardano l'esecuzione passo-passo: si differenziano per la possibilità di entrare e uscire dalle funzioni del programma, come chiariremo meglio in seguito. Il comando "Vai al cursore" (*Run to cursor*) permette di avviare la modalità continua di esecuzione sino al punto del programma in cui si trova il cursore, precedentemente posizionato con il mouse o le freccette della tastiera, mentre "**Auto step**" avvia la modalità passo-passo in automatico: i vari step sono eseguiti senza l'intervento dell'utente, in una sorta di modalità continua di esecuzione.
A differenza di quest'ultima, però, aggiorna la visualizzazione dei registri delle rispettive tendine (del processore e delle periferiche).
Passiamo ora alla pratica e proviamo le modalità di simulazione appena descritte con il

programmino esemplificativo che abbiamo creato; entrando in modalità di simulazione oppure, dall'interno di essa, facendo clic sul pulsante "Reset" (vedi **Fig. 11**), ci si porta nella situazione di **Fig. 10**, ossia: simulatore fermo ("Stop") e puntatore di programma posizionato sulla funzione **main()**.
Per avviare il programma clicchiamo sul tasto "Run": il puntatore di programma (cioè la freccetta gialla) sparisce, in quanto il "processore virtuale" del simulatore non è più fermo su un'istruzione ma sta velocemente eseguendo l'intero programma, che, come già sappiamo, si richiude all'infinito dentro al ciclo "`while()`".
Notare che la scritta in basso a destra nella finestra di AVR Studio è cambiata da "*Stopped*" a "*Running*" ed è comparso un pallino verde col segno '+' al posto di quello giallo col '-'. Ora fermiamo il programma cliccando sul tasto "Pausa": nell'editor del sorgente, il puntatore di programma ricomparirà sull'istruzione che il processore stava per eseguire quando noi l'abbiamo fermato.
Per provare la modalità di esecuzione passo-passo clicchiamo sul comando "*Step Into*", ossia quello più a sinistra dei tre del gruppo "passo-passo" visibili in **Fig. 11**. Si può notare che ad ogni attivazione del comando, il puntatore di programma si sposta sull'istruzione successiva; superata la riga "`PORTB=MiaVariabile`;" il ciclo "`while()`" è terminato e il puntatore ritorna alla prima istruzione del ciclo, ovvero "`MiaVariabile=PINA`;" e così via. Se si vuole far ripartire il programma dall'inizio (posizionando il puntatore sulla riga del main) bisogna cliccare sul tasto "*Reset*".
A questo punto sappiamo come eseguire una simulazione del programma da un microcontrollore "virtuale" realizzato dal software di AVR Studio. La simulazione, peraltro, dovrebbe dare una risposta alla domanda: il programma *funziona*? Per scoprirlo bisogna prestare attenzione alle variabili che vengono gestite dal programma e verificare se, durante la simulazione, vengono modificate come ci si attende; questa verifica va eseguita, almeno in prima battuta, per ciascuna riga di codice.
Resettiamo dunque il simulatore (cliccando sul tasto di "Reset") per far tornare il puntatore di programma all'inizio del **main()**.

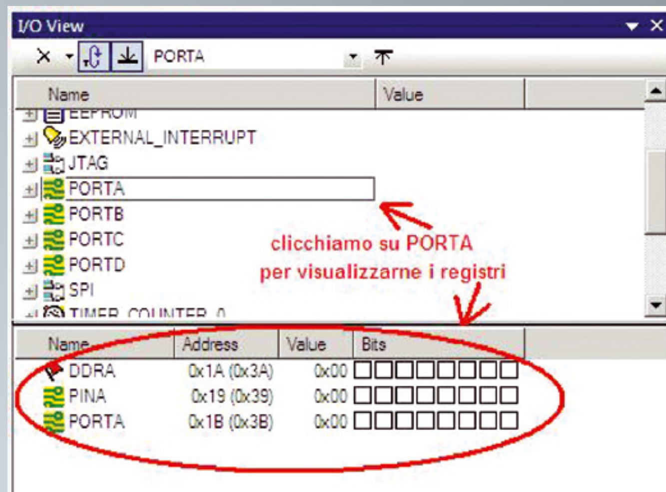


Fig. 12

Poiché la prima riga di codice del nostro programma inizializza il registro DDRA a 0x00, visualizziamone il valore 'reale' aprendo il menu tendina "I/O View" e cliccando su PORTA: nella finestra sottostante compaiono i tre registri relativi a tale periferica (**Fig. 12**), tra cui DDRA, che vediamo valere già 0x00 (come pure gli altri due), in quanto tale valore viene impostato dal microcontrollore in automatico quando viene *resettato* (prima della partenza del programma).
Per questa ragione ci si deve aspettare che l'esecuzione della prima istruzione non ne modifichi il valore.
Clicchiamo sul comando "*Step Into*" e verifichiamolo: il puntatore di programma si sposta dalla riga del `main()` alla riga `DDRB = 0xFF`, il che significa che la CPU virtuale ha superato (ed eseguito) la prima istruzione, anche se non "si nota" in quanto DDRA continua a valere 0x00.
Adesso, nel menu a tendina "I/O View" clicchiamo su PORTB per visualizzarne i registri: vedremo che anch'essi valgono tutti 0x00.
Eseguiamo ora un comando "*Step Into*": il puntatore si sposta all'istruzione successiva e la riga `DDRB=0xFF` viene eseguita: nel menu a tendina dei registri si nota che DDRB è passato da 0x00 a 0xFF ed è stato anche colorato in rosso per evidenziare che il valore è cambiato rispetto al passo precedente.
Il puntatore di programma si trova, adesso, sulla prima istruzione all'interno del ciclo **while()**, segnando che tale riga è quella che sarà eseguita alla prossima attivazione del comando "*Step Into*"; per verificarlo dobbiamo visualizzare il valore di "MiaVariabile" prima e dopo l'esecuzione dell'istruzione.
Portiamoci col cursore del mouse sopra al suo nome, clicchiamo col pulsante destro e, nel menu a tendina che appare, selezioniamo

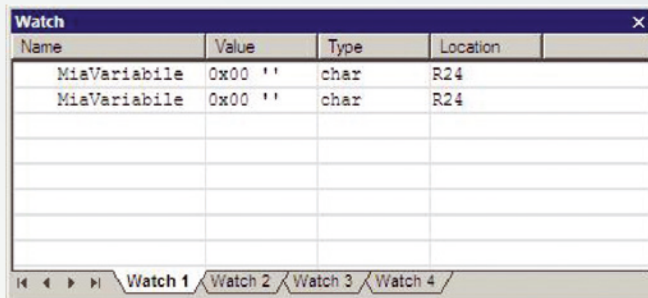


Fig. 13 - Finestra di "Watch".

la voce "Add watch: "MiaVariabile""; allora compare una piccola finestra (detta di *Watch*, cioè osservazione) visibile in **Fig. 13**, col valore di questa variabile, che può essere un numero qualsiasi in quanto non è ancora stata inizializzata con un valore preciso.
Adesso andiamo nel menu a tendina "I/O View" e clicchiamo su PORTA per visualizzarne nuovamente i registri; nel campo "Value" del registro PINA facciamo doppio click, e modifichiamone il valore da 0x00 a 0x55.
Eseguiamo quindi la riga "`MiaVariabile=PINA;`", in cui si trova il puntatore di programma, con il comando "*Step Into*"; nella finestra "*Watch*" si potrà vedere che la variabile "MiaVariabile" ha anch'essa assunto il valore 0x55.
Notate che invece di questo valore potrebbe apparire 85; ciò accadrebbe se la visualizzazione fosse in decimale; in tal caso, per vedere i valori esadecimali bisogna cliccare col tasto destro nella finestra e selezionare "Display all values as Hex".
Si ricordi che PINA contiene, in realtà, il valore relativo allo stato dei piedini *fisici* del PORTA del microcontrollore; però siccome stiamo facendo una simulazione virtuale (il microcontrollore non esiste fisicamente), dobbiamo inserire manualmente un numero a nostro piacimento.
L'istruzione successiva fa l'inversione di segno: con il consueto "*Step Into*" la eseguiamo, ed il valore di MiaVariabile diviene 0xAB che è esattamente il complemento a due di 0x55.
Con un ulteriore "*Step Into*" eseguiamo quindi anche l'ultima operazione, che è il trasferimento del valore in uscita sul PORTB: per verificarlo clicchiamo, nella tendina "I/O View", su PORTB per far comparire nella finestra sottostante i tre registri relativi e vedere che il valore del registro PORTB è effettivamente divenuto 0xAB (a questo punto, in un sistema reale i piedini del microcontrollore del PORTB riporterebbero tale valore).

Poiché il ciclo **while()** è concluso, il puntatore di programma si è riportato sulla prima istruzione del ciclo; a questo punto si può provare a modificare a piacere il valore di PINA per verificare che, alla fine del ciclo, PORTB abbia sempre il complemento a due del valore inserito.
Bene, con questo vi abbiamo introdotto il funzionamento del debugger di AVR Studio nella modalità di simulazione, che ci permette di vedere direttamente su PC come si comporterebbe il microcontrollore durante l'esecuzione delle varie istruzioni, consentendoci altresì di capire se il programma che sviluppiamo funzionerebbe, poi, anche in un sistema reale.
Naturalmente questo comporta diverse limitazioni: nella nostra applicazione, il valore di ingresso (che nella realtà sarebbe lo stato fisico dei bit sui piedini del PORTA del microcontrollore) dev'essere immesso manualmente tramite tastiera in PINA, ma questo si può fare solo se il simulatore non è in modalità "Run"; altrimenti va fermato, oppure bisogna optare per un'esecuzione passo-passo, la quale, pur essendo interessante in quanto permette di vedere il cambiamento di registri e variabili in concomitanza dell'esecuzione delle istruzioni, non permette di eseguire il programma alla sua velocità reale (pochi microsecondi per ciascuna).
L'esecuzione passo passo è da considerare, in ogni caso, un valido strumento, soprattutto a livello "didattico".
Arrivati alla conclusione di questa puntata vi lasciamo con il suggerimento di scrivere qualche programmino che (sulla falsa riga di quello proposto in queste pagine) esegua operazioni tra variabili, per poi simularlo e verificare che le istruzioni vengano eseguite correttamente.
Nella prossima puntata passeremo alla pratica, presentandovi una scheda di sviluppo che ci consentirà di provare i nostri futuri programmi su un microcontrollore reale.

Riferimenti

[1] *http://sourceforge.net/projects/winavr/files/*
[2] *http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725*

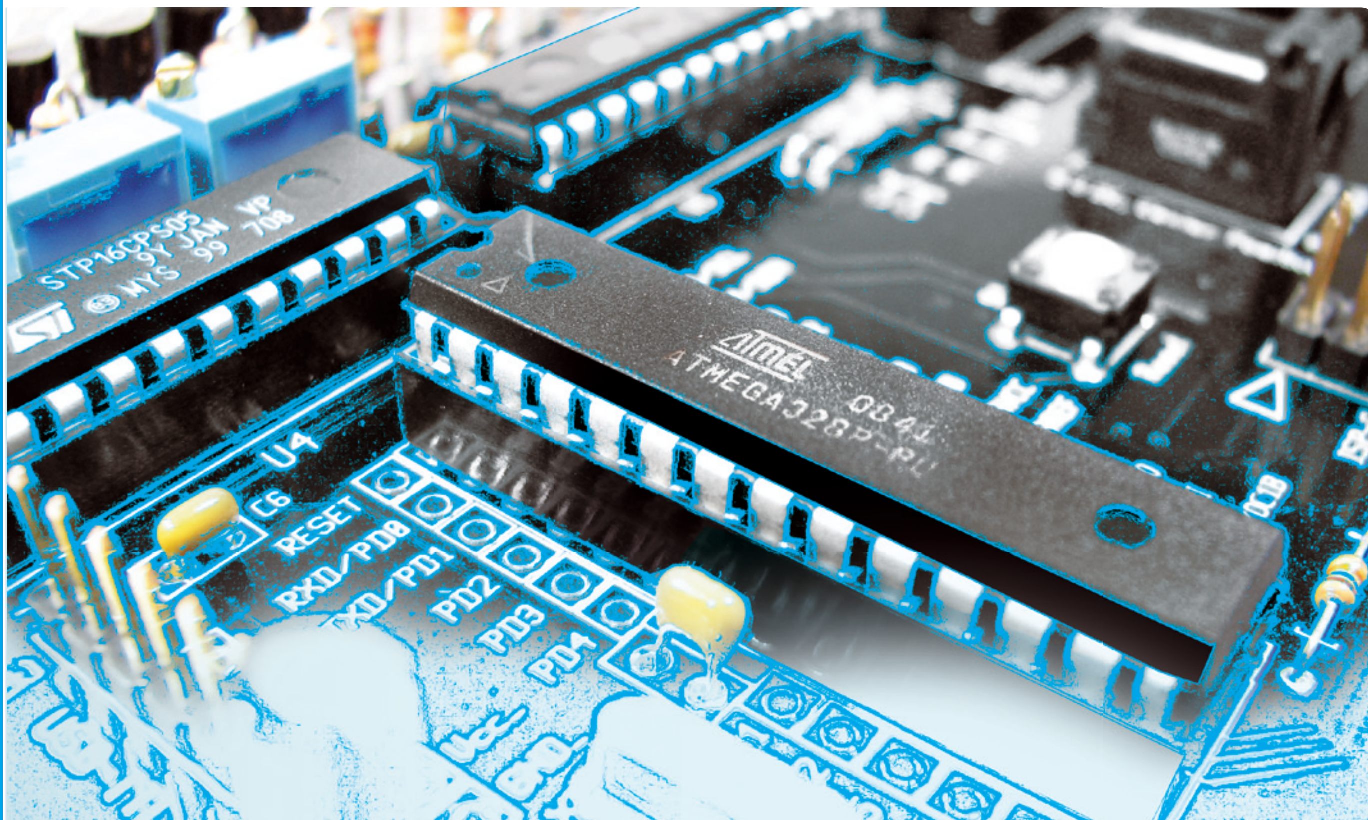
differenza è la volatilità, intendendo con ciò che quando si toglie alimentazione al micro, la memoria perde tutti i dati.
Nel nostro micro la RAM complessiva è composta da 1.120 locazioni da 8 bit ciascuna (quindi 1.120 byte), ed è suddivisa in:

- 1.024 byte per utilizzo generico (chiamata "internal data SRAM memory");
- 32 byte (chiamati *registri* e numerati da R0 a R31) di cui si avvale la CPU per la memorizzazione temporanea di dati ed operandi;
- 64 byte per la gestione delle periferiche, anche questi chiamati *registri (I/O registers* per la precisione) e utilizzati per caricare i valori che impostano il funzionamento delle periferiche del micro, oppure per riportare alla CPU il loro stato (ad esempio la lettura di un port configurato come input).

EEPROM: è una memoria dati ma non volatile, dunque a differenza della RAM mantiene i dati memorizzati anche in assenza di alimentazione; tuttavia la CPU non può accedervi per leggere istruzioni (quindi non è utilizzabile come memoria di programma). Inoltre le procedure di lettura e scrittura non sono "immediate" ma richiedono sequenze di istruzioni ben precise, come vedremo più avanti. Va comunque ricordato che per l'esecuzione di un programma le memorie essenziali sono la Flash (per le istruzioni) e la RAM (per i dati temporanei): la EEPROM è una cosa "in più" che il nostro micro possiede e che può tornare utile per salvare particolari dati che non si vuole vadano persi spegnendo il sistema.
PORT A, B, C, D: il nostro microcontrollore dispone di quattro gruppi di piedini di ingresso/uscita (i PORT appunto) da 8 bit ciascuno; la CPU può accedervi specificando un determinato indirizzo (ogni port ha il proprio) per "comunicare" con l'esterno. Quasi tutti i pin dei port vengono inoltre condivisi con altre periferiche identificate nello schema come *speciali*, ovvero che svolgono funzioni particolari sui segnali in arrivo o verso il mondo esterno. Ogni microcontrollore, oltre ai classici I/O port paralleli, è equipaggiato con un proprio set di periferiche: di seguito elenchiamo brevemente quelle contenute nel microcontrollore ATmega16.

- **Convertitore Analogico/Digitale** (ADC): permette di leggere una tensione continua analogica da alcuni pin (per esempio proveniente da un sensore) e convertirla in un numero binario interpretabile dalla CPU.
- **Timer/Counter**: vi sono 3 moduli indipendenti, utilizzabili come contatori di eventi o temporizzatori.
- **Oscillatore**: fornisce le temporizzazioni alla CPU ed alle periferiche (come i timer); può essere programmato per funzionare autonomamente (senza componenti aggiuntivi), prelevare il segnale da un pin o anche essere collegato ad un quarzo esterno per aumentare la precisione delle temporizzazioni (cosa utile, ad esempio, se si vuol fare un orologio digitale).
- USART: interfaccia per convertire i dati paralleli (come i byte, con cui lavora la CPU) in dati seriali (un bit per volta) utilizzati per colloquiare con alcuni dispositivi esterni con interfacciamento seriale (come mouse, memorie, modem).
- TWI: altra interfaccia seriale, che però supporta un altro protocollo (chiamato I^2C) per il colloquio con dispositivi che lo utilizzano.
- SPI: interfaccia seriale sincrona, utilizzabile, tra l'altro, per programmare il microcontrollore.
- JTAG: interfaccia seriale utile per programmare ed eseguire il debug dei programmi.
- Comparatore analogico: confronta due tensioni V_1 e V_2 applicate su altrettanti pin ed ha in uscita un bit (collegabile anch'esso ad un pin e comunque leggibile dalla CPU) che indica quale delle due tensioni è maggiore (bit a 1 quando $V_1 > V_2$; bit a 0 quando $V_1 < V_2$).

Nel proseguimento del corso vedremo anche come mettere in funzione alcune di queste periferiche, ma per fare ciò è necessario imparare prima a programmare il microcontrollore, ovvero istruirlo ad eseguire le operazioni che desideriamo.
Arrivederci, dunque, alla prossima puntata, il cui scopo sarà proprio quello di fornire le basi per poter scrivere il nostro primo programma per l'ATmega16.



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

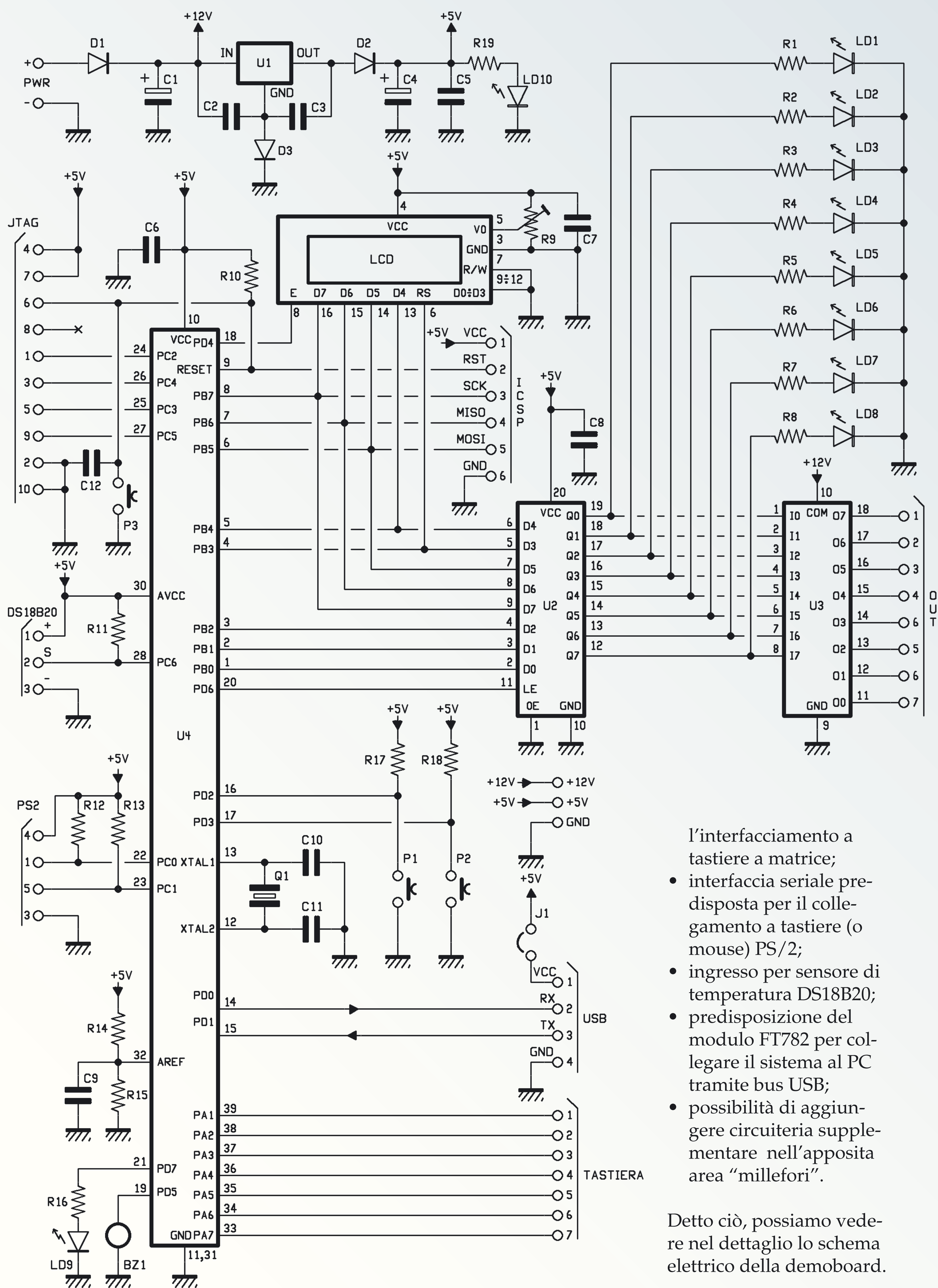
Realizziamo gli strumenti per lavorare praticamente con il microcontrollore ATMEGA16: i sistemi per programmarlo ed una scheda di sviluppo in cui inserirlo per testarne alcune applicazioni. Quarta puntata.

Dopo aver esaminato i concetti e gli strumenti necessari a sviluppare e simulare programmi per microcontrollore, è finalmente giunto il momento di passare alla pratica; in queste pagine, infatti, vi descriveremo e proporremo la realizzazione di una scheda di sviluppo con la quale vedremo "vivere" realmente i nostri programmi!

DEMOBOARD PER ATMEGA16

Questa scheda è stata pensata per ospitare il microcontrollore ATMEGA16 e fungere da supporto hardware ai programmi che proporremo durante questo corso, in modo da consentirvi di sperimentare il funzionamento di diverse periferiche del micro e di dispositivi esterni (display, tastiere, sensori, ecc.); si tratta pertanto di una scheda dimostrativa (che dunque chiameremo nel seguito *demoboard*), dal cui schema si potrà prendere spunto per progettarne una personale oppure per apportare ad essa eventuali modifiche finalizzate ad adattarla alle proprie esigenze. La "dotazione" della scheda comprende:

- dispositivi di visualizzazione (nove LED ed un display LCD alfanumerico da 16 caratteri su due righe);
- 8 uscite open-collector idonee al pilotaggio di carichi resistivi o induttivi, come i relé;
- 7 ingressi analogici/digitali predisposti per

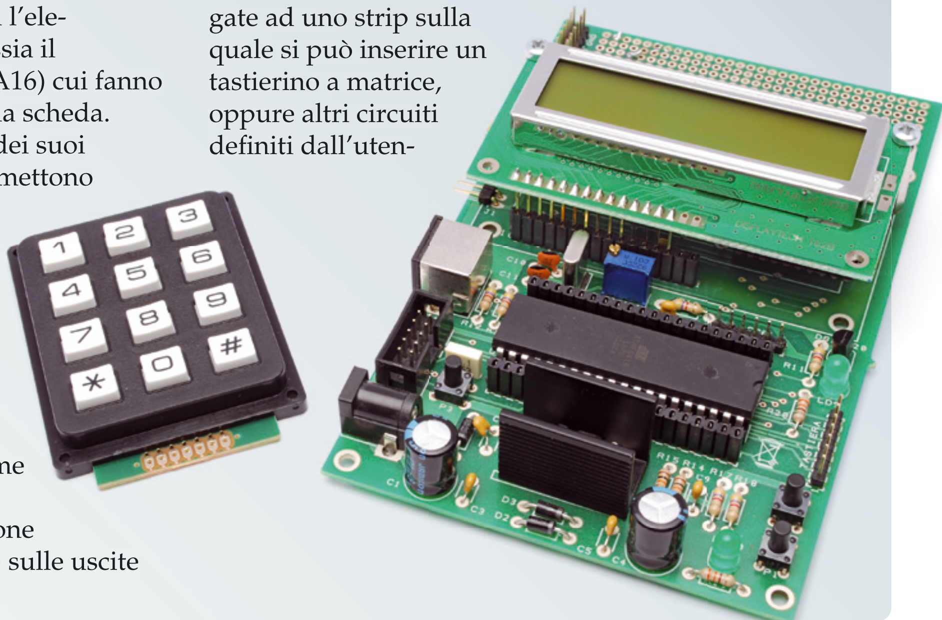


- l’interfacciamento a tastiere a matrice;
- interfaccia seriale predisposta per il collegamento a tastiere (o mouse) PS/2;
- ingresso per sensore di temperatura DS18B20;
- predisposizione del modulo FT782 per collegare il sistema al PC tramite bus USB;
- possibilità di aggiungere circuiteria supplementare nell’apposita area “millefori”.

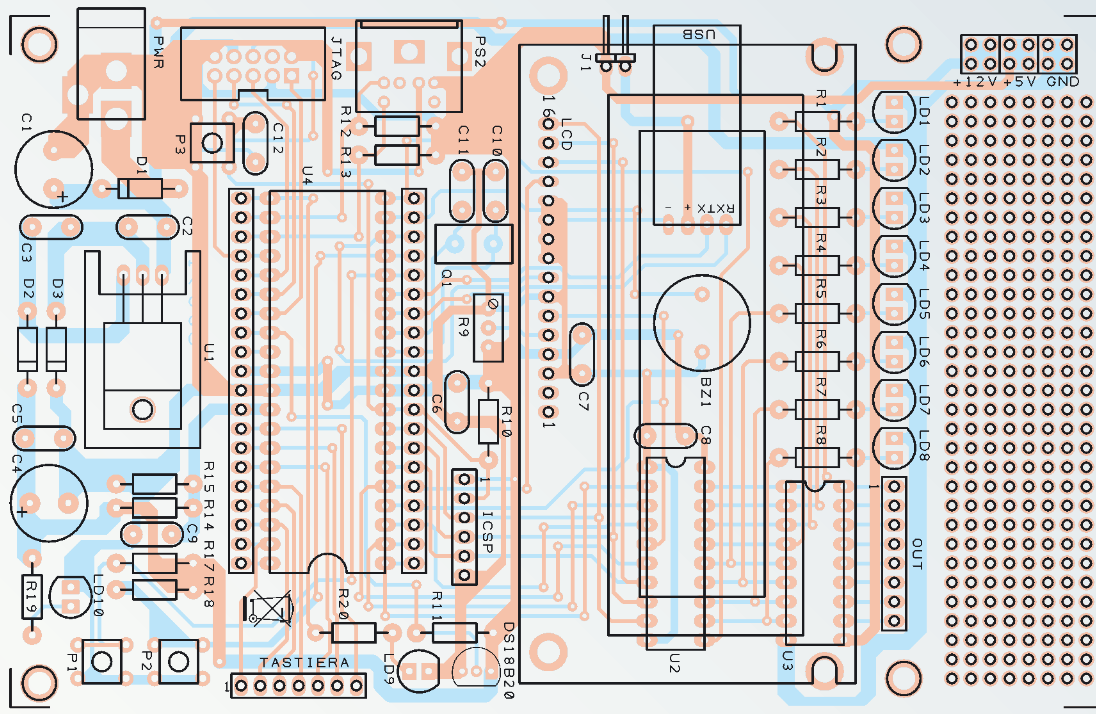
Detto ciò, possiamo vedere nel dettaglio lo schema elettrico della demoboard.

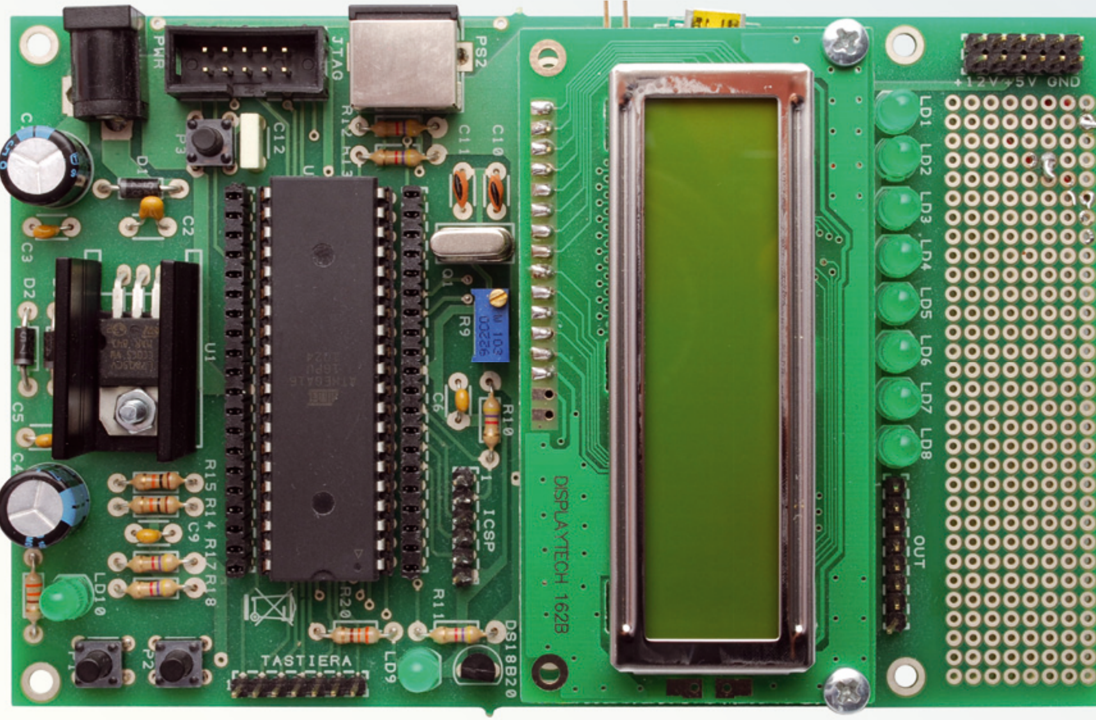
SCHEMA ELETTRICO

La demoboard è costituita da un singolo circuito stampato, il cui schema è visibile in queste pagine. In alto a sinistra abbiamo la sezione di alimentazione: dal connettore plug siglato PWR possiamo alimentare la scheda fornendole una tensione continua compresa tra 8 e 15 volt, anche non stabilizzata, prelevabile da qualsiasi alimentatore da banco o da un adattatore AC/DC a spina, che sia provvisto di cavo di uscita terminante con uno spinotto plug; per quanto riguarda la potenza, una decina di watt sarà sufficiente per la maggior parte delle applicazioni.
Tornando allo schema, il diodo D1 serve da protezione contro involontarie inversioni di polarità, mentre C1 costituisce il filtro d'ingresso; U1 è l'integrato stabilizzatore a 5 V necessario per alimentare il microcontrollore e la logica che si trova collegata al segnale "+5V". Il diodo in serie D2 ci consente di applicare direttamente una tensione esterna di 5 V su tale linea (può accadere collegando alcuni programmatori che forniscono anche l'alimentazione) senza che l'integrato U1 venga danneggiato; notate, però, che D2 introduce una caduta di circa 0,6 V, il piedino *GND* di U1 è stato collegato a massa tramite il diodo D3, in modo da innalzare il valore della tensione dell'uscita *OUT* ad un livello di 5,6V e compensare dunque la caduta di D2. C2 e C3 prevengono auto-oscillazioni ad alta frequenza di U1, ed il LED LD10 indica semplicemente la presenza di tensione nel circuito.
Scendendo nello schema, si nota l'elemento principale del circuito, ossia il microcontrollore U4 (l'ATMEGA16) cui fanno capo tutte le restanti sezioni della scheda.
Il display LCD è collegato a sei dei suoi piedini, attraverso i quali si trasmettono dati e comandi per ottenere la visualizzazione dei caratteri; a tali piedini è anche collegato il connettore a strip ICSP, cui ci si può collegare con un programmatore per la programmazione "In-Circuit" del microcontrollore (spiegheremo più avanti, in queste pagine, come fare). U2 contiene otto flip-flop che consentono la memorizzazione dello stato degli ingressi D<7:0> sulle uscite Q<7:0>, quando viene attivata la linea *Latch Enable* (LE). Le uscite fanno capo ad otto LED che consentono la visualizzazione del loro stato logico; sono inoltre collegate ad U3, un Darlington array che, portando a massa ciascuna delle proprie uscite quando l'ingresso corrispondente va ad 1 logico, può essere utilizzato per pilotare direttamente piccoli carichi resistivi o induttivi come le bobine di relé a bassa tensione di pilotaggio (5÷12 V).
P1 e P2 sono due pulsanti facenti capo ai pin di I/O PD2 e PD3 del micro, mentre P3 permette di attivare il segnale di reset (che l'U4 legge dal piedino 9), forzando il programma del microcontrollore a ripartire dall'inizio senza dover togliere alimentazione alla scheda. Q1 è un quarzo che, insieme a C10 e C11, forma un oscillatore col quale il micro può ricavare le temporizzazioni necessarie al suo funzionamento; va detto che l'ATMEGA16 integra un oscillatore interno che potrebbe fare a meno del quarzo, ma la precisione di cui è capace è inferiore a quella permessa dal quarzo, indispensabile in applicazioni in cui il tempo va misurato con precisione, come ad esempio un orologio. Sotto al quarzo si notano le due linee di comunicazione seriale TX e RX, facenti capo ai GPIO PD0 e PD1 che possono essere collegate, ad esempio, ad un'interfaccia seriale/USB (tipo il modulo FT782 distribuito da Futura Elettronica) per realizzare la connessione con un PC; di questo riparleremo prossimamente. Più in basso ancora, troviamo le linee del PORTA collegate ad uno strip sulla quale si può inserire un tastierino a matrice, oppure altri circuiti definiti dall'uten-



[piano di MONTAGGIO]





Elenco Componenti:

R1÷R8: 330 ohm
R9: Trimmer multigiro 10 kohm
R10÷R13: 4,7 kohm
R14, R15: 10 kohm
R16: 330 ohm
R17, R18: 4,7 kohm
R19: 330 ohm
C1: 470 µF 25 VL elettrolitico
C2: 100 nF multistrato
C3: 100 nF multistrato
C4: 470 µF 25 VL elettrolitico
C5÷C9: 100 nF multistrato
C10, C11: 15 pF ceramico
C12: 100 nF 63 VL poliestere
U1: 7805
U2: 74HC573
U3: ULN2803
U4: Atmel
U5: DS18B20
BZ1: Buzzer senza elettronica
D1÷D3: 1N4007
Q1: Quarzo 20 MHz
LD1÷LD10: LED 5 mm verde
P1÷P3: Microswitch

Varie:

- Zoccolo 9+9
- Zoccolo 10+10
- Zoccolo 20+20
- Vite 12 mm 3 MA
- Dadi 3 MA
- Dissipatore (ML26)
- Jumper
- Strip femmina 4 poli 90°
- Strip maschio 2 poli 90°
- Strip maschio 2 poli (6 pz.)
- Strip maschio 6 poli
- Strip maschio 7 poli
- Strip maschio 8 poli
- Strip maschio 16 poli
- Strip femmina 16 poli
- Strip femmina 20 poli (2 pz.)
- Connettore POD10
- Connettore PS/2 da c.s.
- Plug alimentazione
- C.S. cod.

te. In alto alla sinistra del micro troviamo il connettore JTAG, che può essere utile per la programmazione e per effettuare il debugging del programma lavorando con il micro stesso, anziché (come avevamo visto nella puntata precedente) con il simulatore su PC.
Per far questo occorre, però, disporre di una scheda dedicata supplementare che va collegata da un lato al connettore JTAG, e dall'altro lato al PC tramite porta seriale o USB.
Seguono due connettori utili a collegare al micro il sensore di temperatura DS18B20 ed una tastiera (o mouse) con interfaccia PS/2. Il LED LD9 ed il *cicalino* piezoelettrico BZ1 forniscono all'utente due semplici interfacce di uscita di tipo visivo ed acustico. Infine, le resistenze R14 ed R15 permettono di "programmare" la tensione di fondo scala del convertitore analogico/digitale interno dell'ATMEGA16.

REALIZZAZIONE PRATICA

La demoboard può facilmente essere realizzata anche da chi ha poca pratica con i montaggi elettronici, in quanto è composta esclusivamente da componenti per montaggio tradizionale: armati di un saldatore a punta fine e di una buona lega saldante (tradizionalmente denominata "stagno"), iniziate dai componenti a basso profilo (resistori, diodi, LED, zoccoli degli integrati e strip) per poi passare a quelli più "alti", come condensatori, connettori, ecc.
Prestate particolare attenzione al verso di inserzione dei componenti polarizzati (riferitevi al piano di montaggio nella pagina accanto) e ricordate che i resistori, i condensatori ceramici e in poliestere, il cicalino piezoelettrico ed il quarzo non hanno una polarità.
Sulle sedici piazzole del display LCD suggeriamo di saldare uno strip femmina, in modo da potervi inserire (dopo aver completato la scheda e verificata l'alimentazione) il display stesso, sul quale (se non già presente) dovrete aver precedentemente saldato un pin-strip.
In tal modo il display resta sollevato dalla scheda, per lasciare spazio ai componenti sottostanti.
Nelle quattro piazzole contrassegnate "USB" saldate uno strip femmina a 90° in modo da potervi successivamente inserire il modulo FT782 rivolto con il connettore USB verso l'esterno della scheda.
Per quanto riguarda U1, va montato in orizzontale sopra un apposito dissipatore termico, piegandone i terminali ad angolo retto; fate combaciare i reofori del 7805 con la finestrella del dissipatore e i fori del circuito stampato, quindi fissate il tutto con un bullone 3MA ed il relativo dado. Infine potrete saldare sul lato opposto i tre terminali di U1.
Prima di inserire gli integrati DIP nei rispettivi zoccoli, potete provare a dare alimentazione al sistema collegando nel jack "PWR" un alimentatore da 8÷15 V in continua: il LED LD10 dovrebbe subito accendersi. Verificate anche di avere una tensione di 5 V (±2 %) tra il catodo di D2 ed un qualsiasi punto di massa. Fatto questo, potete togliere alimentazione al circuito ed inserire gli integrati U2, U3 e U4 nei loro zoccoli.
Per ultimo potete inserire il display, facendolo entrare nel rispettivo strip previsto sulla scheda.

LA PROGRAMMAZIONE DEL MICRO

Abbiamo analizzato le procedure di compilazione dei nostri programmi e l'hardware fondamentale per poterli mettere in pratica; l'anello mancante della catena è la programmazione del microcontrollore.
Per eseguire questa operazione è necessario un apposito programmatore in grado di prelevare i dati contenuti nel file oggetto precedentemente creato su PC (come quello in formato *Intel Hex*) e trasferirli, sotto forma di impulsi elettrici ben precisi, nella memoria di programma del microcontrollore stesso.
In particolare, nei microcontrollori AVR si può effettuare la programmazione della memoria trasferendo i dati, in modalità seriale (un bit alla volta), attraverso la *periferica SPI* che fa capo esternamente ad alcuni piedini ben determinati; a questi pin va dunque collegato il programmatore. Nella nostra demoboard il connettore *ICSP*, cui dovremo collegare il programmatore esterno, è infatti connesso a questi piedini.
Sebbene in linea di principio tutti i programmatori siano equivalenti (il loro unico scopo è la programmazione del micro), nello scegliere quello che fa per voi dovete valutare alcune caratteristiche; ad esempio, in ambito industriale, dove bisogna programmare migliaia di microcontrollori al giorno, si valuta la velocità di programmazione: una differenza anche di

soli due secondi per ciascuno ha il suo peso sulla produttività giornaliera.
Nel nostro caso non abbiamo invece particolari problemi e possiamo basare la scelta sul tipo di interfaccia che il programmatore utilizza per il collegamento al PC.

PROGRAMMATORI PER PORTA SERIALE RS-232 OPPURE USB

Essi sono costituiti essenzialmente dai circuiti richiesti per convertire il protocollo seriale RS232 o USB in arrivo dal PC, nel protocollo SPI utilizzato dal micro (in quanto, pur essendo questi tutti protocolli seriali, differiscono notevolmente tra loro). Sono comunque schemi generalmente molto semplici per i quali, anche in virtù della mole di documentazione reperibile in Internet, può valere il discorso dell'autocostruzione. Se invece si desiderasse una soluzione commerciale "pronta all'uso", citiamo il programmatore USB "Pololu USB AVR Programmer", acquistabile dal sito web indicato al punto [1] dei Riferimenti, al prezzo che al momento della stesura di questo corso è dell'ordine di una ventina di euro; tale programmatore può essere gestito da PC con il software Avrdude, che illustreremo più avanti. Inoltre esso mette a disposizione una porta seriale aggiuntiva ed un software chiamato "*SLO-scope*", che permette di trasformarlo in un semplice oscilloscopio a due canali per PC. Nella **Fig. 1** potete vedere come si presenta. Questo programmatore si interfaccia con il microcontrollore attraverso un connettore a vaschetta da 6 poli (altri programmatori possono avere un connettore analogo, ma da 10 poli) i cui piedini sono disposti su due file affiancate (quindi 3+3) a passo 2,54 mm.

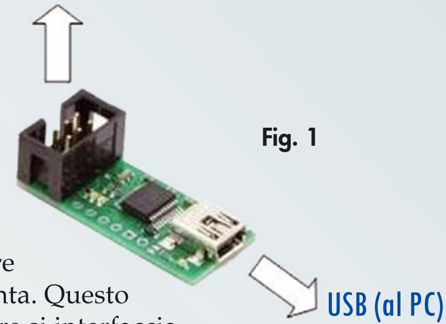
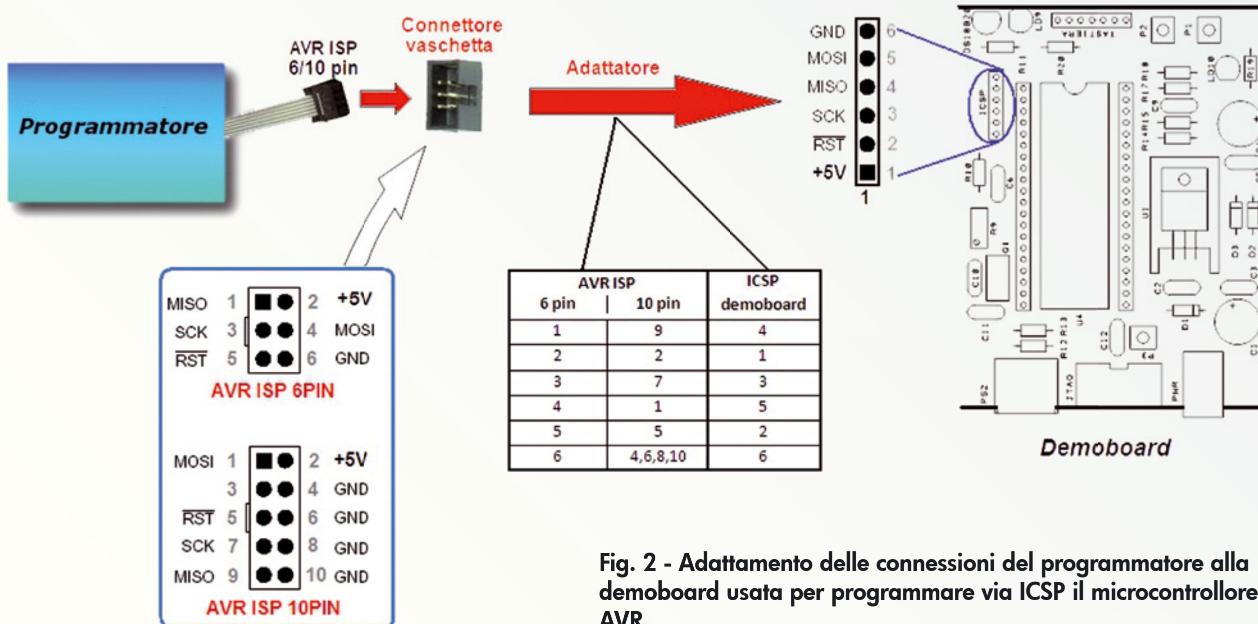


Fig. 1

La connessione dei segnali sui piedini dei connettori aderisce allo standard "AVR ISP", che è lo standard adottato da Atmel per la connessione per la programmazione delle proprie schede di sviluppo equipaggiate con micro AVR. Nel programmatore proposto è compreso anche un cavetto da inserire da un lato nel connettore AVR ISP e dall'altro lato in un analogo connettore a sei poli, sul quale realizzare l'adattamento con i segnali corrispondenti della strip ICSP (vedere la **Fig. 2**) che può essere fatto con fili "volanti" oppure, magari, utilizzando un piccolo ritaglio di circuito millefori (ad esempio una parte dell'area millefori della demoboard stessa). In figura sono riportate anche le connessioni



AVR ISP 6 pin	AVR ISP 10 pin	ICSP demoboard
1	9	4
2	2	1
3	7	3
4	1	5
5	5	2
6	4,6,8,10	6

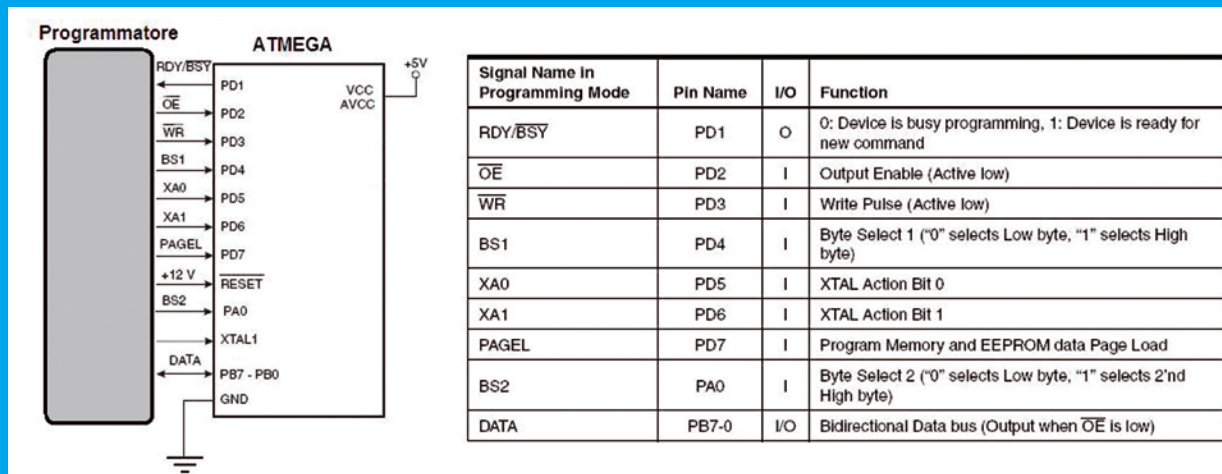
Fig. 2 - Adattamento delle connessioni del programmatore alla demoboard usata per programmare via ICSP il microcontrollore AVR.

Le tipologie di programmazione dei microcontrollori **Atmel ATMEGA**

Per questa famiglia di microcontrollori esistono tre metodi per accedere in scrittura alle memorie interne (Flash ed EEPROM) allo scopo di caricare dati e programmi; ognuno si distingue dagli altri fondamentalmente per il protocollo di comunicazione e per i piedini del micro coinvolti nel processo.

Programmazione parallela

Per eseguire questo tipo di programmazione occorre collegare tutti i piedini del microcontrollore mostrati nella figura seguente ad un apposito programmatore.

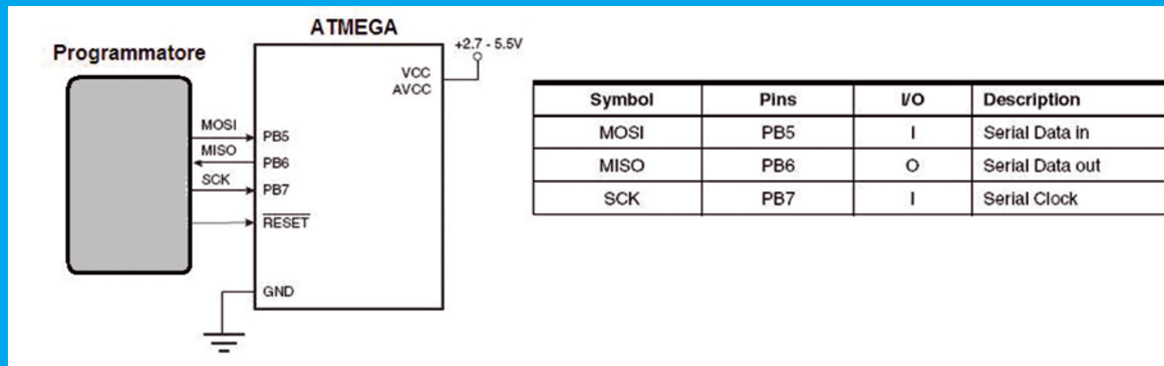


Signal Name in Programming Mode	Pin Name	I/O	Function
RDY/BSY	PD1	O	0: Device is busy programming, 1: Device is ready for new command
OE	PD2	I	Output Enable (Active low)
WR	PD3	I	Write Pulse (Active low)
BS1	PD4	I	Byte Select 1 ("0" selects Low byte, "1" selects High byte)
XA0	PD5	I	XTAL Action Bit 0
XA1	PD6	I	XTAL Action Bit 1
PAGEL	PD7	I	Program Memory and EEPROM data Page Load
BS2	PA0	I	Byte Select 2 ("0" selects Low byte, "1" selects 2'nd High byte)
DATA	PB7-0	I/O	Bidirectional Data bus (Output when OE is low)

Il vantaggio principale della programmazione parallela è la velocità, in quanto i dati vengono trasferiti a 8 bit per volta; lo svantaggio è l'elevato numero di connessioni al programmatore e la necessità di disporre di una tensione di 12 V da applicare al piedino di *Reset* del micro.

Programmazione seriale

Le connessioni da realizzare tra il micro ed il programmatore per ottenere la programmazione via seriale, sono mostrate nella figura in basso.



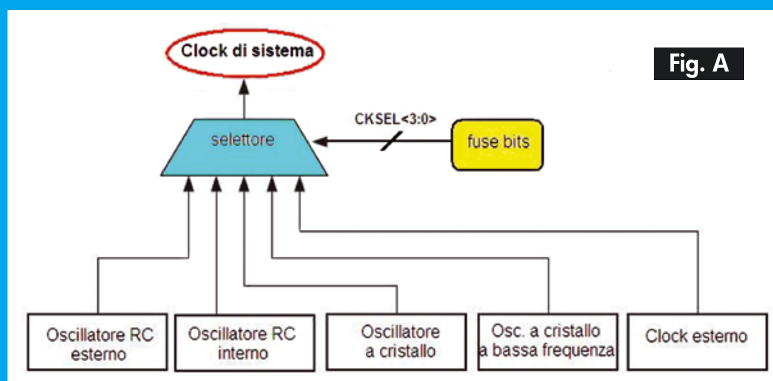
Symbol	Pins	I/O	Description
MOSI	PB5	I	Serial Data in
MISO	PB6	O	Serial Data out
SCK	PB7	I	Serial Clock

In questa modalità di programmazione si utilizza una singola linea (anziché otto) per trasferire i dati, a vantaggio di un ridotto numero di collegamenti, il che fa da contraltare ad una maggior lentezza di programmazione rispetto alla programmazione parallela. Come si può desumere dalla figura, concettualmente il protocollo di comunicazione è molto semplice, in quanto si utilizza una linea per trasmettere i dati dal programmatore al micro (MOSI: *Master Out, Slave In*) ed un'altra per la direzione opposta (MISO: *Master In, Slave Out*) utile alla verifica dei dati scritti; entrambe sono cadenzate dal segnale di *Clock* (SCK) gestito dal programmatore. Questo protocollo di comunicazione è abbastanza comune nel mondo dei microcontrollori, non solo per la programmazione ma anche per l'interfacciamento ad altre periferiche, e va sotto il nome di SPI (*Serial Peripheral Interface*). Dato l'esiguo numero di linee necessarie, questa modalità di programmazione è normalmente più utilizzata di quella parallela, anche perché permette di programmare il microcontrollore direttamente sulla scheda sulla quale dovrà funzionare; dopo aver effettuato la programmazione, il connettore può essere rimosso, soprattutto se nell'applicazione debbono essere utilizzate le linee del bus SPI. Il grosso vantaggio è che il micro può essere già saldato direttamente sulla scheda, e non dev'essere neppure successivamente dissaldato nel caso si debba riprogrammarlo. A questo tipo di programmazione (con il micro già montato sulla scheda di lavoro) ci si riferisce con il termine di *In-System Programming* (detto in breve *ISP*, da non confondere con *SPI*, che è il protocollo di comunicazione col programmatore). Da notare, infine che le modalità di programmazione *seriale* e parallela appena descritte non dipendono dal tipo di interfacciamento tra il programmatore ed il PC (seriale, parallelo o USB di cui abbiamo parlato nell'articolo); infatti il programmatore "BSD" proposto in questo articolo comunica con il microcontrollore utilizzando il protocollo (seriale) SPI, ricostruito *"via software"* dal programma Avrdude che gestisce in modo opportuno i quattro segnali in arrivo dal PC sulla porta parallela:. Quindi in questo caso l'interfaccia *PC↔programmatore* è parallela, mentre quella *programmatore↔microcontrollore* è seriale.

I "FUSE bits"

Ogni microcontrollore necessita per funzionare, oltre al programma vero e proprio, di alcune informazioni preliminari che gli consentano di operare nel modo corretto. L'esempio più importante riguarda il segnale di clock, ovvero la temporizzazione necessaria a scandire le fasi di lettura, decodifica ed esecuzione delle istruzioni: nel caso dei microcontrollori ATMEGA, tale segnale può essere ricavato da un oscillatore collegato ad un piedino, oppure generato da un risonatore esterno, anch'esso collegato ad appositi piedini, o addirittura essere generato da un oscillatore interno al micro stesso. Di tutte queste "sorgenti" di clock ne viene quindi selezionata una per utilizzarla come temporizzazione del sistema (**Fig. A**).
La selezione della sorgente da utilizzare è controllata dal valore dei *"fuse bits"* CKSEL<3:0> che dunque debbono essere impostati (o meglio, programmati) prima che il sistema sia acceso: questo è il motivo per cui normalmente tali bit vengono programmati nel micro insieme al codice del programma vero e proprio. In altri microcontrollori è possibile modificare le impostazioni degli oscillatori mentre il micro è già in funzione, e dunque può essere il programma stesso ad occuparsene, mentre nel nostro caso, no.
Lo specchietto seguente (preso dal data-sheet nella sezione "Memory Programming") mostra l'elenco dei *fuse bits* del micro ATMEGA16; essi sono 16 in tutto, e vengono raggruppati in due byte (*"Fuse high byte"* e *"Fuse low byte"*). Nel microcontrollore questi due byte fanno parte di un'area di memoria, separata da quella di programma, accessibile e riscrivibile tramite il software Avrdude usando i comandi "hfuse" e "lfuse", come visto in queste pagine (**Fig. B**). Nella colonna *Default Value* vi sono i valori impostati "dalla fabbrica", ovvero lo stato in cui si trovano quando comprate un microcontrollore nuovo; poiché quando si cancella la memoria i fuse bits sono riportati tutti a '1', nel datasheet tale stato viene detto "unprogrammed", mentre il livello '0' è chiamato "programmed").
Nel byte "alto" (*Fuse high byte*) vi sono i seguenti bit:

- OCDEN, JTAG; permettono di attivare le funzioni di *on-chip debugging*, attraverso la porta JTAG.
- SPIEN; se posto a '0' abilita la programmazione seriale del micro (attraverso la porta SPI) mentre se viene messo ad '1' non si può più effettuare la programmazione con tale metodo (quindi è sconsigliato impostare '0') e per effettuare ulteriori programmazioni bisogna ricorrere ad un programmatore con accesso parallelo al microcontrollore;
- CKOPT; controlla l'amplificazione dell'oscillatore esterno;
- EESAVE; se posto a '0' la memoria EEPROM, nella fase di cancellazione (propedeutica ad una nuova programmazione) non viene cancellata insieme alla Flash;
- BOOTSZ<1:0>; permettono di impostare la grandezza di un particolare settore della memoria Flash chiamato *boot sector*, utile in particolari applicazioni.
- BOOTRST; se posto a '0', dopo un reset il processore va a leggere il codice dall'inizio della sezione di *boot sector*, piuttosto che dall'inizio della memoria fisica (che è 0x0000).



Nel byte "basso" (*Fuse low byte*) troviamo:

- BODLEVEL; seleziona la soglia di intervento del *brown-out* (in pratica determina il minimo valore di tensione di alimentazione del micro al disotto del quale la CPU viene bloccata nello stato di reset); se il bit è '1' tale valore è circa 2,7 V, ovvero 4 V se il bit è a zero, pertanto se alimentiamo il dispositivo

"AVR ISP" dei connettori a dieci poli, nel caso vi procuraste un programmatore con tale attacco; in tal caso vi sono solo più connessioni verso massa, mentre il piedino 3 non va collegato.
Notate che, siccome il connettore a sei piedini è identico a quello JTAG presente nella demoboard, dovete fare attenzione a non collegare su di esso quello di programmazione, per non rovinare l'ATMEGA16 o il programmatore stesso, in quanto i segnali sono completamente differenti.
Nella **Fig. 3** viene mostrato, tra le altre cose, lo schema delle connessioni del connettore "AVR ISP" a sei piedini (chiamato CON3) con lo strip della demoboard siglato ICSP nello schema elettrico della scheda.

PROGRAMMATORI PER PORTA PARALLELA

In questa categoria rientrano i programmatori più economici in assoluto, anzi praticamente a costo zero, in quanto constano solo dei connettori e di una sorgente di alimentazione per il micro, dato che il protocollo SPI è ricostruito (grazie ad un apposito software che "gira" sul PC) su alcuni segnali della porta parallela, i quali quindi possono essere direttamente collegati alla porta SPI del microcontrollore; ciò può essere fatto solo se il micro viene alimentato a 5 V, che è la tensione con cui lavora la parallela dei computer.
Notate che questo tipo di programmatori in genere funziona solo se il PC è dotato "nativamente" di una porta parallela; nel caso ne fosse sprovvisto, sconsigliamo di ricorrere

Fig. B

Fuse High Byte	Bit No.	Description	Default Value
OCDEN	7	Enable OCD	1 (unprogrammed, OCD disabled)
JTAGEN	6	Enable JTAG	0 (programmed, JTAG enabled)
SPIEN	5	Enable SPI Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size	0 (programmed)
BOOTSZ0	1	Select Boot Size	0 (programmed)
BOOTRST	0	Select reset vector	1 (unprogrammed)

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown-out Detector trigger level	1 (unprogrammed)
BODEN	6	Brown-out Detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)
SUT0	4	Select start-up time	0 (programmed)
CKSEL3	3	Select Clock source	0 (programmed)
CKSEL2	2	Select Clock source	0 (programmed)
CKSEL1	1	Select Clock source	0 (programmed)
CKSEL0	0	Select Clock source	1 (unprogrammed)

a 3,3 V dobbiamo impostare BODLEVEL ad ‘1’, mentre a 5 V conviene metterlo a ‘0’; Il brown-out è una funzionalità molto utile in quanto consente, nella fase di accensione del sistema (in cui la tensione di alimentazione sale gradatamente da zero al valore nominale), di tenere bloccato il micro (nello stato di reset) sino ad un valore che gli consenta di partire regolarmente;

- BODEN; abilita la funzionalità di brown-out se è posto ‘0’ (impostazione sempre consigliabile); Se il bit è a ‘1’ il brown-out rimane invece disabilitato;
- SUT<1:0>; permettono di selezionare un ritardo aggiuntivo alla partenza della CPU dopo la fase di reset;
- CKSEL<3:0>; selezionano la sorgente del clock per la CPU e le periferiche, come già accennato; rimandiamo al data-sheet per ricavare la combinazione di questi bit in base al tipo di oscillatore che si desidera utilizzare (capitolo “System clock and clock options”).

Per fare un esempio pratico, vediamo a che cosa corrispondono i valori dei *fuse bits* che abbiamo impostato nella riga di comando di Avrdude del programma proposto qui, ovvero 0x99 (hfuse) e 0x2E (lfuse). Siccome 0x99 vale 1001 1001 in binario, otteniamo:

- OCDEN, JTAG = 10; *on-chip debugging* disattivo, interfaccia JTAG abilitata;
- SPIEN = 0; programmazione seriale abilitata;
- CKOPT = 1; oscillatore a basso guadagno (se però nel *fuse low byte* viene selezionato l’oscillatore esterno ed esso è un quarzo a frequenza superiore agli 8 MHz, conviene mettere questo bit a ‘0’);
- EESAVE = 1; la EEPROM viene cancellata insieme alla memoria Flash;
- BOOTSZ<1:0> = 00; *boot sector* di 1.024 word;
- BOOTRST = 1; al reset, il processore parte leggendo dalla memoria programma normale.

Per quanto riguarda il *fuse low byte*, abbiamo 0x2E, che in binario vale 0010 1110, quindi:

- BODLEVEL = 0; livello di attivazione del brown-out a circa 4 V, che nella nostra demoboard va bene in quanto il micro viene alimentato a 5 V;
- BODEN = 0; brown-out abilitato;
- SUT<1:0> = 10; tempo di partenza dopo un reset impostato a 65ms;
- CKSEL<3:0>=1110; oscillatore esterno ad alta frequenza (quindi nella demoboard dev’essere presente il quarzo Q1).

Nei programmi che presenteremo successivamente in questo corso, si potranno normalmente lasciare queste impostazioni nei *fuse bits*, se non specificato diversamente. In ogni caso, salvo un’approfondita consultazione del data-sheet per avere piena confidenza con ciò che si sta facendo consigliamo di non cambiare mai i valori dei bit contrassegnati in rosso, pena l’impossibilità di riutilizzare il micro.

a convertitori USB/parallela, in quanto essi possiedono un proprio circuito di emulazione della porta che i software di programmazione come Avrdude non sono in grado di gestire correttamente. In tal caso è meglio ricorrere piuttosto ad un programmatore USB vero e proprio.
Uno dei possibili schemi di interfacciamento tra la porta parallela ed il microcontrollore ATMEGA16 è riportato in **Fig. 3**: esso fa riferimento al cosiddetto “Programmatore BSD” (*http://www.bsdhome.com/avrdude/*) pensato per funzionare con il programma Avrdude.
I segnali necessari sono quattro (più la massa) e vengono direttamente portati, tramite altrettanti resistori (inseriti allo scopo di proteggerle le linee corrispondenti) ai piedini del micro, secondo quanto mostrato nella **Fig. 3**; a lato è riportata, per chiarezza, anche la disposizione fisica dei quaranta piedini nel contenitore *dual-in-line* dell’ATMEGA16.
Il connettore CON2 consente, collegandolo tramite sei fili alla porta ICSP della demoboard, di programmarvi direttamente il micro senza doverlo neppure rimuovere, da cui il significato del nome ICSP (*In-Circuit Serial Programming*, ovvero programmazione seriale direttamente sulla scheda). In tal caso il programmatore di **Fig. 3** si riduce ai connettori CON1 e CON2 ed alle relative connessioni.
Nel riquadro tratteggiato della stessa figura sono proposte tre aggiunte allo schema “base” del programmatore:
1) connettore CON4=predisposto per colle-

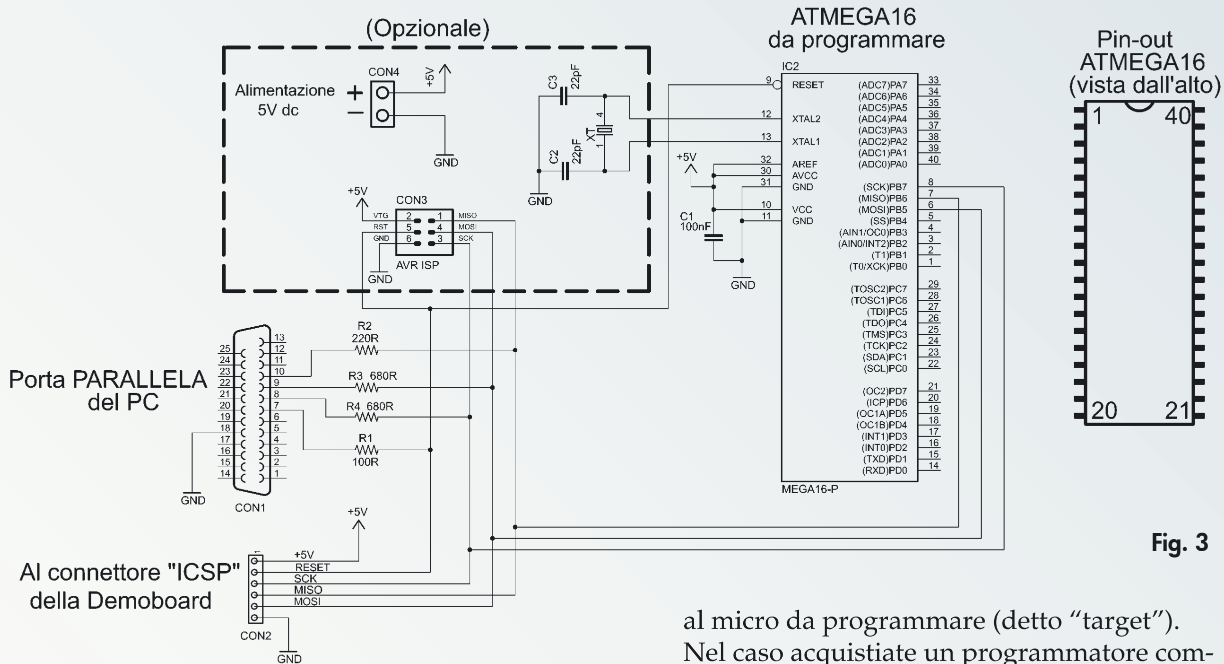


Fig. 3

garvi una sorgente di alimentazione che fornisca una tensione di +5V per alimentare il micro; è necessario se il programmatore non viene collegato alla demoboard (dalla quale preleva anche l'alimentazione, sul piedino 1 di CON2);

2) connettore CON3=è il connettore a vaschetta con le connessioni standard "AVR ISP" a sei piedini dei programmatori commerciali: se li utilizzate (al posto della porta parallela), il loro connettore va inserito in esso;
3) quarzo XT=necessario se il microcontrollore è già stato programmato per utilizzare un oscillatore esterno (tramite i bit CKSEL<3:0> dei *fuse bits*, come spieghiamo più avanti); un tipico valore di frequenza del quarzo potrebbe essere 8 MHz.

Quindi, se non avete realizzato la demoboard o comunque volete programmare i microcontrollori sul programmatore, il connettore CON2 non è necessario ma vi serve un'alimentazione a +5 V esterna (e ben stabilizzata) da applicare su CON4.

IL SOFTWARE DI PROGRAMMAZIONE AVRDUDE

Ogni programmatore necessita di un software di gestione attraverso il quale l'utente seleziona innanzitutto il microcontrollore, il file oggetto con cui programmarlo ed eventuali altre opzioni; in seguito è sempre questo software a prelevare i dati dal disco del PC, trasferirli attraverso la periferica prescelta (seriale, parallela, USB) al programmatore "fisico" e quindi al micro da programmare (detto "target").

Nel caso acquistiate un programmatore commerciale, il relativo software viene normalmente fornito a corredo dal produttore; se invece optate per l'autocostruzione, il software dovete procurarvelo da voi.

Notate che installando il pacchetto WinAVR vi trovate compreso anche Avrdude, che è un ottimo programma gratuito in grado di interfacciarsi con diversi programmatori USB, seriali e paralleli, tra cui il programmatore USB "Pololu" e quello parallelo "BSD". Di seguito spiegheremo nel dettaglio come utilizzare quest'ultimo.

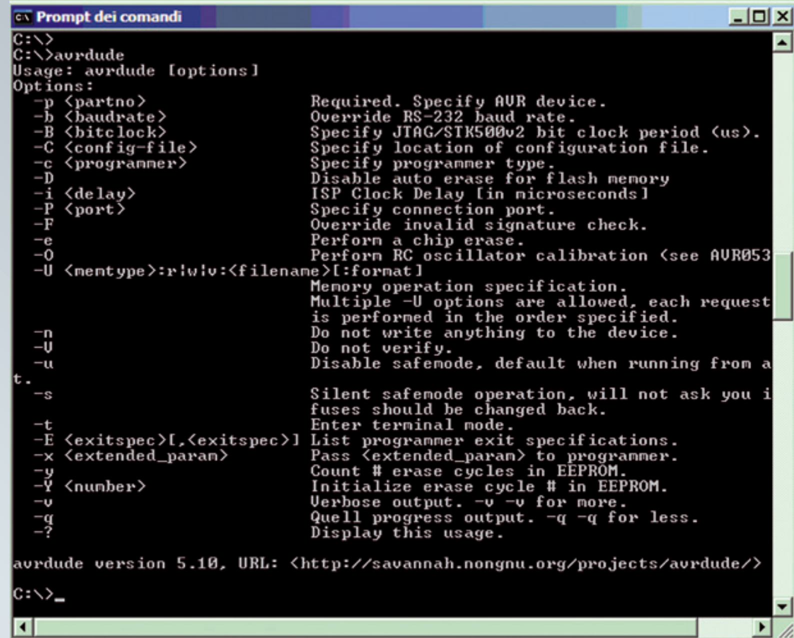
Tra l'altro, a chi ne fosse interessato, segnaliamo che Avrdude è disponibile anche per il sistema operativo Linux.

Avrdude è essenzialmente un programma a linea di comando, per cui dev'essere invocato da una qualsiasi console di Windows; apritene una (accedendo da *Start->Esegui* e scrivendo *cmd* nella finestra di dialogo), digitate nella riga di comando il termine `avrdude` e premete il tasto INVIO. A questo punto dovrebbe apparire la schermata di **Fig. 4**, nella quale si osserva che il programma "risponde" (dunque l'avete installato), e mostra un sommario dei comandi da far seguire all'invocazione stessa, necessari per fargli eseguire una precisa azione (programmazione, verifica, lettura, ecc.).

Se avete già realizzato il programmatore proposto in **Fig. 3** (connettore per porta parallela, connettore ICSP e relativi collegamenti) siete finalmente in grado di programmare "concretamente" il microcontrollore.

Nota: se è la prima volta che usate Avrdude ed avete un sistema operativo Windows 2000

Fig. 4



o superiore (quindi i Windows XP, Vista, 7), dovete anche preliminarmente eseguire il file *install_giveio.bat* che trovate nella cartella *C:\WinAVR-20100110\bin* (da Esplora Risorse di Windows, basta fare doppio clic sul nome del file), che serve a consentire ad Avrdude di accedere ai registri della porta parallela. Questa operazione dovete comunque eseguirla solo la prima volta. Fatto ciò, potete attenervi alla seguente procedura:

1) eseguite i collegamenti tra il connettore ICSP della demoboard e CON2 sul programmatore, ed alimentate la demoboard (8÷15 Vcc sull'ingresso PWR); se non usate la demoboard, inserite direttamente il microcontrollore nello zoccolo del programmatore (IC2) e fornite i +5 V su CON4 (fate attenzione alla polarità);
2) dalla console di Windows spostatevi sulla cartella che contiene il file *PrimoProgetto.hex* precedentemente creato dalla compilazione del sorgente; se avevate creato il progetto in "C:\AVR\PrimoProgetto", si trova in "C:\AVR\PrimoProgetto\default" quindi nella console digitate il comando `cd C:\AVR\PrimoProgetto\default` seguito da INVIO.
3) impartite il comando `avrdude -p m16 -c bsd -P lpt1 -u -U flash:w:PrimoProgetto.hex -U hfuse:w:0x99:m -U lfuse:w:0x2E:m` seguito da INVIO, per avviare Avrdude.

Prima di impartire questo comando vi conviene verificare il numero della porta parallela alla quale avete collegato il programmatore e, se è diversa da LPT1, al termine `lpt1` sostituite quello corretto: per esempio, se il vostro computer ha due parallele e siete collegati alla LPT2, il comando da impartire diventa `avrdude -p m16 -c bsd -P lpt2 -u -U flash:w:PrimoProgetto.hex -U hfuse:w:0x99:m -U lfuse:w:0x2E:m` , sempre seguito da INVIO.
Durante l'esecuzione del comando, Avrdude mostrerà la sequenza delle operazioni in corso (cancellazione, programmazione e verifica della memoria) fino alla fatidica scritta finale, "*AVRDUDE done! Thank you*", indicante che il programma è stato avviato.
Riguardo alla "criptica" riga di comando con cui abbiamo avviato Avrdude, notate che è semplicemente formata dal comando `avrdude` e da una serie di parametri descritti di seguito:

"`-p m16`"= seleziona il microcontrollore da programmare ('m16' sta per ATMEGA16);
"`-c bsd`"= seleziona il programmatore collegato al computer; in questo caso è il BSD;
"`-P lpt1`" = seleziona la porta del PC cui il programmatore è collegato: in questo caso, la "LPT1" (porta parallela numero 1);
"`-u`" = consente la modifica dei *fuse bits*;
"`-U flash:w:PrimoProgetto.hex`" = opera la scrittura in flash del file *PrimoProgetto.hex*;
"`-U hfuse:w:0x99:m`" = programma il byte *hfuse* con il valore 0x99;
"`-U lfuse:w:0x2E:m`" = programma il byte *lfuse* con il valore 0x2E.

Normalmente, nello sviluppo di un firmware capita di dover riprogrammare il microcontrollore numerose volte (ciò è possibile grazie alla memoria di programma di tipo Flash); per non dovere riscrivere sempre la lunga riga di comando, è consigliabile creare un file *batch*, ovvero un file di testo (al cui interno scriviamo l'intera riga di comando di Avrdude mostrata al punto 3) della procedura appena descritta, salvandolo poi con l'estensione **.bat**; ad esempio, se lo salviamo col nome *programma_micro.bat*, dal prompt dei comandi (sotto la stessa directory dove il file batch si trova) ci basta digitare `programma_micro` seguito da INVIO, per rieseguire AVRDUDE con tutti i parametri specificati. Il file batch va salvato ed eseguito dalla cartella in cui è contenuto il file '.hex' di

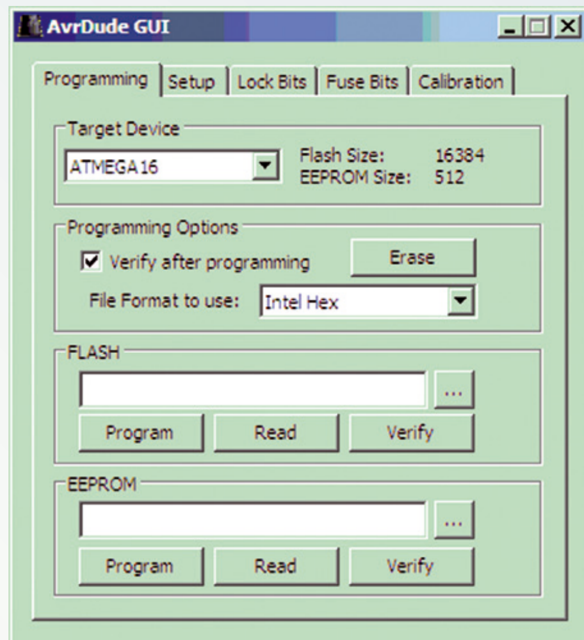


Fig. 5

programmazione del micro ("C:\AVR\Primo-Progetto\default" nell'esempio).
Il nome del file da caricare nel micro **.hex** specificato nella riga di comando dev'essere ovviamente lo stesso di quello contenuto nella cartella. Invece alcuni parametri come i *"fuse bits"* (descritti nel relativo specchietto di questa puntata del corso) possono essere modificati secondo esigenze particolari che analizzeremo in seguito. Per comprendere appieno come utilizzare i parametri di Avrdude, il nostro consiglio è di consultare il manuale del programma, che si trova in *"<cartella installazione WinAVR>\doc\avrdude"* (tipicamente *"C:\WinAVR-20100110\doc\avrdude"*).
Potete trovare una rapida guida che vale la pena di leggere, in Internet sul sito citato al punto [3] dei Riferimenti.

RIGA DI COMANDO O GUI?

A qualcuno sarà gradito sapere che esiste anche un'interfaccia grafica (*GUI*) del programma Avrdude, decisamente più intuitiva e "user-friendly" del tradizinoale prompt dei comandi; questa interfaccia si scarica gratuitamente dal sito web citato nel punto [4] dei Riferimenti, perché non fa parte del pacchetto WinAVR. Dunque, se volete usare la GUI scaricate e scompattate i file in una nuova cartella qualsiasi (ad esempio *"C:\Programmi\AvrdudeGUI"*); poiché non vi è nessuna installazione ulteriore da eseguire, potete avviare subito il programma semplicemente facendo doppio click su *avrdudeGUI.exe;* allora comparirà la finestra mostrata in **Fig. 5**.

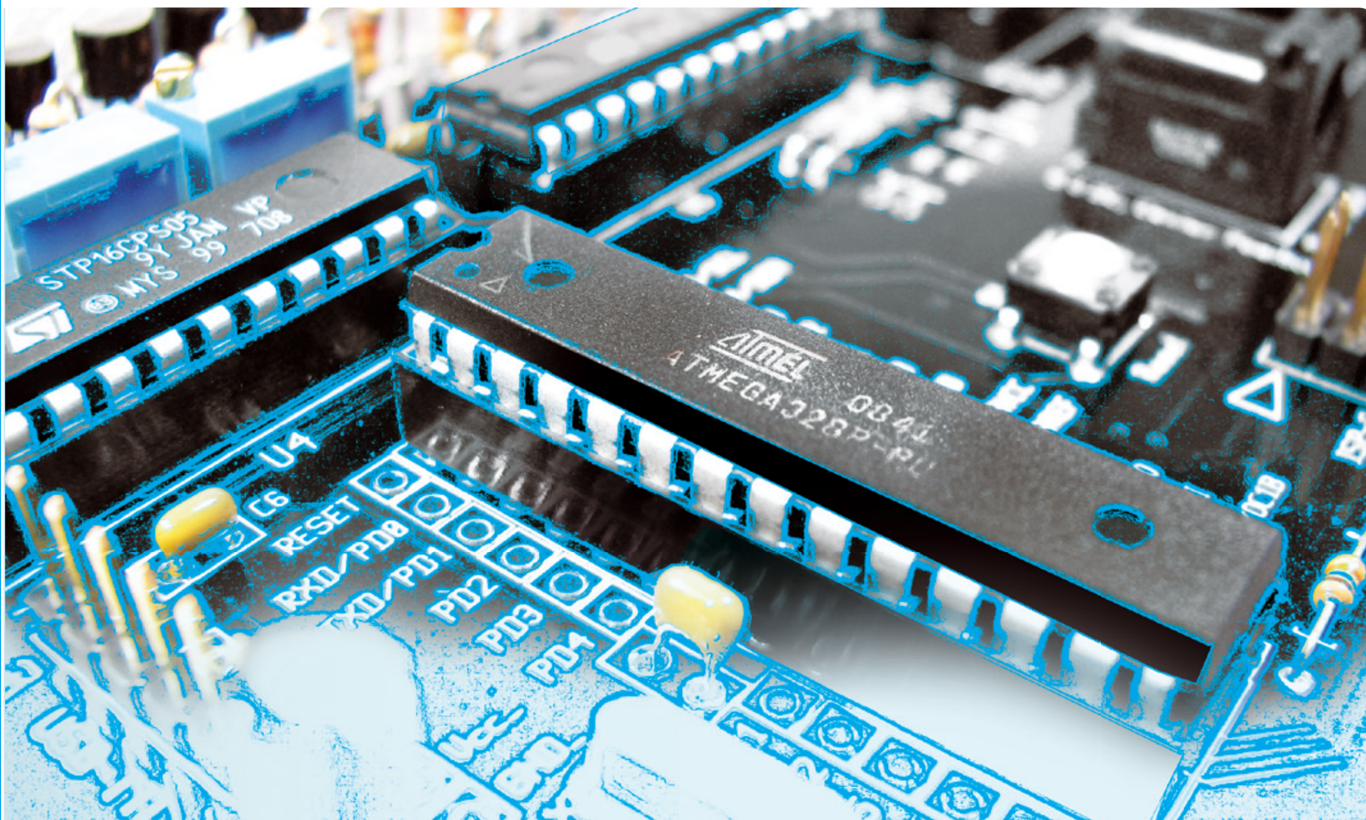
Il funzionamento dell'interfaccia grafica è molto intuitivo: innanzitutto si imposta, sotto la linguetta "Setup", il tipo di programmatore utilizzato (ad esempio, se usate quello parallelo proposto nell'articolo, "BSD") e la porta del PC cui esso è collegato; in seguito si vanno a impostare i valori dei Lock e Fuse bits.
Poi si fa clic sulla linguetta "Programming" e in questa bisogna scegliere il modello del microcontrollore che si desidera programmare (il "Target Device", ad es. ATMEGA16); dopo, accedendo alla casella "FLASH" si seleziona il file oggetto (con il formato "Intel Hex" si possono usare i file con estensione **.hex**) con cui programmarlo (notate che cliccando sul tastino coi tre puntini si possono sfogliare le cartelle alla ricerca del file desiderato).
Infine, cliccando su "Program" (sempre nel riquadro "FLASH") il file oggetto viene scaricato nel microcontrollore. Al termine della procedura, cliccando su "Verify" si può verificare la correttezza della scrittura in Flash appena conclusa. Il tasto "Erase"permette di cancellare la memoria di un microcontrollore già programmato, prima di una successiva scrittura.

CONCLUSIONI

Bene, in queste pagine abbiamo presentato la scheda demoboard che utilizzeremo come base "hardware" ed un semplice programmatore: insieme all'ambiente di sviluppo *AVR Studio* ed al compilatore *avr-gcc*, ci siamo dotati di tutti gli strumenti necessari allo sviluppo pratico di applicazioni basate sul nostro ATMEGA16, che inizieremo dalla prossima puntata.
La scheda descritta utilizza componenti facilmente reperibili ovunque mentre il master del circuito stampato può essere scaricato dal sito della rivista.

Riferimenti

[1] http://www.robotstore.it/product/153/Pololu-USB-AVR-Programmer-per-microcontrollori-ATMEL.asp
[2] http://www.robot-italy.com/product_info.php?cPath=1_69&products_id=908
[3] http://www.ladyada.net/learn/avr/avrdude.html
[4] http://sourceforge.net/projects/avrdudegui/



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

Presentiamo gli strumenti per lavorare praticamente con il microcontrollore ATMEGA16: apprendiamo i sistemi per programmarlo costruendo una scheda di sviluppo in cui inserirlo. Quinta puntata.

In questa puntata descriviamo una applicazione pratica basata sui concetti sinora discussi, per mettere in funzione il microcontrollore ATMEGA16 della nostra demoboard.
Ebbene ci siamo: dopo avere visto che cos'è un microcontrollore, avere imparato a scrivere, compilare e simulare i programmi, ed infine avere discusso l'utilizzo degli strumenti hardware (programmatore e demoboard) è venuto il momento di mettere tutto insieme per sviluppare un'applicazione che faremo concretamente eseguire dal microcontrollore ATMEGA16: trattandosi della prima prova pratica in assoluto, il suo scopo sarà l'accensione di un LED ad esso collegato. Anche se la cosa potrebbe suonare apparentemente banale, tenete presente che questa prima applicazione permetterà di imparare nuovi concetti di programmazione e preparare la base per la realizzazione di programmi molto più complessi.

PREPARAZIONE DEL SISTEMA

Per lo sviluppo delle applicazioni pratiche vi

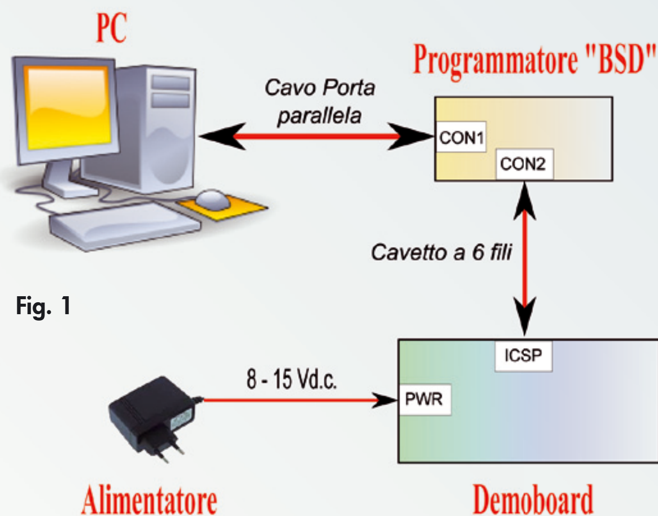


Fig. 1

consigliamo di allestire un "banco di prova" (il cosiddetto setup) che, come schematizzato in **Fig. 1**, è composto da un PC (su cui sono installati WinAVR e AVR Studio), la demoboard ed il programmatore parallelo "BSD".
Se tuttavia il vostro PC non dispone di una porta parallela, in base a quanto esposto nella scorsa puntata, dovrete ricorrere ad un programmatore seriale (se avete disponibile tale porta) oppure USB (come il modello "Pololu USB AVR Programmer", reperibile su Internet). Il cavo per la connessione alla porta parallela può essere acquistato in qualunque negozio di informatica; e va collegato al PC sulla relativa porta (vaschetta D-SUB a 25 piedini). Il cavetto a sei fili che collega il programmatore al connettore ICSP della demoboard, invece, si può auto costruire con sei spezzoni di filo lunghi al massimo 20cm, che si possono saldare ai capi di un connettore strip femmina per poterla collegare e scollegare agevolmente sul corrispondente correttore maschio della scheda. Con questo sistema potete programmare l'ATMEGA16 direttamente sulla demoboard, senza doverlo sempre sfilare per inserirlo nello zoccolo del programmatore. Utilizzando un programmatore commerciale, che "esce" con un connettore a 3+3 oppure 5+5 piedini detto "AVR ISP", dovrete anche realizzare le connessioni di adattamento (Figura 2 della scorsa puntata).
Infine potete alimentare il sistema collegando un alimentatore che fornisca una tensione compresa tra 8 e 15V. In **Fig. 2** si può osservare la disposizione dei componenti del sistema che abbiamo utilizzato per sviluppare i programmi del corso.
Nota: anche se non avete ancora costruito una demoboard, per eseguire l'esercitazione di questa puntata è sufficiente collegare il micro (anche utilizzando collegamenti filati su una scheda millefori) secondo lo schema proposto nella Figura 3 della scorsa puntata, e collegarvi un diodo Led, tramite un resistore da 220-330 ohm, tra il piedino 21 e la massa (come nello schema della demoboard). Ovviamente dovrete anche fornire un'alimentazione di 5V stabilizzati.

E LUCE SIA!

Innanzitutto scegliamo quale LED utilizzare: sulla demoboard abbiamo disponibile LD9, che è collegato direttamente sul pin PD7 del microcontrollore tramite la resistenza di limitazione di corrente R16. Poiché LD9 è collegato verso massa, potremo accenderlo mandando PD7 a +5V (1 logico) e spegnerlo mettendo PD7 a 0V (0 logico). Una volta eseguiti questi collegamenti possiamo scrivere un primo listato del programmino: per farlo

Fig. 2

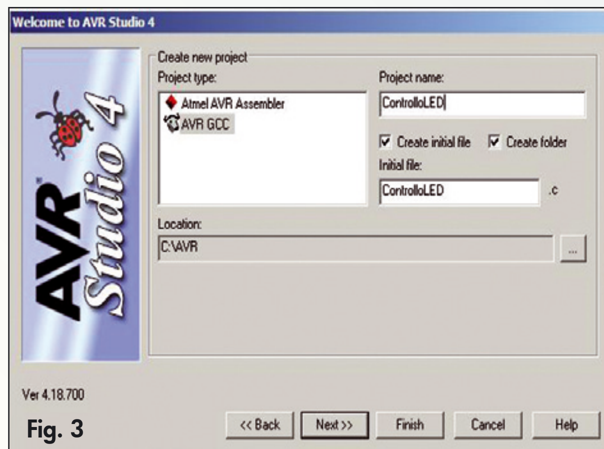


Fig. 3

```
C:\AVR\ControlloLED\ControlloLED.c
#include <avr/io.h>

void main(void) {

/* Inizializzazione */

    DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB sono impostati come uscite */
    DDRC = 0b10111100;  /* PC0, PC1 e PC6 sono ingressi */
    DDRD = 0b11110011;  /* PD2 e PD3 sono ingressi */

/* Fine inizializzazione */

    PORTD = 0x80;  /* scrive '1' su PD7 -> Led acceso */

    while (1) {
        /* main loop: ciclo infinito (vuoto in questo caso) */
    }
}
```

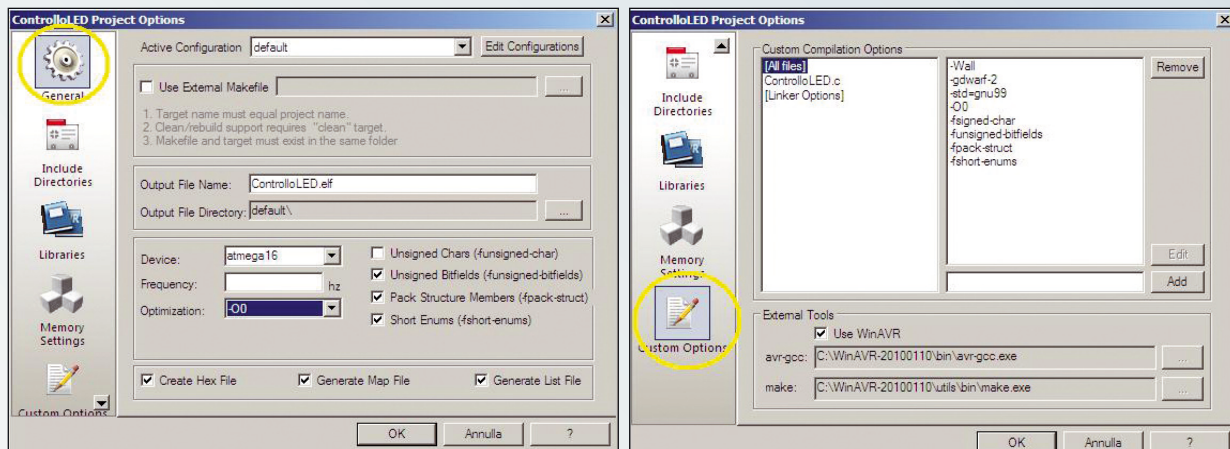
Fig. 4

creiamo innanzitutto in AVR Studio un nuovo progetto, chiamandolo "ControlloLED": sul "Project Wizard" di AVR Studio (da barra menu: Project → Project Wizard) clicchiamo sul tasto "New project", quindi su "AVR GCC" e compiliamo la finestra come riportato in **Fig. 3**; completiamo cliccando su "Finish".
Il nostro programma, visibile in **Fig. 4**, andrà immesso nella finestra di editor del sorgente (ControlloLED.c).
La spiegazione del suo funzionamento è molto semplice: all'interno della funzione principale `main()` eseguiamo innanzitutto la corretta inizializzazione della direzione dei piedini di tutti i port (ricordate quanto detto in proposito nella seconda puntata?) in accordo con le connessioni del micro nel nostro sistema. Dallo schema elettrico della demoboard risulta che tutti i pin dei port A, B, C, D devono essere configurati come uscite tranne PC0, PC1, PC6, PD2 e PD3 i quali, essendo collegati al potenziale di +5V tramite resistori di *pull-up*, possono essere impostati come ingressi anche se le periferiche a cui sono collegati non vengono (per il momento) utilizzate.
L'istruzione

```
PORTD = 0x80;
```

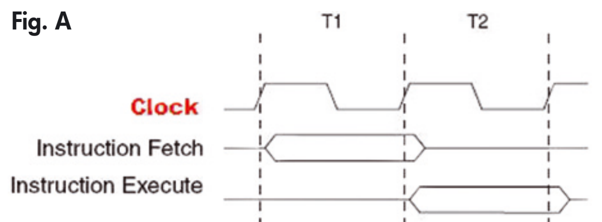
serve a scrivere '1' in PD7: infatti 0x80 (oppure `0b10000000`) equivale in binario a 1000 0000, ovvero mettiamo a '1' il bit più a sinistra (detto Most Significant Bit o MSB, il bit "più significativo" del byte) che, nella numerazione da 0 a 7, è appunto il settimo. Per completezza diciamo che analogamente il bit più a destra è detto LSB (da Least Significant Bit) ovvero bit meno significativo. Gli altri sette bit del port (PD<6:0>) vengono invece impostati a zero. Scrivere '1' su PD7 significa che quando il processore eseguirà quell'istruzione il piedino omonimo verrà collegato alla tensione di alimentazione dell'ATMEGA16 (+5V) e il LED si accenderà.
Il programma entra infine in un ciclo (*loop*) ottenuto con l'espressione `while` (1), che si richiude su se stesso all'infinito, senza peraltro eseguire nulla (il corpo del ciclo, tra le parentesi graffe, è vuoto). È comunque necessario poiché sappiamo che nei microcontrollori la funzione principale `main()` non deve mai terminare.
Passiamo alla compilazione: innanzitutto, trattandosi di un nuovo progetto, bisogna preliminarmente impostare le opzioni di configurazione (da menu: Project → Configuration Options) come avevamo già visto nella terza puntata e che sono riassunte per comodità in **Fig. 5**.

Fig. 5

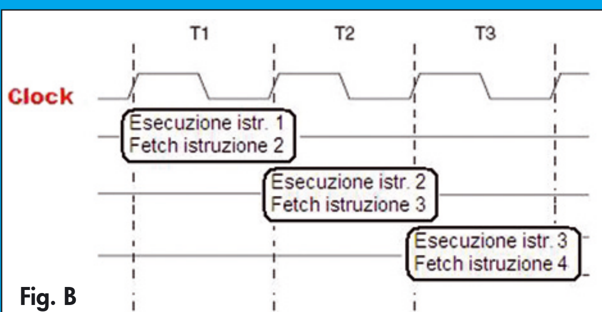


La velocità della CPU

Avevamo già visto che il cuore dei microcontrollori, la CPU, trasforma le istruzioni in arrivo dalla memoria di programma in azioni concrete attraverso le operazioni di lettura, decodifica ed esecuzione. Tali operazioni non avvengono in maniera istantanea e contemporaneamente, ma sono cadenzate dal clock (orologio) del sistema: esso è un segnale che, cambiando continuamente stato (0 → 1 → 0 → 1...) ad intervalli di tempo perfettamente regolari, gestisce la temporizzazione di tutte le fasi delle attività svolte della CPU. La figura seguente mostra come avviene un ciclo completo di esecuzione di un'istruzione, in relazione al segnale di clock, per la CPU dei micro AVR.



La prima fase è il prelievo dell'istruzione dalla memoria (*fetch*), che dura il tempo T1 pari ad un periodo completo del segnale di clock; la seconda fase è la decodifica ed esecuzione dell'istruzione, che impiega il tempo T2 coincidente anch'esso con un periodo di clock. Pertanto, per ogni istruzione il tempo impegnato dalla CPU vale complessivamente due periodi di clock. Nella realtà la nostra CPU è un pochino più sofisticata e permette, durante la fase di esecuzione di un'istruzione, di prelevare contemporaneamente il codice dell'istruzione successiva: grazie a questo doppio lavoro, il tempo effettivo di elaborazione di ogni istruzione viene dimezzato e va a coincidere esattamente con un periodo di clock.



Ma quanto vale questo periodo di clock? I microcontrollori ATMEGA permettono di scegliere la sorgente del segnale di clock tra diverse sorgenti, attraverso l'impostazione di alcuni *fuse bits* (CKSEL<3:0> come visto nella scorsa puntata) che impostiamo in fase di programmazione del microcontrollore: nei nostri esercizi, nella riga di comando di avrdude il valore dei fuse bit è dato dai parametri:

```
-U hfuse:w:0x99:m -U lfuse:w:0x2E:m
```

I bit CKSEL<3:0> sono i quattro meno significativi di lfuse, pertanto valgono "1110": in base al datasheet (paragrafo "Clock Sources"), con questi valori selezioniamo l'oscillatore collegato al quarzo esterno Q1. Poiché nella demoboard tale quarzo è da 16MHz, il periodo di clock vale 1/16.000.000 = 0,0625µs e questo è dunque il tempo che la CPU impiega per eseguire ogni singola istruzione. Naturalmente qui parliamo di istruzioni del processore in *linguaggio macchina*, quelle "a basso livello" per intenderci, e non a quelle che noi scriviamo nel codice sorgente in linguaggio C: una riga di codice C può essere tradotta dal compilatore in diverse istruzioni in linguaggio macchina dunque il tempo di esecuzione aumenta in proporzione. Nota: in realtà anche alcune istruzioni a basso livello possono durare più cicli di clock: per conoscerne il numero bisogna consultare il manuale "Instruction set" del processore.

Lanciamo quindi il comando Build (da menu: Build → Build, oppure da tastiera con F7, oppure ancora dall'icona relativa sulla barra degli strumenti): la finestra dei messaggi (in basso) si auto-posiziona sulla linguetta "Build" per mostrare i processi di compilazione e, se tutto 'va bene', apparirà infine la fatidica scritta:

```
Build succeeded with 1 Warnings...
```

Se andiamo a indagare scorrendo all'indietro la finestra dei messaggi scopriremo che il "warning" riportato, come già visto, è relativo al fatto che la funzione `main()` non ritorna 'int' ma 'void': funzionalmente non cambia nulla quindi possiamo ignorarlo.
Importante: controllate che fra i messaggi vi sia anche la scritta "`Device: atmega16`", in quanto può succedere che chiudendo e riaprendo il progetto, AVR Studio decida "autonomamente" di modificare il micro selezionato (che deve essere l'ATMEGA16, altrimenti i programmi non potranno funzionare quando li caricheremo sul micro vero e proprio); in caso contrario bisogna andarlo a selezionare nuovamente nella finestra di **Fig. 5**.
Completata con successo la compilazione, troverete che è stato creato all'interno della cartella "C:\AVR\ControlloLED\default" il file "ControlloLED.hex" che contiene il *codice oggetto* (ovvero le istruzioni del nostro programma tradotte in istruzioni nel *linguaggio* della CPU dell'ATMEGA16) da scaricare nella memoria programma del microcontrollore: per fare questo trasferimento (file su PC → memoria µC) ci serviremo del programmatore. Come avevamo già visto, il programma che utilizziamo per gestire le fasi di programmazione si chiama "*avrdude*", e funziona da linea di comando (non considereremo la variante grafica *avrdudeGUI* che pure è utilizzabile): conviene pertanto creare un *file batch* che esegua automaticamente le operazioni richieste ad *avrdude* in modo da evitare di immetterle manualmente tutte le volte. Per farlo, con "Esplora risorse" di Windows ci spostiamo nella cartella "C:\AVR\ControlloLED\default", clicchiamo col tasto destro del mouse e selezioniamo "Nuovo → Documento di testo" (**Fig. 6**).
Chiamiamo questo documento "ControlloLED.bat": Windows informa che modificando

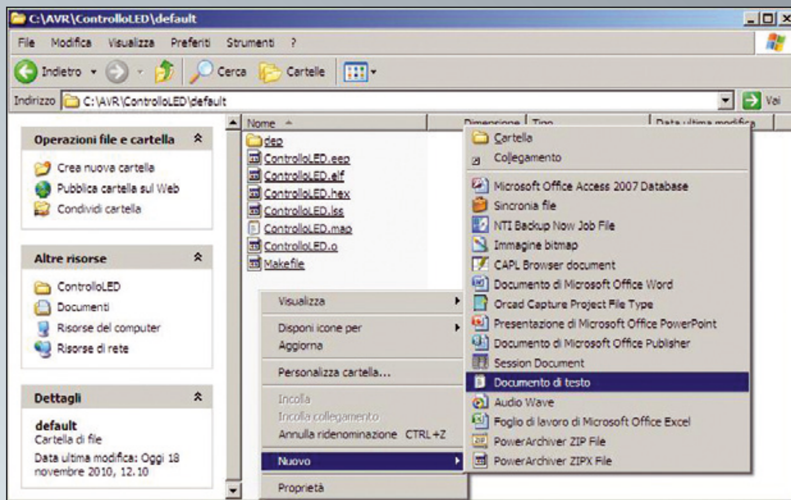


Fig. 6

l'estensione del file esso potrebbe diventare inutilizzabile, ignoriamo l'avviso e confermiamo la scelta. Editiamo quindi questo file cliccandovi sopra col tasto destro e selezionando la voce "Modifica" (siccome è un file con estensione "*bat*", se vi cliccassimo con il tasto sinistro anziché editarlo lo manderemmo in esecuzione). Si apre un blocco note: andiamo a scrivervi (su un'unica riga):

```
avrdude -p m16 -c bsd -P lpt1 -u
-U flash:w:ControlloLED.hex -U
hfuse:w:0x99:m -U lfuse:w:0x2E:m
```

Salviamo infine il file, dopodiché possiamo chiuderlo.
A proposito del parametro "-P lpt1", se la porta parallela utilizzata non è la LPT1 bisognerà sostituire LPT1 con la porta corretta.
Apriamo ora un terminale (Prompt dei comandi) di Windows e posizioniamoci sulla cartella contenente il file batch appena creato digitando:

```
CD \AVR\ControlloLED\
default <invio>
```

Possiamo quindi effettuare la programmazione del microcontrollore scrivendo nel terminale:

```
ControlloLED <invio>
```

Il file batch passa in esecuzione, invocando a sua volta avrdude (con i parametri specificati), il quale esegue tutte le fasi richieste dalla programmazione; il loro esito viene riportato sul terminale, come mostrato in **Fig. 7**, sulla quale sono state chiaramente evidenziate. Possiamo osservare che la prima operazione compiuta da avrdude è l'interrogazione del microcontrollore per verificare che esso sia un ATMEGA16, dunque compatibile col parametro 'm16' che avevamo specificato all'inizio. La riga in cui compaiono numerosi caratteri cancelletto '#' serve a simulare una barra di progressione per evidenziare l'avanzamento dell'operazione in corso. Di fatto, l'identificazione del microcontrollore è molto rapida, di conseguenza si nota solo la riga ormai completa, con a fianco la percentuale di completamento 100% ed il tempo (approssimativo) effettivamente impiegato.
Segue l'operazione di cancellazione della memoria FLASH, indispensabile in quanto essa potrebbe già contenere un programma diverso dal nostro (non dimentichiamo che tale memoria si può riscrivere fino a 10.000 volte!), che va preliminarmente eliminato.

Prompt dei comandi

```
C:\AVR\ControlloLED\default>avrdude -p m16 -c bsd -P lpt1 -u -U flash:w:Controll
oLED.hex -U hfuse:w:0x99:m -U lfuse:w:0x2E:m

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.00s

avrdude: Device signature = 0x1e9403
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "ControlloLED.hex"
avrdude: input file ControlloLED.hex auto detected as Intel Hex
avrdude: writing flash (194 bytes):

Writing | ################################################## | 100% 0.07s

avrdude: 194 bytes of flash written
avrdude: verifying flash memory against ControlloLED.hex:
avrdude: load data flash data from input file ControlloLED.hex:
avrdude: input file ControlloLED.hex auto detected as Intel Hex
avrdude: input file ControlloLED.hex contains 194 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.06s

avrdude: verifying ...
avrdude: 194 bytes of flash verified
avrdude: reading input file "0x99"
avrdude: writing hfuse (1 bytes):

Writing | ################################################## | 100% 0.00s

avrdude: 1 bytes of hfuse written
avrdude: verifying hfuse memory against 0x99:
avrdude: load data hfuse data from input file 0x99:
avrdude: input file 0x99 contains 1 bytes
avrdude: reading on-chip hfuse data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 1 bytes of hfuse verified
avrdude: reading input file "0x2E"
avrdude: writing lfuse (1 bytes):

Writing | ################################################## | 100% 0.00s

avrdude: 1 bytes of lfuse written
avrdude: verifying lfuse memory against 0x2E:
avrdude: load data lfuse data from input file 0x2E:
avrdude: input file 0x2E contains 1 bytes
avrdude: reading on-chip lfuse data:

Reading | ################################################## | 100% 0.00s

avrdude: verifying ...
avrdude: 1 bytes of lfuse verified

avrdude done.  Thank you.

C:\AVR\ControlloLED\default>_
```

Identificazione micro

Cancellazione FLASH

Scrittura FLASH del fil "ControlloLED.hex"

Verifica scrittura

Scrittura e verifica del byte "hfuse"

Scrittura e verifica del byte "lfuse"

Fig. 7

Fatto ciò, *avrdude* va a leggere il file oggetto (ControlloLED.hex) presente sul PC e lo "riversa" byte per byte sulla FLASH del micro. La fase successiva è la lettura della FLASH stessa per verificare che contenga esattamente il programma che vi avevamo appena scritto; questa fase non è strettamente indispensabile ma caldamente consigliata, in quanto se a causa di un disturbo durante la scrittura viene alterato anche solo un bit della FLASH, questo significa cambiare un'istruzione o un dato del programma, che non funzionerà più. In tal caso, eseguendo la verifica, *avrdude* potrà comunicare che la programmazione non è andata a buon fine; l'utente può allora provare a ripetere l'operazione eseguendo nuovamente il file batch "ControlloLED.bat"; se l'errore persiste, può darsi che la memoria del micro sia danneggiata ed in tal caso occorre sostituire il dispositivo. Se però è la prima volta che si effettua l'operazione di programmazione, conviene prima verificare la correttezza dei collegamenti tra micro e programmatore.
Le ultime operazioni sono la scrittura (e la verifica) dei "fuse bits". Notare che se avrdude arriva a queste fasi, vuol dire che le operazioni precedenti sono andate a buon fine, altrimenti s'interrompe al primo errore. Pertanto per conoscere l'esito di tutto il processo di programmazione è sufficiente verificare che si arrivi alla programmazione (e verifica) di *lfuse*. La frase di cortesia finale "avrdude done. Thank you" compare invece sempre (anche in caso di errore!).
In ogni caso, se tutto è andato bene, al termine della programmazione avrdude fa partire immediatamente il nuovo programma, quindi vedrete il led LD9 accendersi: in tal caso congratulazioni, il programma funziona!

NUOVO ESPERIMENTO

Possiamo iniziare a complicare il programma, visto che siamo in grado di comandare l'accensione del led: facciamolo lampeggiare! Per fare ciò, basta ovviamente mandare consecutivamente a '1' e a '0' il piedino PD7; modifichiamo pertanto il programma sorgente di **Fig. 3** come mostrato nel **Listato 1**.
Molto semplicemente, dopo aver mandato PD7 a '1' vi è subito l'istruzione che lo riporta a '0'; poiché entrambe le istruzioni sono all'interno del ciclo 'infinito' while(1), esse verranno ripetute indefinitamente, ottenendo su PD7 la sequenza 0-1-0-1-0-1...
Compiliamo il programma con AVR Studio e quindi riprogrammiamo il micro chiamando nuovamente "ControlloLED.bat" dalla console del prompt dei comandi: avendolo già digitato prima, basta premere il tasto "freccia su" per far ricomparire il comando, e premere quindi il tasto Invio.
Dopo che *avrdude* ha compiuto il suo lavoro, possiamo osservare LD9: esso apparirà probabilmente ancora come se fosse costantemente acceso, dandoci l'impressione che il programma non stia funzionando correttamente. In realtà non è così: il programma funziona ed il LED lampeggia, però ad una velocità tale che i nostri occhi non sono in grado di distinguere tra l'istante nel quale il led è acceso e quando è spento. Infatti il microcontrollore impiega pochissimo tempo a decodificare ed eseguire ogni singola istruzione (tempo dell'ordine dei microsecondi, si veda il riquadro "La velocità della CPU") per cui possiamo pensare che il nostro LED si accenda e si spenga all'incirca un milione di volte al secondo: difficile da percepire, non vi pare? L'unico indizio osservabile del suo stato *on-off* è una luminosità leggermente inferiore a quella che aveva quando era acceso in modo continuativo, poiché il nostro occhio fa "la media" tra la luminosità che caratterizza lo stato acceso e lo stato spento.

Listato 1

```
/*********** Programmino per far lampeggiare LD9 **********/
#include <avr/io.h>
void main(void) {
/* Inizializzazione */
        DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
        DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
        DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* Fine inizializzazione */
        while (1) { /* ciclo infinito */
                PORTD = 0x80; /* scrive '1' su PD<7> => LED acceso */
                PORTD = 0x00; /* scrive '0' su PD<7> => LED spento */
        }
}
```

Listato 2

```
/*********** Programmino per far lampeggiare LD9 **********/
#include <avr/io.h>
void main(void) {
        uint32_t ContaRitardo; /* variabile di conteggio nei cicli for */
/* Inizializzazione */
        DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
        DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
        DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* Fine inizializzazione */
        while (1) { /* ciclo infinito */
                PORTD = 0x80; /* scrive '1' su PD<7> => LED acceso */
                for (ContaRitardo = 0; ContaRitardo < 300000; ContaRitardo ++)
                        ;
                PORTD = 0x00; /* scrive '0' su PD<7> => LED spento */
                for (ContaRitardo = 0; ContaRitardo < 300000; ContaRitardo ++)
                        ;
        }
}
```

Per poter osservare una corretta sequenza di accensioni e spegnimenti del led dobbiamo rallentare notevolmente la sequenza 0-1 su PD7, quindi abbiamo due possibilità:

1) Rallentiamo il clock della CPU del microcontrollore;
2) Gli facciamo "perdere tempo" tra l'esecuzione delle istruzioni di accensione e di spegnimento.

L'ipotesi 1) è irrealistica, poiché se utilizziamo l'oscillatore interno il suo clock più basso è comunque 1 MHz, che corrisponde ad una velocità di esecuzione potenziale di un milione di istruzioni al secondo ed inoltre verrebbero penalizzati i tempi di esecuzione di tutte le istruzioni presenti nel programma (oltre a quelle utilizzate per il lampeggio del LED).
Pertanto rimane l'opzione 2): dopo aver acceso il LED facciamo in modo che la CPU "ritardi un certo tempo X" prima di eseguire l'istruzione che lo spegne; analogamente, dopo averlo spento deve attendere un altro tempo X prima di riaccenderlo. Se ad esempio X = 0,5 secondi, otterremo che il LED sta acceso per mezzo secondo e spento per un altro mezzo secondo, dunque lo vedremo lampeggiare a 1 Hz.
Per ritardare il processore gli facciamo eseguire altre istruzioni, poiché sappiamo che per ognuna impiega un certo tempo, per quanto piccolo: così se un'istruzione dura 1 µs, se gliela facciamo eseguire 500.000 volte otteniamo il ritardo di mezzo secondo. Naturalmente non dobbiamo scriverla 500.000 volte nel programma ma utilizziamo *un ciclo*, ad esempio il "for", nella seguente maniera:

```
for (ContaRitardo = 0; ContaRitardo
< VALORE_MASSIMO; ContaRitardo ++);
```

La spiegazione è semplice: la variabile "ContaRitardo" viene innanzitutto inizializzata a zero, e poi incrementata fino a quando raggiunge il numero VALORE_MASSIMO (che nel programma verrà associato a un valore preciso). Per una spiegazione esaustiva del funzionamento dei cicli 'for' del C si può naturalmente far riferimento ad Internet, dove la documentazione è sovrabbondante; a noi basta sapere che questa riga di codice C si traduce in un certo numero di istruzioni per la CPU, che essa deve ripetere VALORE_MASSIMO volte per realizzare appunto l'incremento della variabile "ContaRitardo" da zero a VALORE_MASSIMO.
Per provare in pratica questi concetti, provate a modificare il programma come mostrato nel **Listato 2**.
In tale listato abbiamo inserito i cicli 'for' dopo le istruzioni di accensione e spegnimento del LED, sostituendo 300.000 al valore VALORE_MASSIMO: quindi in ciascun ciclo la variabile "ContaRitardo" viene incrementata da 0 a 300.000, causando un ritardo complessivo di esecuzione di circa mezzo secondo. Verificatelo dopo una nuova compilazione del programma e programmazione del micro: il LED lampeggerà a circa 1 Hz. A proposito di "ContaRitardo", essa è stata dichiarata all'inizio del programma come `uint32_t`, ovvero variabile intera senza segno (**uint** significa **u**nsigned **int**eger) a 32 bit, poiché per arrivare a 300.000 non poteva essere né da 8 bit (valore massimo 255) e nemmeno da 16 bit (massimo 65535).
Naturalmente potete provare a variare i numeri nei cicli 'for' per verificare come cambiano i tempi di accensione e spegnimento del LED in relazione ai loro valori.
Nota: a differenza del linguaggio ANSI C standard, la libreria del compilatore avr-gcc (chiamata *avr-libc*) ha una notazione

Listato 3

```
void FunzioneRitardo(void) /* dichiarazione della funzione */
{
/* corpo della funzione */
     uint32_t ContaRitardo; /* variabile di conteggio */
     for (ContaRitardo = 0; ContaRitardo < 300000; ContaRitardo ++)
;
} /* fine funzione */
```

molto semplice per indicare il tipo di variabili intere e la loro grandezza: "int**N**_t", dove **N** va sostituito con la dimensione voluta (8, 16, 32, 64). Se la variabile è senza segno (non codificata in complemento a 2), bisogna far precedere il nome con una 'u' (ad esempio uint8_t è una variabile da 8 bit senza segno). Per maggiori dettagli, consultate la guida della libreria avr-libc contenuta nel documento "avr-libc-user-manual.pdf" in "C:\WinAVR-20100110\doc\avr-libc".

LE FUNZIONI

Su questo programma, seppure molto semplice, possiamo già introdurre alcune migliorie, sia a livello di leggibilità che di efficienza. Abbiamo visto che per rendere visualizzabile il lampeggio del LED dobbiamo scrivere nel sorgente due identici cicli 'for' di ritardo in due punti diversi del programma, con la conseguenza che il codice oggetto che il compilatore genera per il singolo ciclo 'for' verrà anch'esso duplicato nel file eseguibile del microcontrollore, con il risultato di una maggiore occupazione della memoria FLASH che dovrà contenerlo. Per evitare questo "spreco" di memoria si potrebbe cercare di raggruppare il codice del ciclo 'for' in un'area separata, dalla quale possa venire richiamato (dal programma principale) ogni volta che ve ne sia la necessità: il funzionamento complessivo non cambia, ma in questo modo il codice del ciclo 'for' viene scritto una sola volta nel file oggetto e quindi in memoria, con conseguente risparmio di spazio. Questo meccanismo è detto *chiamata a funzione*, dove per "funzione" s'intende la parte di codice relativa al ciclo 'for', che viene "chiamata" (ovvero eseguita) dal programma principale (che in C è la nostra funzione `main()`) quando e quante volte ce n'è bisogno.
La nostra nuova funzione va scritta come riportato nel **Listato 3**; il nome "FunzioneRitardo" è arbitrario, si può dare quello che si preferisce, basta che non contenga spazi: per esempio il nome "Pippo" va bene, "Funzione ritardo" no.
La sintassi della dichiarazione è simile a quella della funzione `main()`: il `void` tra parentesi indica che la funzione non riceve parametri dal chiamante mentre il `void` prima del nome indica che, quando termina, non restituisce nulla al chiamante. Nel corpo della funzione abbiamo riportato il codice del ciclo 'for', nonché la dichiarazione della variabile "ContaRitardo": se l'avessimo lasciata nel `main()` non sarebbe stata visibile all'interno di questa funzione ed il compilatore avrebbe segnalato l'errore; inoltre, nel `main()` questa variabile non viene utilizzata per null'altro.
Per chiamare la nuova funzione dal nostro programma principale dobbiamo scrivere, nei punti in cui vogliamo che venga eseguita:
`FunzioneRitardo();`
Quando il programma "incontra" questa istruzione (che è in realtà la *chiamata* alla funzione), passa ad eseguire il codice contenuto nel corpo della funzione stessa; al termine, il processore torna ad eseguire il codice del programma principale dal punto immediatamente successivo alla chiamata della funzione.
Il **Listato 4** riporta il codice completo del programma modificato con l'adozione della nuova funzione: in rosso sono visibili le righe relative alle modifiche introdotte.
Notate che prima dell'inizio della funzione `main()` è stata inserita una riga con la dichiarazione della funzione (detta *prototipo*) che verrà poi utilizzata nel corso del programma: questo serve per dire al compilatore, quando incontra nel `main()` le chiamate a `FunzioneRitardo()`, che esiste una funzione con tale nome, il cui codice tuttavia viene 'incontrato' dopo: infatti l'abbiamo scritto, nel file sorgente, dopo la funzione `main()`.
Si può notare come ne risulti un codice sorgente più "snello" e leggibile: una qualsiasi funzione, anche complessa, viene sostituita nella funzione principale `main()` con una sola riga di chiamata, in tutti i punti necessari, e magari chiamando la funzione stessa con un nome esplicativo del suo scopo.
L'altro vantaggio, come già detto, è una riduzione delle dimensioni del codice oggetto prodotto dal compilatore, tanto maggiore quante più volte la funzione dev'essere utilizzata dal programma principale.

Listato 4

```
/*********** Programmino per far lampeggiare LD9 **********/
#include <avr/io.h>
void FunzioneRitardo(void); /* prototipo della funzione di ritardo */
void main(void) { /* funzione principale */
/* Inizializzazione */
        DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
        DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
        DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* Fine inizializzazione */

  while (1) {
        /* ciclo infinito */
        PORTD = 0x80; /* scrive '1' su PD<7> => LED acceso */
        FunzioneRitardo(); /* prima chiamata alla funzione di ritardo */
        PORTD = 0x00; /* scrive '0' su PD<7> => LED spento */
        FunzioneRitardo(); /* seconda chiamata alla funzione di ritardo */
  }
}
/* fine funzione principale main()*/

void FunzioneRitardo(void) { /* codice della funzione di ritardo */
        uint32_t ContaRitardo;
        for (ContaRitardo = 0; ContaRitardo < 300000; ContaRitardo ++)
                ;
}
```

DEFINIZIONE DI COSTANTI

Abbiamo visto che la velocità del lampeggio varia cambiando il numero massimo di conteggi del ciclo 'for'. Tale numero si trova 'annidato' dentro al programma; la prima impressione è che non sia di immediata reperibilità, specialmente se vi trovaste a doverlo modificare dopo qualche mese da quando l'avete scritto. Per migliorare la leggibilità, in C è possibile definire i valori costanti utilizzati dal programma direttamente all'inizio del programma stesso utilizzando la seguente sintassi:

```
#define RITARDO_CICLO 300000
```

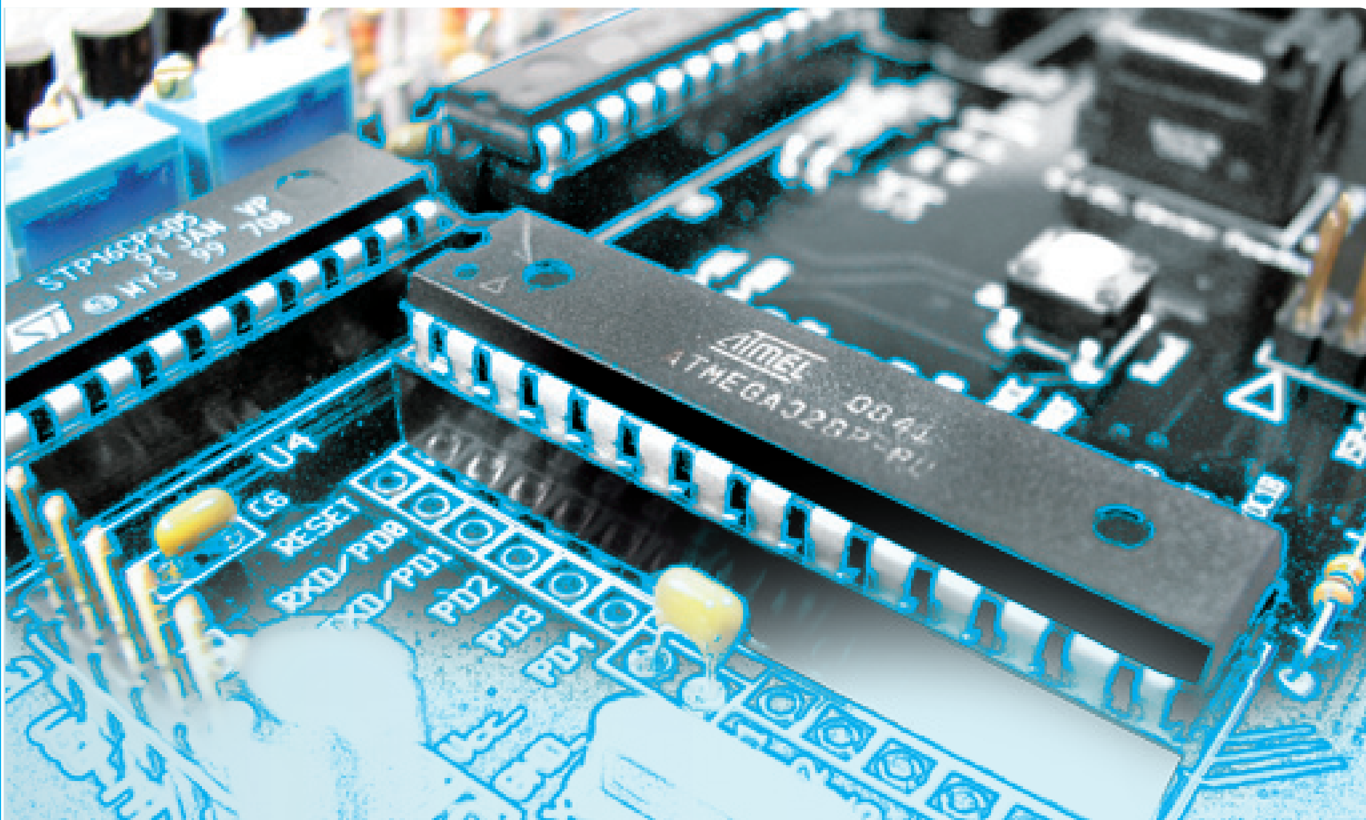
La riga definisce una costante chiamata "RITARDO_CICLO" alla quale abbiamo assegnato il valore 300.000. Se la scriviamo in cima al nostro sorgente (prima della funzione `main()` per intenderci), essa potrà in seguito essere utilizzata ovunque nel programma, così ad esempio nel ciclo 'for' potremo scrivere:

```
for (ContaRitardo = 0; ContaRitardo
< RITARDO_CICLO; ContaRitardo ++);
```

senza alterare il funzionamento del programma: il compilatore (più esattamente il preprocessore) provvederà a sostituire RITARDO_CICLO con 300.000 ovunque essa venga richiamata. Il vantaggio di questa soluzione è che avendo definito il numero in una riga ad inizio programma, è più facile da trovare e modificare a piacere, soprattutto se nel programma la costante in questione viene usata più volte (nel nostro listato 2 ad esempio era usata due volte, per i due cicli 'for'), lasceremo così al compilatore il compito di trovare ed operare le sostituzioni per tutte le occorrenze della costante all'interno del programma. Da osservare che, nel linguaggio C, i nomi che fanno riferimento a delle costanti vengono scritti utilizzando esclusivamente lettere maiuscole: nel nostro caso abbiamo infatti "RITARDO_CICLO" e non "Ritardo_Ciclo" oppure "ritardo_ciclo". Questa non è una regola del linguaggio (il compilatore non dà errore scrivendoli minuscoli) ma una convenzione universalmente riconosciuta, allo scopo di rendere le costanti più evidenti all'interno del programma a vantaggio della leggibilità.

Per completezza è giusto accennare al fatto che `#define` è una *direttiva* del preprocessore del compilatore C che permette, oltre alla semplice dichiarazione di costanti, di definire vere e proprie *macro*, ovvero intere righe di codice che poi vengono sostituite nel programma in modo analogo a quanto abbiamo visto per le costanti; rimandiamo alla documentazione sul linguaggio C per un eventuale approfondimento.

Terminiamo quindi questa puntata pratica sul microcontrollore nella quale abbiamo visto come gestire lo stato di un piedino di I/O del microcontrollore, in questo caso per accendere un LED, e l'utilizzo delle funzioni in un programma. Dalla prossima lezione passeremo ad affrontare lo sviluppo di applicazioni più complesse grazie anche all'utilizzo delle periferiche interne dell'ATMEGA16 che inizieremo ad esplorare.Vi aspettiamo!



OPEN SOURCE

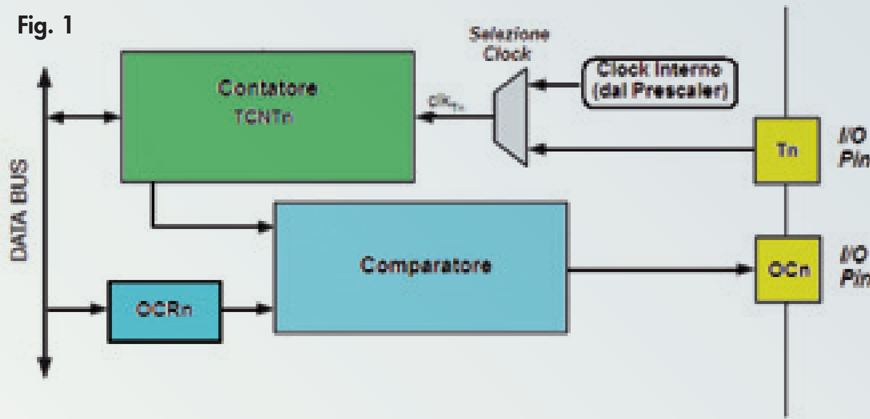
dell'ing. OSVALDO SANDOLETTI

Iniziamo l'esplorazione delle periferiche interne dell'ATMEGA16 analizzando quelle dedicate al conteggio di tempi ed eventi, per capire quali sono i loro compiti e come possano essere utilizzate all'interno di un programma. Sesta puntata.

Iniziamo in questa puntata l'esplorazione delle periferiche interne dell'ATMEGA16 analizzando quelle dedicate al conteggio di tempi ed eventi, per capire quali sono i loro compiti e come possano essere utilizzate all'interno di un programma.
Nella descrizione delle caratteristiche del microcontrollore ATMEGA16, nella prima puntata del corso, avevamo brevemente riportato la lista delle numerose periferiche che esso contiene al suo interno, oltre alla CPU e alle memorie. Tra queste vi sono i ***Timer-Counter*** (abbreviati anche in T/C nel seguito), ovvero circuiti contatori in grado appunto di contare il tempo (come un cronometro) oppure sequenze di '0' e '1' applicate ai loro ingressi. Ad oggi, praticamente ogni microcontrollore sul mercato annovera almeno un timer/counter tra le proprie periferiche; l'ATMEGA16 ne ha 3 distinti (numerati da 0 a 2: T/C0, T/C1, T/C2), e ciascuno può essere programmato per eseguire una delle seguenti funzioni.

• Conteggi di tempo: sull'ingresso del contatore è applicato un segnale di clock (derivato

Fig. 1



da quello della CPU o da un oscillatore) per cui l'uscita del contatore riporta un numero sempre crescente e proporzionale al tempo "che trascorre". Il contatore può anche essere azzerato, od inizializzato a un preciso numero, dal programma. Con questa modalità si può realizzare un cronometro;

- Conteggio di eventi: l'ingresso del contatore è collegato ad un piedino di I/O del micro, per cui l'utente gestisce il conteggio dall'esterno: il contatore incrementa (oppure decrementa) di una unità ad ogni transizione 0 → 1 oppure 1 → 0 applicata al piedino. Con questa modalità si possono realizzare ad esempio dei conta-pezzi;
- Generazione di eventi: l'uscita del contatore (che può essere collegata ad un pin di I/O del micro) può essere configurata per cambiare il suo stato (da '0' a '1' o viceversa) quando il conteggio raggiunge un determinato valore, anch'esso impostabile da programma: con questa modalità si possono realizzare dei temporizzatori.

In **Fig. 1** possiamo vedere lo schema a blocchi semplificato di ciascuno dei T/C, dove al suffisso 'n' delle varie definizioni bisogna sostituire nella pratica il numero 0, 1 oppure 2, a seconda che si consideri rispettivamente il timer/counter 0, 1 oppure 2. Sulla destra della figura vi è il selettore della sorgente di clock del contatore, che può provenire da un piedino di I/O esterno (T*n*) oppure dal clock della CPU (attraverso un "*Prescaler*" tramite il quale possiamo dividere la frequenza di un certo fattore impostabile da programma).
Il segnale così ricavato (detto **clk**$_{Tn}$) è utilizzato per pilotare il contatore vero e proprio, chiamato **TCNT*n***. Esso può essere programmato per venire incrementato oppure decrementato ad ogni transizione del segnale **clk**$_{Tn}$ ed è collegato al data-bus del microcontrollore: può pertanto in qualsiasi momento essere letto (oppure anche scritto) dal programma. Per esempio, se vogliamo leggere il valore del contatore TCNT0 (ovvero quello relativo al T/C0) e salvarlo nella variabile "Lettura" scriveremo la seguente espressione:

```
Lettura = TCNT0;
```

Oltre alla lettura, nei contatori TCNT*n* possiamo anche scrivervi un qualsiasi valore, dal quale inizierà il conteggio; ad esempio per caricare in TCNT2 il valore zero scriviamo:

```
TCNT2 = 0x00;
```

Se in seguito il contatore TCNT2 riceverà un segnale di clock al suo ingresso, comincerà a contarlo a partire da zero: 1, 2, 3, 4, ...

Nota: nell'ATMEGA16 i contatori TCNT0 e TCNT2 sono da 8 bit mentre TCNT1 è a 16 bit; questo significa che per i primi due il numero massimo raggiungibile è 0xFF (255 in decimale), mentre in TCNT1 è 0xFFFF (65535). Per tutti, il ciclo di clock successivo al raggiungimento del numero massimo causa l'azzeramento del contatore che dunque riparte a contare da 0x00, generando un segnale cosiddetto "*Timer Overflow*" del timer/counter *n* (TOV*n*). Tale segnale è un bit del registro TIFR (*Timer Interrupt Flag Register*, sul quale torneremo più avanti): normalmente è a '0', e se va ad '1' significa che vi è stato un timer overflow; il programma può dunque andare a leggerne lo stato in modo da essere a conoscenza che un contatore "ha sforato" ed agire di conseguenza.
Definizione: i bit che segnalano particolari eventi (come i TOV*n* che segnalano l'overflow dei rispettivi timers) vengono detti flag (bandiera in italiano). Generalmente essi sono posti a '1' per segnalare l'evento, e poi devono essere riportati manualmente a '0' dal programma.
Proseguendo nella descrizione dello schema della **Fig. 1** incontriamo un comparatore, che confronta continuamente il valore di TCNT*n* con il numero contenuto nel registro OCR*n*, che normalmente viene precaricato dal pro-

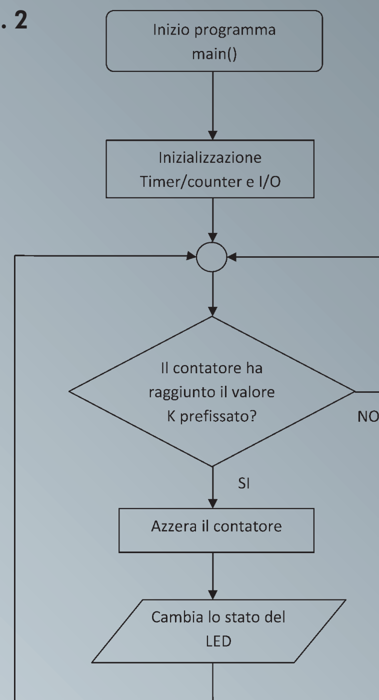
gramma con un valore voluto. Quando TCNT*n* raggiunge (durante il suo conteggio) tale numero, il comparatore "scatta" e segnala l'evento mandando ad '1' il rispettivo bit chiamato OCF*n* (*Output Compare Flag*) pure contenuto nel registro TIFR. Come si nota ancora nella **Fig. 1**, l'uscita del comparatore può anche venire indirizzata su un piedino di I/O (chiamato OC*n*: *Output Compare* del canale *n*) per poter controllare direttamente un circuito esterno al microcontrollore.
Sul datasheet troverete, nelle sezioni relative, gli schemi a blocchi e la descrizione dettagliata di ciascun timer/counter: potrete vedere che, sebbene come principio di funzionamento essi aderiscano tutti allo schema di **Fig. 1**, fra di loro vi sono alcune differenze che possono influenzare il progettista nella scelta di uno piuttosto che un altro. Per evidenziare brevemente queste differenze, riassumiamo la loro descrizione:

- Timer/counter 0: è composto essenzialmente dai blocchi funzionali di **Fig. 1**; il contatore (TCNT0) e il registro del comparatore (OCR0) sono a 8 bit;
- Timer/counter 1: ha due unità indipendenti di "Output Compare", ovvero contiene due registri OCR (chiamati OCR1A e OCR1B), due comparatori e due uscite OC1A e OC1B anch'esse indipendenti tra loro. Possiede anche un'unità di *Input Capture* (ICP) che permette di "fotografare" il contenuto del contatore TCNT1 nel momento in cui un particolare ingresso (collegato ad un pin esterno del microcontrollore) cambia di stato. Il valore catturato in quell'istante viene memorizzato in un registro chiamato ICR1, dal quale il programma può leggerlo. Da ricordare infine che tutti i registri del Timer/counter 1 (TCNT1, OCR1A, OCR1B, ICR1) sono a 16 bit;
- Timer/counter 2: è simile al Timer/counter 0, però l'ingresso di clock esterno è sostituito da un completo modulo oscillatore, cui si può collegare un quarzo esterno dal quale ricavarsi il segnale di temporizzazione indipendentemente dal clock di sistema. I suoi registri sono da 8 bit.

USIAMO I T/C

Per vedere concretamente i timer/counter (T/C) in azione, ne utilizzeremo uno per fare lampeggiare (ancora!) il nostro LED della demoboard.
La scorsa puntata avevamo visto che la soluzione per ottenere un lampeggio "visibile" era l'inserimento di un opportuno ritardo dopo ogni cambiamento di stato del LED; tale ritardo si era ottenuto facendo eseguire al processore, tramite un ciclo '`for`', un elevato numero di istruzioni "inutili".
Adesso vediamo invece come far contare questo ritardo da un T/C, liberando dall'incombenza il processore (che quindi può eseguire altri "incarichi" del programma).

Fig. 2



Per fare ciò utilizzeremo il T/C in modalità di conteggio di tempo: lo inizializziamo per far sì che esso conteggi il clock fornitogli dalla CPU e (sempre tramite programma) leggiamo continuamente il risultato del conteggio: quando viene raggiunto un certo valore 'K' (che poi stabiliremo), il programma cambia lo stato del LED ed azzera il contatore (per fare ripartire il conteggio da 0x00), e il ciclo si ripete.
Chi trovasse complicata la procedura può consolarsi sapendo che esiste una maniera molto più elegante ed intuitiva per descriverla: il *diagramma di flusso* (si veda il riquadro relativo per un approfondimento). In **Fig. 2** viene proposto il diagramma di flusso del nostro nuovo programma, relativo alla procedura sopra descritta (il nostro *algoritmo*).
Prima di iniziare la scrittura del programma (che costituisce l'*implementazione* dell'algoritmo) dobbiamo ancora decidere quale dei tre T/C messi a disposizione dal nostro micro possiamo utilizzare: teoricamente tutti hanno la funzionalità richiesta, però dobbiamo fare due conti per capire quale ritardo massimo possono garantire. Innanzitutto, qual è il segnale di clock che utilizziamo per il conteggio del timer? Siccome nella demoboard non vi sono sorgenti esterne, dobbiamo utilizzare il clock interno "di sistema" del microcontrollore

Fig. 3

	CS02	CS01	CS00	Description
Contatore fermo	0	0	0	No clock source (Timer/Counter stopped).
Clock interno	0	0	1	$clk_{I/O}$/(No prescaling)
	0	1	0	$clk_{I/O}$/8 (From prescaler)
	0	1	1	$clk_{I/O}$/64 (From prescaler)
	1	0	0	$clk_{I/O}$/256 (From prescaler)
	1	0	1	$clk_{I/O}$/1024 (From prescaler)
Clock esterno	1	1	0	External clock source on T0 pin. Clock on falling edge
	1	1	1	External clock source on T0 pin. Clock on rising edge.

che, come avevamo visto nella quarta puntata, può a sua volta essere selezionato da diverse sorgenti: scegliamo quella utilizzata anche nelle esercitazioni passate (quindi non dovremo neanche modificare i fuse-bits nelle impostazioni di avrdude), ovvero il quarzo esterno da 16MHz. Tale frequenza può essere divisa dal *prescaler* del timer/counter per un fattore selezionabile dai tre bit CS*n*<2:0> (ove n è sempre il numero del timer). Con tali bit si sceglie anche quale sorgente di clock utilizzare per il T/C: riportiamo per esempio, in **Fig. 3**, la tabellina della verità dei bit CS0<2:0> (quindi relativi al T/C0); questi bit sono contenuti nel registro TCCR0.
Se questi sono tutti e tre a zero, nessun segnale di clock viene inviato al contatore che resta pertanto fermo. Le combinazioni da 001 (1 decimale) a 101 (5 decimale) permettono di utilizzare il clock di sistema del microcontrollore (qui indicato come "$clk_{I/O}$"), diviso per il fattore mostrato in **Fig. 3**: nel nostro caso, poiché $clk_{I/O}$ = 16MHz, se CS0<2:0> = 001 non vi è divisione e il contatore riceve la frequenza di 16MHz; se CS0<2:0> = 101, il fattore di divisione del prescaler vale 1024 e il contatore incrementerà con una frequenza di 16 MHz / 1024 = 15625 Hz.
Le ultime due combinazioni dei bit CS0<2:0> impostano il timer/counter per prelevare il clock dal piedino esterno T0, e l'incremento del contatore avviene sulla transizione 1→0 (se CS0<2:0> = 110) oppure 0→1 (quando CS0<2:0> = 111) del segnale.
Noi come già detto utilizziamo il clock interno; poiché, come abbiamo visto, la frequenza minima ottenibile è 15625 Hz (usando il rapporto di divisione più elevato), il tempo di ogni impulso di clock che va ad incrementare il contatore sarà pari a 1/15625 = 64µs. Ma allora, se volessimo far lampeggiare il nostro LED a 1Hz, dobbiamo realizzare un ritardo di 500ms sia per la fase 'ON' che per quella 'OFF', e in quell'intervallo di tempo il contatore conterà (scusate il gioco di parole!) 500ms / 64 µs = 7812 impulsi: questo è il valore 'K' con cui il contatore va confrontato nel programma. Questo numero impone anche la scelta sul timer/counter da utilizzare: abbiamo visto che i T/C 0 e T/C 2 hanno contatori (TCNT0 e TCNT2 rispettivamente) da 8 bit, quindi il massimo numero raggiungibile è 255 e quindi non vanno bene: bisogna utilizzare necessariamente il T/C 1 in quanto il suo contatore (TCNT1) è da 16 bit e può contare fino a 65535.
Dal datasheet si evince che anche per questo timer vi sono tre bit di programmazione del prescaler, chiamati CS1<2:0> e contenuti nel registro TCCR1B; anche in questo caso il fattore di divisione massimo è 1024 (ponendo sempre CS1<2:0> = 101) quindi le considerazioni ed i valori calcolati sopra rimangono validi.
Riassumendo:

1) Utilizziamo il T/C 1, alimentato dal clock di sistema (16 MHz) diviso per 1024 (CS1<2:0> = 101): il contatore TCNT1 incrementa pertanto il conteggio ogni 64 microsecondi;
2) Quando TCNT1 raggiunge il numero 7812 significa che sono trascorsi 500 millisecondi, per cui il programma deve cambiare lo stato del LED (da acceso a spento o viceversa) e azzerare il contatore TCNT1;
3) Si aspetta che TCNT1 raggiunga nuovamente il numero 7812, e si torna al punto 2).

INIZIALIZZARE PRIMA DELL'USO

Come (quasi) tutte le periferiche di un microcontrollore, anche i timer/counters richiedono di essere *inizializzati* per divenire operativi, andando ad impostare il valore dei bit di specifici registri di controllo.
Datasheet alla mano, nel paragrafo "Register description" della sezione "16 bit Timer/Counter 1" troviamo la lista e la spiegazione dettagliata di tutti i registri di cui il T/C1 fa uso. I due registri che "contengono" i bit di controllo del T/C1 son TCCR1A e TCCR1B; la modalità di conteggio è stabilita ad esempio dal valore assegnato ai quattro bit WGM1<3:0> (suddivisi tra TCCR1A e TCCR1B) in base alla tabella "Waveform Generation Mode Bit Description"; in **Fig. 4** riportiamo una parte di tale tabella, evidenziando anche la posizione dei bit WGM1<3:0> nei registri TCCR1A e TCCR1B.
Si può notare che ad ogni combinazione di

questi bit corrisponde una diversa modalità di conteggio, e ve ne sono ben 16 in totale; siccome noi utilizziamo in quest'applicazione il T/C1 come normale contatore di impulsi a 16 bit dobbiamo impostarlo nel modo '0' (prima riga): infatti la colonna "Timer/counter Mode of operation" riporta la scritta "Normal", e la colonna "TOP" (che dice qual è il massimo numero cui arriva il contatore prima di azzerarsi) riporta 0xFFFF (65535). Sempre dalla tabella si desume che per ottenere il modo '0' dobbiamo impostare WGM1<3:0> = 0000.
Nel registro TCCR1A vi sono altri 6 bit, che però dobbiamo impostare a '0' in quanto servono ad attivare particolari funzioni dell'unità di *Output Compare* che noi non utilizziamo.
Il registro TCCR1B contiene due bit (ICNC1 e ICES1) relativi all'unità di Input Compare: li lasciamo a '0' in quanto anche questa funzionalità non viene utilizzata. TCCR1B contiene infine i tre bit CS1<2:0> di controllo del clock del contatore, che come discusso sopra dobbiamo impostare a 101.
I registri OCR1A, OCR1B, e ICR1 sono relativi alle funzionalità di output compare e input capture del modulo, e quindi non dobbiamo inizializzarli.
Il registro TIMSK serve ad abilitare gli interrupt legati al timer/counter, e vedremo in seguito a cosa serve; per il momento lasciamo disabilitate queste caratteristiche scrivendo '0' nei suoi bit.
Il registro TIFR contiene infine i flag del T/C1, ovvero dei bit che divenendo '1' segnalano il verificarsi di un evento legato alla periferica: il programma può quindi leggerli per sapere se l'evento è accaduto o meno.

Listato 1

```
/**************************************************************************
 *   Programma per far lampeggiare il LED LD9 della DEMOBOARD         *
 *    usando il TIMER1 in modalità NORMALE per il conteggio del tempo *
 **************************************************************************/
#include <avr/io.h>

void main(void) {

/* Inizializzazione */
         DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* impostazione T/C 1 in mod. normale, con prescaler 1024 */
         TCCR1A = 0;
         TCCR1B = (1 << CS12) | (1 << CS10);
         TIMSK = 0;

/* Fine inizializzazione */

         while (1) {   /* ciclo infinito */

                  if (TCNT1 == 7812) { /* se il contatore ha raggiunto
7812...*/
                           PORTD = PORTD ^ 0x80; /* complementa PD<7> */
                           TCNT1 = 0;     /* azzera contatore */
                  }
         }
}
```

SCRITTURA DEL PROGRAMMA

Avendo tutti i dati che ci occorrono passiamo finalmente alla scrittura del programma, riportata nel **Listato 1**. In AVR Studio potete decidere se sovrascrivere il file sorgente "ControlloLED.c" della precedente esercitazione, oppure creare un progetto ex-novo; noi preferiamo risparmiare tempo e sovrascrivere, tanto lo scopo del programma è il medesimo!
In tal modo potrete verificarne il funzionamento con la consueta procedura di compilazione e programmazione. Qui vorremmo soffermarci per commentare alcune righe del programma: nella sezione di inizializzazione dopo aver impostato la direzione dei quattro I/O port del micro (in base a quanto visto nelle puntate precedenti) ed aver forzato a zero il livello dei loro pin impostati come uscite, si passa a caricare i valori prima discussi su TCCR1A (cioè 0) e TCCR1B (cioè zero su tutti i bit tranne CS12 e CS10); per quest'ultimo scriviamo

```
TCCR1B = (1 << CS12) | (1 << CS10);
```

che corrisponde a caricare in TCCR1B il risultato dell'OR logico (che in C si scrive col carattere '`|`') tra il valore delle due parentesi: `(1 << CS12)` e `(1 << CS10)`; vediamo allora qual è tale valore.
Durante la compilazione, il preprocessore sostituisce preliminarmente il nome dei bit CS12 e CS10 con la loro rispettiva posizione (2 e 0) nel registro TCCR1B, che è definita (insieme a quelle degli altri registri delle periferiche) nell'header *io.h*; è pertanto equivalente a scrivere `(1 << 2)` e `(1 << 0)`.
Nel linguaggio C l'espressione "`N << X`" fa scorrere a sinistra i bit del numero N di X posizioni, quindi se N = 1 (in binario 0000 0001)

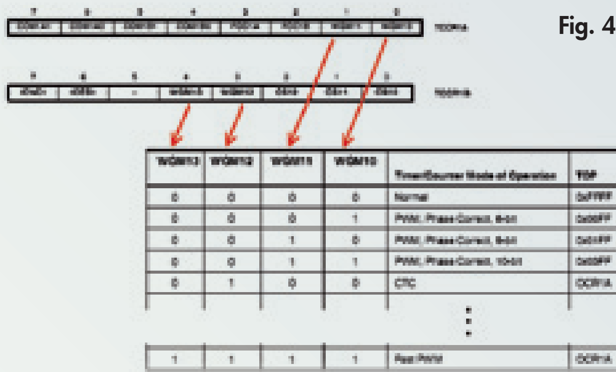


Fig. 4

otteniamo che:
1 << 0 = 1 (0000 0001)
1 << 1 = 2 (0000 0010)
1< < 2 = 4 (0000 0100)
e così via. Quindi i valori ricavati dalle due parentesi tonde sono:

```
(1 << CS10) = 0000 0001 << 0 → 0000 0001
(1 << CS12) = 0000 0001 << 2 → 0000 0100
```

che infine vengono messi in OR tra loro per ottenere il valore 0000 0101 (= 0x05) da caricare in TCCR1B.
Chiaramente avremmo potuto scrivere direttamente

```
TCCR1B = 0x05;
```

oppure, utilizzando la notazione binaria, anche

```
TCCR1B = 0b00000101;
```

ma usando il nome dei bit lasciamo al preprocessore del compilatore il compito di sostituirli con la rispettiva posizione all'interno del registro: il codice rimane più leggibile (scrivendo "`TCCR1B = 0x05;`" come faremmo a capire che sono stati messi a '1' i bit CS0 e CS2?), e a prova di errore!
Tornando al codice dell'inizializzazione, scriviamo 0 in TIMSK per assicurarci che tale sia il suo valore (quindi di tutti i suoi bit), anche se il datasheet afferma che è già lo stato in cui si trova il registro dopo il reset del processore (si veda, nella descrizione del registro, il valore dei bit della riga "Initial value"): riscriverlo *non fa mai male*.
Nel *main-loop* del programma facciamo semplicemente quanto previsto nell'algoritmo e riportato nel diagramma di flusso di **Fig. 2**: TCNT1 è continuamente confrontato col numero K (ovvero 7812), se uguale si aggiorna lo stato del LED e si azzera il contatore. Per complementare il LED usiamo l'operatore EX-OR *bit a bit* del C con la seguente istruzione:

```
PORTD = PORTD ^ 0x80;
```

siccome 0x80 vale in binario 1000 0000, il bit "più a sinistra" (il più significativo) di PORTD viene complementato: se era '0' diventa '1' e viceversa. Poiché tale bit comanda lo stato del piedino PD7 a cui è collegato il LED, otteniamo il lampeggio.

UN'ALTRA POSSIBILITÀ

Abbiamo visto che ogni timer/counter contiene oltre al contatore anche un'unità di comparazione (**Fig. 1**), ed il T/C1 in particolare ne ha addirittura due (canali A e B), possiamo demandare ad essa il compito del confronto del valore di TCNT1 con il numero 7812, anziché andarlo a fare manualmente nel programma. È infatti sufficiente caricare nel registro OCR1A (decidiamo di usare il canale A) il numero 7812 e, ogni volta che TCNT1 raggiunge tale valore, viene automaticamente alzato (va ad '1') il flag OCF1A del registro TIFR. Se inoltre impostiamo il T/C1 per lavorare nella modalità CTC (*Clear Timer on Compare*) che, come si vede dalla tabella di **Fig. 4** si ottiene impostando WGM1<3:0> = 0100 (quindi solo WGM12 a '1') , il contatore TCNT1 si azzera automaticamente quando raggiunge il valore di OCR1A, e pertanto non bisogna più farlo manualmente: il compito del programma nel main-loop rimane quello di aspettare che OCF1A vada a '1' per complementare il LED. Nel **Listato 2** mostriamo dunque il programma completo: si nota che il flag OCF1A del registro TIFR viene letto con l'istruzione:

```
if (TIFR & (1 << OCF1A))
```

ed il "corpo" dell'`if` (tra parentesi graffe) viene eseguito solo se tale flag vale '1'. In tal caso la prima istruzione è il ri-azzeramento di tale flag, necessario in quanto dobbiamo "prepararlo" per il prossimo ciclo: se rimanesse ad '1' non si potrebbe sapere quando il comparatore scatta di nuovo. Per molti flag dell'ATMEGA16 tra cui questo, in base alla descrizione sul datasheet, l'azzeramento si ottiene scrivendovi un '1' (strano, ma è così); nel programma utilizziamo l'operazione OR logico bit a bit del C:

```
TIFR |= (1 << OCF1A);
```

Listato 2

```
/*************************************************************************
 *   Programma per far lampeggiare il LED LD9 della DEMOBOARD            *
 *    usando il TIMER1 in modalità CTC per il conteggio del tempo        *
 *************************************************************************/

#include <avr/io.h>

void main(void) {

/* Inizializzazione */
DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */

/* impostazione T/C 1 in mod. CTC, con prescaler 1024 */
         TCCR1A = 0x00;
         TCCR1B = (1 << CS12) | (1 << CS10) | (1 << WGM12);
         OCR1A = 7812; /* costante di comparazione */
         TIMSK = 0;

/* Fine inizializzazione */

         while (1) {   /* ciclo infinito */

                  if (TIFR & (1 << OCF1A)) {    /* OCF1A è a 1 ? */
                           TIFR |= (1 << OCF1A);    /* allora riportalo a 0 */
                           PORTD = PORTD ^ 0x80;    /* e complementa PD<7> */
                  }
         }
}
```

Con tale istruzione scriviamo '1' nel bit OCF1A del registro TIFR, lasciando invariati gli altri bit del registro; l'effetto è che OCF1A viene rimesso a zero.

UTILIZZARE LE... INTERRUZIONI

Il compito del *main-loop* del **Listato 2** è quello di verificare ininterrottamente lo stato del flag OCF1A che (avendo caricato OCR1A con 7812) sappiamo andare ad '1' ogni 500ms. Supponiamo che, quando ciò accade, la CPU impieghi quindi 10 microsecondi per azzerare il flag e complementare il pin PD7: allora per i successivi 500.000 – 10 = 490.000 microsecondi la CPU continuerà a controllare (inutilmente) il flag in attesa che torni '1', senza poter eseguire eventuali altre operazioni.

Nota: l'operazione di lettura ciclica dello stato di una periferica (per verificare l'accadere o meno di un evento) si definisce polling.

Pertanto sarebbe molto più efficiente un meccanismo "automatico" che verifichi il cambiamento di stato del flag e lo comunichi alla CPU (che nel frattempo sta eseguendo altri compiti) solo quando ciò accade: in tal caso la CPU interrompe temporaneamente l'esecuzione di quello che stava facendo e va ad eseguire il lavoro richiesto da questo evento, ovvero il complemento dell'uscita PD7; fatto ciò, riprende il compito che aveva interrotto. Questo meccanismo esiste, ed è basato sulle interruzioni (*interrupt*): una interruzione è un segnale (proveniente da una periferica quando accade un particolare evento, oppure dall'esterno) trasmesso alla CPU: quando lo riceve, essa blocca il programma che stava eseguendo in quel momento per andare ad eseguire una particolare funzione di gestione dell'interruzione, detta isr (*Interrupt Service Routine*). Al termine di questa funzione la CPU torna al programma che stava eseguendo nel momento dell'interruzione, riprendendo dal punto esatto in cui era stato interrotto: l'interruzione non ne deve alterare lo svolgimento, anche se ovviamente lo rallenta (la CPU non può eseguire due programmi contemporaneamente: mentre esegue la *isr*, il programma principale è fermo).

Nota: poiché un'interruzione può essere generata da un evento indipendente dal programma principale (può avvenire in un punto qualsiasi durante l'esecuzione dello stesso), diremo che la isr è una funzione asincrona, per differenziarla da una funzione normale (sincrona) che invece viene invocata dal programma principale stesso in un punto preciso.

Esemplifichiamo il concetto delle interruzioni applicandolo al nostro caso: il timer/counter 1 del micro ATMEGA16 permette di inviare segnali di interruzione alla CPU al verificarsi dei seguenti eventi:

- Si è alzato il flag TOV1 (dovuto all'overflow del contatore TCNT1);
- Si è alzato il flag OCF1A oppure il flag OCF1B poiché il relativo comparatore (A o B) è scattato (l'evento è detto pertanto *Output compare match*);
- Si è alzato il flag ICF1 (relativo ad un evento di Input Compare).

Come si vede, ciascuna interruzione che si può inviare alla CPU è legata allo stato di un flag della periferica. In alcuni microcontrollori vi è un solo segnale di interruzione globale che

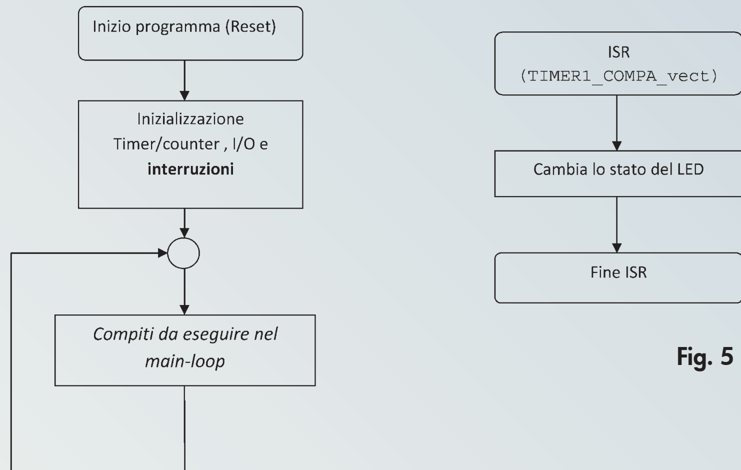


Fig. 5

"lancia" un'unica *isr*; all'interno della quale è compito del programma verificare quale flag è attivo per sapere qual è l'evento verificatosi. Nei micro ATMEGA invece ogni evento chiama una specifica isr; pertanto, a noi interessa implementare la isr relativa all'evento "Output compare match" del canale A, che viene chiamata in concomitanza dell'attivazione del flag OCF1A. Per far questo dobbiamo scrivere nel sorgente, prima o dopo la funzione main() ma comunque all'esterno di essa, il seguente codice:

```
ISR (TIMER1_COMPA_vect) {
      /* corpo della funzione */
}
```

La parola 'ISR' è una macro, che (semplificando) possiamo paragonare ad una parola chiave del compilatore avr-gcc per informarlo che quella che segue non è una funzione "normale" ma una isr, ovvero una funzione che il processore chiamerà automaticamente al ricevimento di un segnale d'interruzione. Tra parentesi bisogna inserire il nome preciso della isr che vogliamo invocare; l'ATMEGA16 possiede 20 diverse sorgenti d'interruzione (sommando quelle esterne e quelle legate alle periferiche interne), ognuna delle quali viene collegata ad un proprio indirizzo (detto vettore dell'interruzione) per chiamare una diversa isr. In avr-gcc questi vettori sono associati ad un nome: "`TIMER1_COMPA_vect`" è il nome del vettore associato all'interruzione che ci interessa gestire nel nostro programma (attivata con il flag OCF1A). La lista dei nomi dei vettori d'interruzione del compilatore avr-gcc si trova nel documento "avr-libc-user-manual" contenuta in <cartella installazione WinAVR>\doc, nella sezione "Interrupts". Tra le parentesi graffe della funzione (il suo corpo) si scrive ciò che vogliamo eseguire quando l'interruzione viene attivata. Possiamo adesso stendere il diagramma di flusso del nuovo programma, mostrato in **Fig. 5**.
A sinistra abbiamo il flusso del programma principale, che alla partenza esegue come di consueto tutte le inizializzazioni; inoltre deve *abilitare le interruzioni* che vogliamo vengano attivate, per le quali avremo cura di scrivere una isr di gestione. Per l'ATMEGA16 significa fare due cose:

1) Abilitare l'interruzione legata all'evento "Output compare match" del canale A del T/C 1: si tratta di mettere a '1' il bit OCIE1A (*Output Compare Interrupt Enable 1* del canale A) nel registro TIMSK (Timer/counter Interrupt Mask register). Si dice in gergo che così facendo "smascheriamo" l'interruzione, che in caso contrario non può essere generata;
2) Abilitare la CPU a ricevere le interruzioni attivando un bit di abilitazione "globale"; in *avr-gcc* lo otteniamo scrivendo l'espressione: "`sei();`". Se nel corso del programma volessimo in seguito disabilitare le interruzioni, dobbiamo scrivere "`cli();`". Tenere presente che se nel programma non scriviamo nulla, le interruzioni sono disabilitate per impostazione predefinita.

Tornando al diagramma di flusso, il programma principale si chiude poi nel *main loop* dove può eseguire qualunque compito richiesto (nel nostro caso non deve fare più nulla), sino a che non viene *chiamata* la isr.
Sulla destra vi è il semplice diagramma della isr: l'unico compito è il complemento dell'uscita PD7 per il lampeggio del LED, dopodiché la isr termina (in genere ad eccezione del `main()` tutte le funzioni, e anche le isr, hanno una "fine") e la CPU torna ad eseguire le istruzioni del main-loop.
Il programma completo è quindi mostrato nel **Listato 3**.
Notare che, utilizzando questa logica, il programma è più semplice del precedente: non è più nemmeno necessario dover azzerare manualmente il flag OCF1A in quanto, come riporta il datasheet, viene fatto automatica-

mente alla chiamata del relativo vettore d'interruzione (la isr) .

Listato 3

```
/***************************************************************************
 *   Programma per far lampeggiare il LED LD9 della DEMOBOARD          *
 *    usando il TIMER1 in modalità CTC con le interruzioni             *
 ***************************************************************************/

#include <avr/io.h>
#include <avr/interrupt.h> /* necessario includere questo header */

void main(void) {

/* Inizializzazione */

         DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */

/* impostazione T/C 1 in mod. CTC, con prescaler 1024 */

         TCCR1A = 0x00;
         TCCR1B = (1 << CS12) | (1 << CS10) | (1 << WGM12);
         OCR1A = 7812;
         TIMSK = (1 << OCIE1A); /* abilitazione interrupt Su Output Compare
Match */

         sei(); /* abilita la CPU a ricevere le interruzioni */

/* Fine inizializzazione */

         while (1) {   /* ciclo infinito */
                  /* non dobbiamo far nulla! */
         }
}
/* fine programma principale main() */
/* Segue la isr di Output Compare Match del Canale A del Timer 1 */

ISR (TIMER1_COMPA_vect) {
         PORTD = PORTD ^ 0x80; /* complementa PD<7> */
}
```

LA MODULAZIONE PWM

Uno dei sistemi più utilizzati per codificare l'informazione di un segnale digitale è la variazione del tempo in cui resta a livello '1' (detto T_{ON}) rispetto al tempo in cui vale '0' (T_{OFF}), pur mantenendo costante la *somma* dei due (ovvero il periodo T del segnale: $T = T_{ON} + T_{OFF}$); quindi se aumenta T_{ON} deve diminuire T_{OFF} e viceversa. Per quantificare l'informazione codificata con questo sistema non si utilizza la misura del valore di T_{ON} oppure T_{OFF}, ma si ricava il cosiddetto *ciclo utile* (o *duty cycle* in inglese) "D" pari al rapporto percentuale tra il tempo TON e il periodo T: $D = (T_{ON} / T) * 100$; quindi questa grandezza sarà compresa tra 0 (se il segnale è sempre a '0') e 100 (se il segnale è sempre a '1').

In **Fig. 6** potete vedere un esempio di segnale con tre diversi *duty cycle*.

Questa tecnica di codifica dell'informazione (contenuta in D) si dice *modulazione della larghezza dell'impulso* (abbreviato in PWM, dall'inglese *Pulse Width Modulation*). Tra le altre applicazioni, essa è molto sfruttata nel mondo dei microcontrollori per realizzare economici convertitori digitali/analogici: il segnale PWM, generato direttamente da un pin di I/O del micro, può essere filtrato con semplici circuiti per ricavare in uscita una tensione proporzionale a D. Per tale motivo le periferiche timer/counter di molti microcontrollori annoverano la possibilità di generare automaticamente segnali modulati in PWM, ed il nostro ATMEGA16 non fa eccezione: i suoi tre T/C possono essere configurati per tale scopo, ed i segnali PWM risultanti vengono resi disponibili all'utilizzatore sui rispettivi piedini OC_N.

Per vedere un'applicazione pratica, utilizzeremo la modulazione PWM per **variare la luminosità** del 'solito' LED DL9 della demoboard, che finora avevamo fatto solo lampeggiare. Se infatti esso viene alimentato con un segnale PWM, di frequenza abbastanza elevata affinché i nostri occhi non lo vedano lampeggiare, la luminosità che percepiremo sarà proporzionale al *duty cycle* del segnale.

Il LED è collegato sul piedino 21 (PD7) che, guardando sul datasheet al paragrafo "Pin Configurations", è condiviso con il segnale

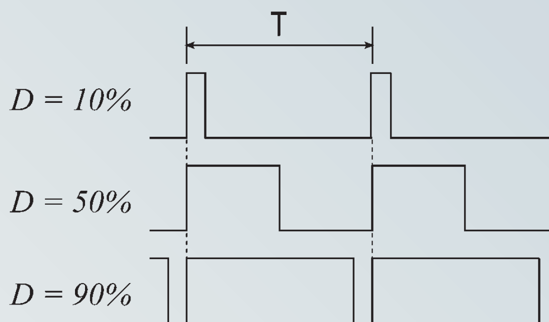


Fig. 6

I diagrammi di flusso

In fase di concepimento di un programma informatico, sia che riguardi grossi calcolatori piuttosto che un microcontrollore, dopo aver capito "bene" quali sono i dati di ingresso a disposizione e che cosa bisogna fare, si delinea il *procedimento* che consenta di arrivare alla soluzione del problema. Tale procedimento viene detto algoritmo e consiste nella descrizione delle operazioni da eseguire, in un preciso ordine, per ottenere il risultato atteso. Per rendere intuitiva questa descrizione, normalmente si rappresenta l'*algoritmo* con uno schema a blocchi grafico in cui le frecce

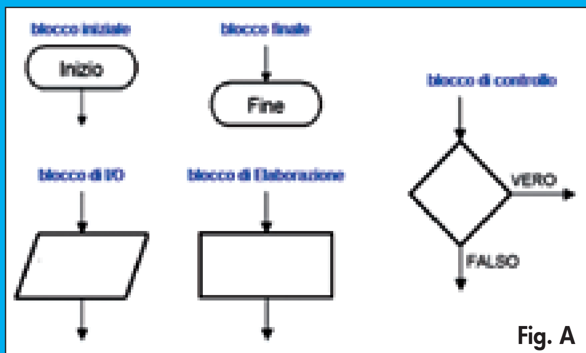


Fig. A

che collegano i vari blocchi danno l'idea della direzione del programma e della sequenza di esecuzione delle varie parti; per tale motivo essi vengono detti *diagrammi di flusso* (*flowcharts*). I blocchi che li compongono possono avere diverse forme, relative al tipo di funzione che devono rappresentare; inoltre normalmente contengono del testo esplicativo dell'operazione che eseguono. Il sito Wikipedia (*http://it.wikipedia.org/wiki/Diagramma_a_blocchi*) descrive bene questi concetti ed i tipi di blocchi che si possono incontrare in questi diagrammi (riportati in **Fig. A**); per la verità ve ne sono anche di altri tipi, ma quelli mostrati sono sufficienti per rappresentare tutte le situazioni (Inizio programma, elaborazione, decisione, input/output, fine programma).

Nel diagramma di flusso di **Fig. 2** ad esempio usiamo già tutti questi tipi di blocchi: notare che il primo è leggermente ovalizzato poiché non rappresenta nessuna operazione ma indica dov'è l'inizio del programma (e non vi è ovviamente la fine, essendo il diagramma della funzione `main()`).

Vi è poi un blocco rettangolare che esegue le operazioni specificate internamente, quindi troviamo un pallino detto "connettore": è semplicemente il punto di unione di più linee (i "flussi" del programma), che affluiscono tutte nel successivo blocco di confronto (rombo). Da lì, se TCNT1 è diverso da K il programma "torna indietro" per fare un nuovo confronto, altrimenti si azzera il contatore e si complementa lo stato del LED (questa è un'operazione di uscita, e dunque va rappresentata con l'apposito simbolo del trapezio, anche se in alcuni casi per semplicità si vede utilizzare sempre il rettangolo).

Siccome la rappresentazione di algoritmi tramite diagrammi di flusso è una pratica molto utilizzata, vi sono numerosi programmi che permettono di realizzarli; alcuni addirittura consentono di estrarre dallo schema il programma vero e proprio, magari in linguaggio C. Chiaramente parliamo di programmi professionali e non gratuiti, se non per periodi limitati di prova. Comunque per realizzare semplici schemi come quelli dei nostri esercizi, vale la pena sapere che anche Microsoft Word mette a disposizione tutti i simboli necessari per il disegno (su Word 2007 vi si accede dal menu Inserisci → Forme → Diagrammi di flusso).

di uscita del timer/counter 2 (OC2). Questo significa che possiamo configurare il T/C2 per generare un segnale PWM su OC2, ed "instradarlo" internamente sul piedino 21 del micro, il quale lo rende disponibile ai circuiti esterni al microcontrollore, nel nostro caso LD9. Si noti anche che questo piedino 21, quando è configurato per riportare il segnale OC2, viene scollegato dal PORTD (pertanto qualsiasi valore che scriveremo successivamente su PD7 tramite programma non verrà più trasferito al piedino).

Questo discorso è valido per tutti i pin di I/O dei micro che sono condivisi con qualche periferica interna: se è configurata opportunamente, essa "prende il controllo" del piedino che non è più gestibile come I/O generico (fino a quando non disabilitiamo questa funzione). Nel datasheet al paragrafo "Alternate Port Functions" del capitolo "I/O Ports" vengono dettagliati questi concetti, oltre a descrivere la "funzione secondaria" che può assumere ciascun piedino. Queste informazioni sono importanti quando, in fase di definizione di un nuovo schema, bisogna decidere i collegamenti e le assegnazioni dei vari pin di I/O del microcontrollore ai circuiti della scheda: nel nostro caso era necessario collegare LD9 proprio al piedino 21, in quanto è l'unico a cui possiamo mandare il segnale di PWM del T/C2. Il T/C2 (ma il discorso è uguale per gli altri 2 timer/counters) prevede due modalità di PWM:

1) "Fast": il contatore TCNT2 incrementa sempre da 0x00 a 0xFF per poi ripartire da 0x00, come nella modalità di conteggio normale; l'uscita OC2 viene alzata quando vi è la transizione 0xFF → 0x00, e riportata a '0' quando TCNT2 raggiunge il valore contenuto in OCR2;
2) "Phase correct": il contatore TCNT2 incrementa da 0x00 a 0xFF e poi decrementa da 0xFF a 0x00; l'uscita è azzerata (0) quando TCNT2 raggiunge OCR2 in "salita" (nella fase di incremento), ed è poi alzata (1) quando TCNT2 raggiunge nuovamente OCR2 in "discesa".

Per i nostri scopi scegliamo di usare la modalità "Fast", e il valore di OCR2 è proporzionale al duty cycle voluto: questo registro è da 8 bit e dunque ha 256 combinazioni possibili

(da 0 a 255): pertanto la risoluzione, intesa come la minima variazione di duty cycle che possiamo ottenere incrementando o decrementando di un'unità OCR2, vale 100/256 = 0,39%: questo valore viene detto granularità del PWM per la nostra periferica. Se in particolari applicazioni ci servisse una maggiore "finezza" di regolazione dovremmo usare il T/C 1 poiché ha registri da 16 bit (65536 combinazioni): la *granularità* risultante sarebbe pertanto del 100/65536 = 0,0015%.

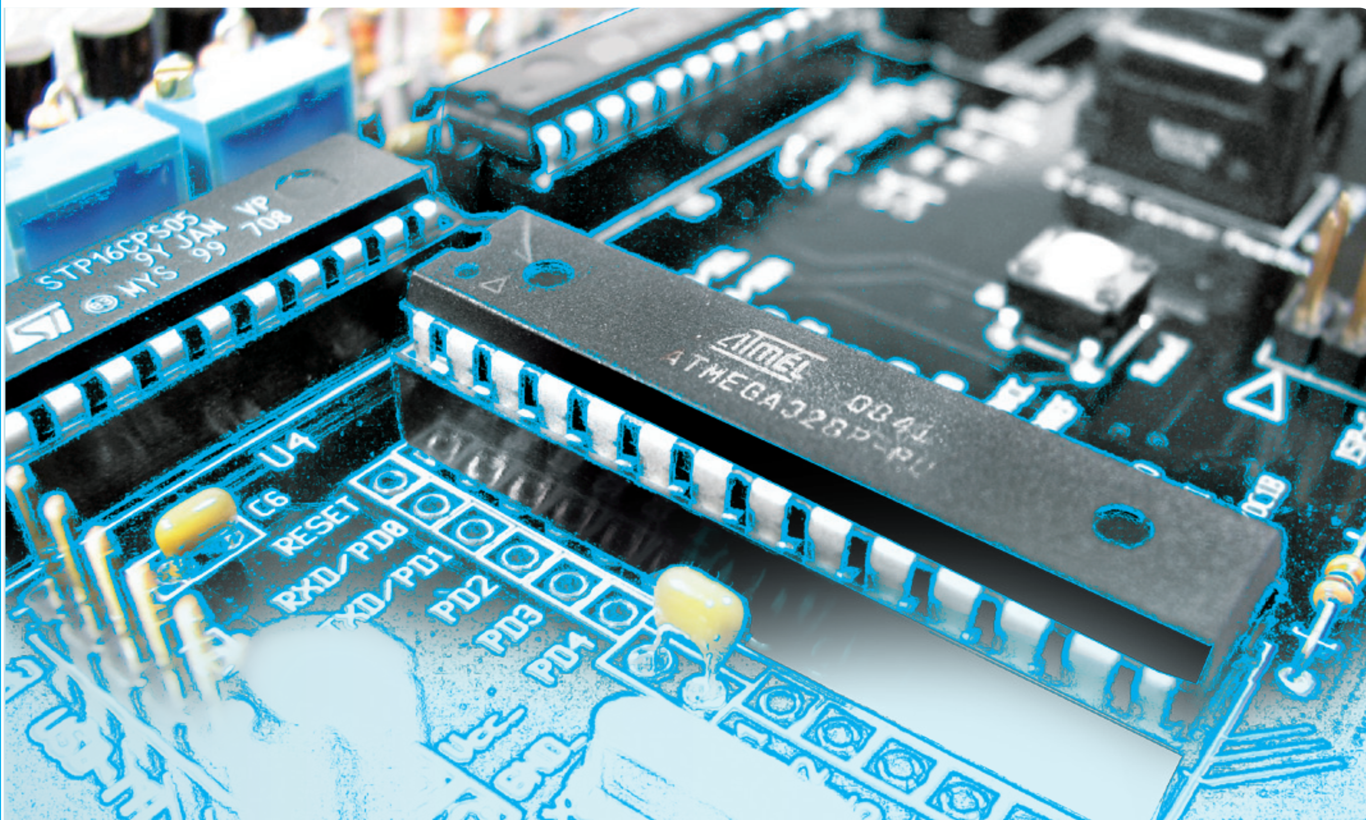
Listato 4

```
/***********************************************************************
 *   Programma per variare la luminosità del LED LD9 della DEMOBOARD  *
 *    usando il TIMER2 in modalità "Fast PWM"                         *
 ***********************************************************************/

#define DUTY_CYCLE 200 /* impostare un valore tra 0 e 255 */

#include <avr/io.h>

void main(void) {

/* Inizializzazione */

         DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB come uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */

/* impostazione T/C 2 come fast PWM, no prescaler, uscita su pin OC2 */
         TCCR2 = (1 << CS20) | (1 << WGM21) | (1 << WGM20) | (1 << COM21);
         OCR2 = DUTY_CYCLE; /* carica in OCR2 il valore definito sopra */

/* Fine inizializzazione */

         while (1) {   /* ciclo infinito */

                  /* non dobbiamo far nulla! */

         }
}
```

L'impostazione del T/C 2 per questa funzionalità è abbastanza semplice: vi è un unico registro (TCCR2) contenente tutti i bit di configurazione necessari:

- Bit del prescaler CS22, CS21, CS20: li impostiamo per il minimo fattore di divisione (1, cioè nessuna divisione del clock) per ottenere una frequenza tale che il nostro occhio non veda "vibrare" la luce del LED (nel datasheet vi è la formula per ricavare la frequenza esatta del segnale PWM in base al clock in ingresso al T/C e alla divisione del prescaler);
- Bit di modalità funzionamento WGM21,WGM20: li mettiamo entrambi a '1' per configurare il T/C nella modalità "Fast PWM";
- Bit COM21, COM20: essi vanno impostati per consentire al timer di "prendere il controllo" del piedino 21 (PD7), in caso contrario in uscita non arriverà il segnale PWM; le loro combinazioni sono esplicate nella tabellina "Compare Output Mode, Fast PWM Mode" nel paragrafo "Register Description" del T/C 2. Nel nostro caso impostiamo COM2<1:0> = 10; si noti che se COM2<1:0> = 11 il segnale PWM sul piedino viene invertito, e la luminosità diminuirebbe al crescere del valore di OCR2.

Nel **Listato 4** si può veder il programmino per accendere il LED con una luminosità proporzionale al valore definito per comodità nella costante DUTY_CYCLE: variandolo (e quindi ricompilando e riprogrammando la scheda) potrete vedere come cambia la luminosità. Notare solo che anche mettendo zero il LED non viene completamente spento, poiché la periferica, quando TCNT2 torna a zero, alza sempre il piedino prima di fare la comparazione con OCR2 e abbassarlo di nuovo: questo produce degli "spilli" di tensione che bastano ad accendere, seppur molto brevemente, il LED.
Per rendere più interessante l'esercizio potreste fare incrementare e decrementare in modo automatico la luminosità, caricando OCR2 nel *main loop* non con un valore costante ma con una variabile globale (da 8 bit, dichiarata ad esempio come "`uint8_t MiaVar;`" prima della funzione `main()`), fatta incrementare e/o decrementare ad intervalli di tempo ragionevoli per osservare la variazione; tale variabile potrebbe ad esempio essere incrementata nella isr del T/C1 (nel **Listato 3**, al posto del complemento di PD7). Volete provarci?
Vi diamo un mese di tempo, comunque state tranquilli: nella prossima puntata pubblicheremo la soluzione, oltre a proseguire nell'esplorazione delle funzionalità della demoboard.



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

Descriviamo il funzionamento di alcune periferiche di input/output presenti nella scheda demoboard: buzzer, LED, tasti e display alfanumerico. Settima puntata.

Nella scorsa puntata ci siamo lasciati con la proposta di realizzare un programma per variare in modo automatico la luminosità del LED LD9 della demoboard, dal minimo al massimo e viceversa, creando un effetto "pulsante". Nel **Listato 1** vi è la soluzione al problema, con la quale potrete confrontare quella scritta da voi, che potrebbe peraltro funzionare bene anche se diversa: ricordate che non c'è quasi mai un modo univoco di concepire e implementare un algoritmo!
In questo programma si utilizzano due *variabili globali*, denominate così perché accessibili e utilizzabili in tutte le funzioni del programma (che nel nostro caso sono la `main()` e la isr (`TIMER1_COMPA_vect`)). Le variabili globali si dichiarano al di fuori delle routine; le nostre le chiameremo '`direzione`' e '`luce`', entrambe da 8 bit. La prima assume solo i valori zero e uno (bastava un bit, ma la variabile più piccola dichiarabile è da 8 bit!) per indicare al programma qual è la *fase* in corso: se zero la luminosità sta crescendo, se uno sta diminuendo; '`luce`' contiene invece il valore di luminosità (da 0 a 255).
Nel *main-loop* non vi sono istruzioni: tutto si svolge nella isr (Interrupt Service Routine) del timer/counter 1 (`TIMER1_COMPA_vect`), che viene attivata ogni 7ms circa poiché abbiamo caricato 100 in OCR1A nella fase di inizializzazione.
Ad ogni chiamata di questa isr si verifica in che fase ci troviamo: se '`direzione`' è zero la lumi-

Listato 1

```
/**********************************************************************
 *   Programma per rendere “pulsante” il LED LD9 della DEMOBOARD      *
 *    usando il T/C 1 per la velocità del lampeggio                   *
 *    e T/C 2 per generare il PWM                                     *
 **********************************************************************/

#include <avr/io.h>
#include <avr/interrupt.h>

/* Dichiarazione variabili globali */
uint8_t luce = 0;          /* livello di luminosità */
uint8_t direzione = 0;     /* 0 = lum. crescente; 1 = lum. calante */

void main(void) {

/* Inizializzazione */

         DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB sono uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* forziamo a ‘0’ logico tutti i pin dei port impostati come uscite */
         PORTA = PORTB = PORTC = PORTD = 0;

/* impostazione T/C 1 in mod. CTC, con prescaler 1024 */

         TCCR1A = 0x00;
         TCCR1B = (1 << CS12) | (1 << CS10) | (1 << WGM12);
         OCR1A = 100;               /* con 100 il timer scatta ogni 7 ms circa */
         TIMSK = (1 << OCIE1A); /* abilitaz. int. Su Output Compare Match */

         sei(); /* abilita la CPU a ricevere le interruzioni */

/* impostazione T/C 2 come fast PWM, no prescaler */
         TCCR2 = (1 << CS20) | (1 << WGM21) | (1 << WGM20) | (1 << COM21);

/* Fine inizializzazione */

         while (1) {   /* main loop: ciclo infinito */

                  /* non dobbiamo far nulla! */

         }
}

/**** ISR T/C 1: con OCR1A = 100 è chiamata ogni 7ms ****/
ISR (TIMER1_COMPA_vect) {
         if (direzione == 0) { /* fase di luminosità crescente */
                  if (luce < 255)  /* limita l’intervallo di ‘luce’ a 255 */
                           luce ++;     /* incrementa ‘luce’ */
                  else
                           direzione = 1;/* se ‘luce’ = 255 inizia la fase calante*/
         }
         else {   /* fase di luminosità calante */
                  if (luce > 0)  /* controlla se ‘luce’ > 0 */
                           luce --;    /* se sì decrementala */
                  else
                           direzione = 0; /* altrimenti passa a fase crescente */
         }
         OCR2 = luce; /* infine aggiorna OCR2 per variare il PWM sul LED */
}
```

nosità deve crescere e quindi incrementiamo di uno la variabile ‘`luce`’ con l’istruzione C: “`luce ++;`” . Se però ‘`luce`’ vale 255 la luminosità è arrivata al massimo, quindi iniziamo la fase calante mettendo ‘1’ nella variabile ‘`direzione`’. Alla prossima chiamata della isr si andrà ad eseguire il codice relativo alla fase di luminosità calante sin quando ‘`luce`’, decrementata ogni volta, non arriva a zero (luminosità minima): in tal caso ‘`direzione`’ è riportata a zero ed il ciclo ricomincia. Prima di “uscire” dalla isr, viene inoltre sempre aggiornato il valore del PWM al LED (impostato da OCR2) con il valore corrente di ‘`luce`’.

BUZZER

Sulla demoboard vi è un piccolo cicalino piezoelettrico, che se alimentato con una tensione variabile produce una nota acustica. Questo dispositivo è collegato sul piedino PD5, che

se guardate nel capitolo del datasheet "Pin Configurations" corrisponde anche all'uscita *Output-Compare* del canale A del Timer/counter 1 (OC1A): pertanto possiamo utilizzare questa periferica per accendere il buzzer ad una frequenza programmata. Nel **listato 1**, apportate questa semplice modifica nella sezione di inizializzazione: al posto di

```
TCCR1A = 0x00;
```

scrivete

```
TCCR1A = (1 << COM1A0);
```

che corrisponde, come ormai avrete capito, a settare a '1' il bit COM1A0 del registro TCCR1A, mentre negli altri bit del registro viene scritto '0'.

In accordo col datasheet, questo abilita il piedino PD5 ad essere complementato tutte le volte che nel timer/counter 1 vi è un evento di *output compare* del canale A (lo stesso che attiva anche la isr). Riprogrammando il micro col programma così modificato sentirete uscire dal buzzer, più che un suono, una specie di ronzio; d'altra parte il periodo del segnale che "spariamo" dal piedino è pari al doppio del tempo tra due successivi eventi di output-compare, quindi la frequenza è circa 1/(14 ms) = 70 Hz.

Per sentire una nota acustica 'degna' (dipende anche dalle caratteristiche del buzzer) dobbiamo andare almeno sui 3-400Hz: per fare questo facciamo contare il T/C 1 solo sino a 20 invece che a 100, quintuplicando la velocità degli eventi di output-compare:

```
OCR1A = 20;
```

La nota acustica prodotta sarà senz'altro più apprezzabile e vedrete che il led, ovviamente, pulserà più velocemente (esattamente 5 volte più veloce). Naturalmente anche con il buzzer potrete sbizzarrirvi nel programma per variarne la frequenza, e magari rendere la variazione automatica realizzando così una sirena!

USCITE DI "POTENZA"

La demoboard mette a disposizione, tramite U3, otto uscite cosiddette "open collector" in quanto si comportano come degli interruttori verso massa, che vengono chiusi quando il rispettivo ingresso IN di U3 è messo a '1', oppure rimangono aperti quando l'ingresso IN di U3 viene posto a '0'. Con questo tipo di uscite si possono, ad esempio, pilotare direttamente piccoli carichi, come lampadine o relè, alimentabili con la tensione di +12V prelevabile dalla demoboard stessa, e resa disponibile (insieme al +5V) su alcune piazzole vicino all'area millefori. Si tenga però presente che la corrente massima disponibile dipende dall'alimentatore della demoboard e comunque non deve superare i 500mA (poiché dev'essere tollerata dal diodo D1). Può pure essere utilizzata la linea a +5V, ma in questo caso la corrente massima non dovrebbe eccedere i 200mA. In **Fig. 1** vi sono alcuni esempi di collegamento dei carichi sulle uscite O<7:0> di U3.

Per monitorare lo stato delle uscite sono stati posti otto LED (LD<8:1>) sugli ingressi di U3, che si accendono quando il rispettivo canale è attivo (a '1') e di conseguenza l'uscita di U3 è a massa.

Gli otto segnali che pilotano gli ingressi di U3 ed i LED non sono collegati direttamente al microcontrollore ma sulle uscite di U2, un *D-latch* ottuplo siglato 74HC573. Questo dispositivo è una sorta di porta di comunicazione a 8 bit tra gli ingressi D<7:0> e le uscite Q<7:0>. Quando il piedino LE (*Latch Enable*) è a '1' logico gli ingressi sono collegati alle rispettive uscite, che ne rispecchiano quindi i livelli logici: se D<7:0> = 0x3F, anche Q<7:0> = 0x3F. Quando

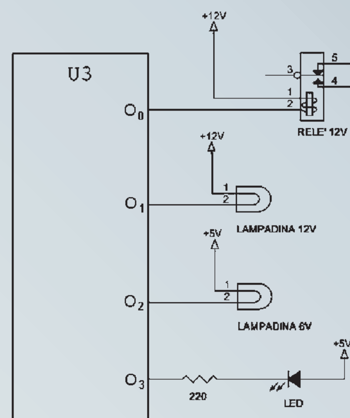


Fig. 1

LE viene posto a '0' le uscite memorizzano il valore che avevano nell'istante in cui 'LE' passa da '1' a '0' e lo mantengono, indipendentemente dalle variazioni successive degli ingressi; quindi se nella transizione 1 → 0 di LE in uscita vi era 0x3F, questo valore permane per tutto il tempo in cui LE è a zero anche se gli ingressi assumono un altro valore.

Anche dopo aver capito come funziona U2 potrebbe però nascere una domanda: perché

usare questo integrato invece di pilotare direttamente U3 con un port del microcontrollore? La risposta è che le linee del micro che utilizziamo per i canali di potenza (quelli relativi a U3) sono anche utilizzate, come si vede nello schema, per pilotare il display LCD di cui parleremo più avanti nella puntata. Il problema è che quando mandiamo i dati relativi al display non vogliamo che essi vadano ad influire lo stato delle linee di U3, pertanto U2 serve a bloccare questi dati (lasciando LE a '0' quando "colloquiamo" col display).
Questo stratagemma nasce dal fatto che il nostro ATMEGA16, seppure ben "dotato", non ha un numero infinito di linee di I/O e questo sistema ci consente di risparmiarne qualcuna: se usassimo tutte linee separate, ce ne servirebbero 8 (per U3) + 6 (per il display) = 14 pin di I/O. Adottando U2 ne bastano 10 (8 linee di dati condivise più i segnali di LE per il latch ed E del display).
Passiamo dunque a scrivere un programmino per gestire le linee di potenza: non ci occorrerà collegare dei carichi sulle uscite di U3 in quanto, come già detto, il loro stato può essere monitorato dai LED LD<8:1>.
Queste otto linee fanno capo, attraverso U2, all'intero PORTB del micro, pertanto per scrivervi un valore l'algoritmo sarà il seguente:

- caricare il valore desiderato sul PORTB;
- alzare (0 → 1) e poi abbassare di nuovo (1 → 0) il segnale LE di U2, per memorizzare il valore.

Con questo sistema si aggiorna contemporaneamente lo stato di tutte le linee.
Il segnale LE fa capo al pin I/O PD6, dunque la transizione voluta si ottiene con le istruzioni:

```
PORTD |= 0x40; /* alza PD6 senza alterare
gli altri bit */
PORTD &= ~0x40; /* abbassa nuovamente PD6
senza alterare gli altri bit */
```

Poiché 0x40 vale 0100 0000 in binario, l'operazione di OR bit a bit del C (operatore '|') va ad alzare il solo sesto bit del PORTD, senza alterare gli altri. Parimenti, siccome l'operatore '~' del C fa la negazione bit a bit, si ottiene che ~0x40 = 1011 1111 per cui l'operazione di AND bit a bit del C (operatore '&') va ad azzerare sempre e solo il sesto bit di PORTD.
Siccome l'algoritmo sopra va eseguito ogni volta che vogliamo aggiornare il valore delle linee di potenza, conviene "rinchiuderlo" in una funzione che poi sarà chiamata dal programma tutte le volte necessarie.
Il programmino di esempio proposto prevede di attivare una uscita per volta, dalla prima alla ottava, per poi invertire il percorso e tornare alla prima, e così via: negli otto LED si visualizzerà un effetto "ping-pong". Per cadenzare lo scorrimento utilizziamo il T/C1, demandando alla sua isr il compito di aggiornare la visualizzazione ed il verso avanti-indietro, esattamente come avevamo fatto per creare il lampeggio pulsante di LD9 (**Listato 1**). Similmente a quel programma, per tenere traccia del verso di percorrenza, utilizziamo la variabile '`direzione`' : 0 = avanti (da LD1 a LD8), 1 = indietro (da LD8 a LD1). Quando gli estremi LD1 o LD8 sono raggiunti, la variabile '`direzione`' viene cambiata ed il ciclo si ripete.
Nel **listato 2** è riportato il sorgente completo del programma: essendo strutturalmente molto simile a quello analizzato nel **listato 1**, ne discutiamo solo le poche differenze. Come già suggerito, abbiamo scritto una piccola funzione (`ImpostaLineePotenza`()) per effettuare l'aggiornamento delle linee di uscita del *D-latch* U2: a tale funzione viene passato come argomento un numero a 8 bit con il valore che vogliamo assumano le linee.
Nella fase di inizializzazione del micro, durante l'azzeramento dei port, è stato aggiunto il codice

```
PORTD = 0x40; PORTD = 0;
```

in modo da generare un impulso sul segnale 'Latch Enable' di U2: poiché i bit di dati sono tutti '0' (l'istruzione precedente azzerava anche PORTB), l'effetto è che all'accensione del sistema tutte le linee di potenza vengono inizializzate a zero (inattive). Questo è importante poiché avrete notato che in mancanza di questa inizializzazione, quando si alimenta la scheda i LED LD<9:1> si accendono in maniera "casuale": il D-latch 74HC573 non possiede un sistema di reset che inizializzi i suoi circuiti interni ad uno stato noto, pertanto all'accensione non è specificato lo stato delle sue uscite.
La variabile `PosizioneLed` viene inizializzata con il bit meno significativo a uno: questo '1' sarà poi fatto scorrere in avanti e indietro rispettivamente con le istruzioni del C:

```
PosizioneLed = PosizioneLed << 1; /* spo-
```

Listato 2

```
/*************************************************************************
 *  Programma per la gestione delle linee di potenza della demoboard  *
 *    con effetto "palleggio"                                         *
 ************************************************************************/

#include <avr/io.h>
#include <avr/interrupt.h>

/* Prototipi delle funzioni */
void ImpostaLineePotenza(uint8_t Dato);

/* variabili globali */
uint8_t PosizioneLed = 0x01; /* il bit a '1' determina il LED acceso */
uint8_t direzione = 0;      /* 0 => da LD1 a LD9; 1 => da LD9 a LD1 */

void main(void) {

/* Inizializzazione */

         DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB sono impostati come uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* forziamo a '0' logico tutti i pin dei port impostati come uscite */
         PORTA = PORTB = PORTC = 0;
         PORTD = 0x40; /* impulso su 'LE' per spegnere le linee di potenza */
         PORTD = 0;

/* impostazione T/C 1 in mod. CTC, con prescaler 1024 */
         TCCR1A = 0;
         TCCR1B = (1 << CS12) | (1 << CS10) | (1 << WGM12);
         OCR1A = 1500;     /* 1500: il timer scatta ogni circa 100 ms */
         TIMSK = (1 << OCIE1A);     /* abilitaz. int. Su Out Compare Match*/
         sei(); /* abilita la CPU a ricevere le interruzioni */
/* Fine inizializzazione */

         while (1) {   /* main loop: ciclo infinito */
                  /* non dobbiamo far nulla! */
         }
}

ISR (TIMER1_COMPA_vect) {
         if (direzione == 0) { /* scorrimento LD1 -> LD8 */
                  if (PosizioneLed != 0x80)  /* se non è ancora acceso LD8 */
                           PosizioneLed = PosizioneLed << 1;   /* accendi il LED successivo */
                  else
                           direzione = 1;/* se era acceso LD8, cambia direzione */
         }
         else {   /* scorrimento LD8 -> LD1 */
                  if (PosizioneLed != 0x01)  /* se non è ancora acceso LD1 */
                           PosizioneLed = PosizioneLed >> 1;   /* accendi il LED precedente */
                  else
                           direzione = 0; /* se era acceso LD1, cambia direzione */
         }
         ImpostaLineePotenza(PosizioneLed); /* infine aggiorna i LED */
}

/*
 * Funzione ImpostaLineePotenza()
 *   memorizza sulle uscite di U2 il dato contenuto
 *   nella variabile passata come argomento
 */
void ImpostaLineePotenza(uint8_t Dato) {
         PORTB = Dato;
         PORTD |= 0x40; /* alza PD6 senza alterare gli altri bit */
         PORTD &= ~0x40; /* abbassa PD6 senza alterare gli altri bit */
}
```

```
sta di una posizione a sx */
PosizioneLed = PosizioneLed >> 1; /* spo-
sta di una posizione a dx */
```

nella isr del T/C1, nella quale ci si occupa anche di gestire la direzione del "palleggio" ed infine di aggiornare lo stato delle uscite con la chiamata a `ImpostaLineePotenza()`, passandogli come argomento il byte con la posizione aggiornata (il bit che è a '1') del LED da accendere.

Pertanto, anche in questo caso il *main-loop* è vuoto: il processore vi "gira dentro" in attesa

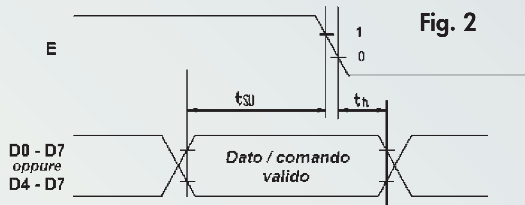


Fig. 2

Characteristic	Symbol	Min.	Unit
Data set-up time	t_{SU}	60	ns
Data hold time	t_h	10	ns

Fig. 3

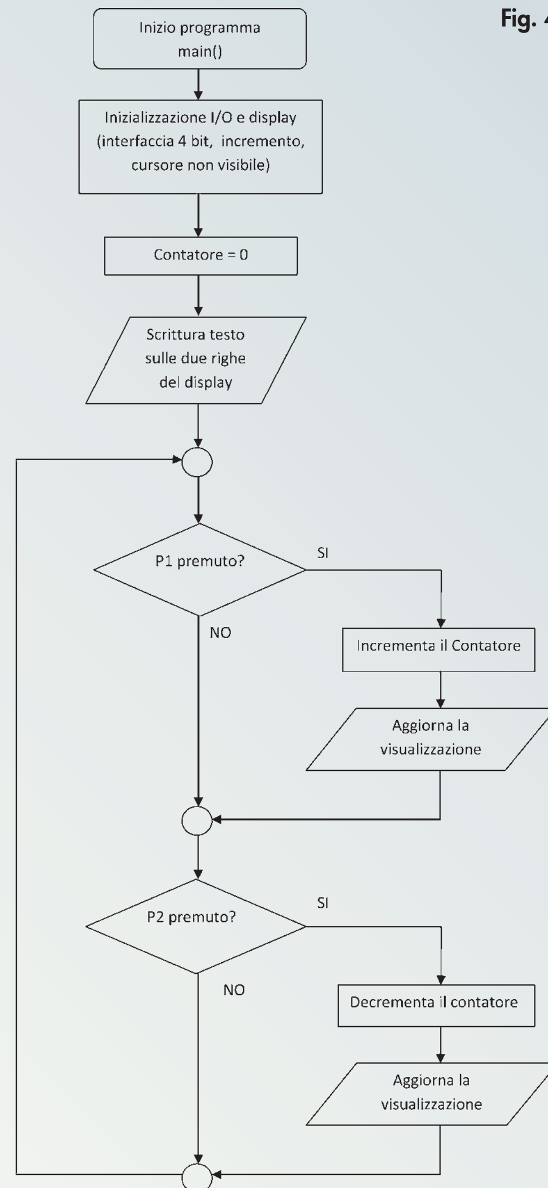


Fig. 4

della isr, la esegue, e torna al *main-loop* in attesa della prossima.
Questo è solo un esempio di come gestire le linee di potenza: di tutto il programma voi potreste mutuare la sola funzione `ImpostaLineePotenza()` per scrivere del codice che soddisfi le vostre esigenze, ad esempio collegando sulle uscite di U3 dei relè potreste pilotare delle lampade o motori a 230V per realizzare lampeggianti, automatismi, e così via.

LEGGIAMO GLI INGRESSI

Passiamo ora a vedere come fare per leggere con il micro ATMEGA16 gli stati logici dei pin di I/O. Come già sappiamo, bisogna innanzitutto configurarli come ingressi, mettendo a zero il bit relativo del registro di direzione: se vogliamo impostare come ingresso il piedino PC0 dobbiamo mettere zero il bit 0 del registro DDRC (Data Direction Register C); ad esempio con l'istruzione

```
DDRC = 0b11111110;
```

poniamo come ingresso il pin PC0, e come uscite tutti gli altri del port C (i piedini PC<7:1>).
Fatto questo, nel programma possiamo leggere (o "campionare", come talvolta si dice) lo stato del piedino PC0 andando ad interrogare il registro "PINC" (Port C Input Pin register), il cui bit 0 rifletterà il valore del piedino, ad esempio:

```
if (PINC & 0x01)  /* se il pin PC0 è '1'
esegui il corpo dell'if */
```

Oppure, utilizzando l'operatore "NOT" del C (il carattere '!'), invertiamo la logica e verifichiamo lo stato '0':

```
if (!(PINC & 0x01))  /* se il pin PC0 è '0'
esegui il corpo dell'if */
```

Facciamo un esempio concreto: nella demoboard vi sono due pulsanti, P1 e P2, collegati tramite resistore di pull-up ai piedini PD2 e PD3. Tali piedini sono normalmente a '1' (per via dei resistori di pull-up R17 e R18), e vengono posti a '0' premendo il tasto relativo. Con queste informazioni, nel listato 3 è mostrato un programmino che accende LD9 quando premiamo P1, e lo spegne premendo P2.
Un'ultima annotazione: l'header "io.h" del WinAVR definisce il numero dei bit di tutti i registri; ad esempio il bit 2 del registro PIND viene definito come:

```
#define PIND2 2
```

E pertanto nel programma per testare il pin PD2, al posto di

```
if (!(PIND & 0b00000100))
```

Listato 3

```
/*************************************************************************
 * Programma per accendere LD9 con P1 e spegnerlo con P2                 *
 *************************************************************************/

#include <avr/io.h>

void main(void) {

/* Inizializzazione */

         DDRA = DDRB = 0xFF;  /* Tutti i pin di PORTA e PORTB sono impostati come uscite */
         DDRC = 0b10111100; /* PC0, PC1 e PC6 sono ingressi */
         DDRD = 0b11110011; /* PD2 e PD3 sono ingressi */
/* forziamo a ‘0’ logico tutti i pin dei port impostati come uscite */
         PORTA = PORTB = PORTC = 0;
         PORTD = 0x40; /* impulso su ‘LE’ per spegnere le linee di potenza*/
         PORTD = 0;

/* Fine inizializzazione */

         while (1) {   /* main loop: ciclo infinito */

                  if (!(PIND & 0b00000100)) /* PD<2> = 0 -> P1 è premuto */
                           PORTD |= 0x80;             /* accendi LD9 */

                  if (!(PIND & 0b00001000)) /* PD<3> = 0 -> P2 è premuto */
                           PORTD &= ~0x80;            /* spegni LD9 */
         }
}
```

potreste anche scrivere, se vi sembra più chiaro:
`if (!(PIND & (1 << PIND2)))`
e questo ovviamente vale per tutti gli altri piedini dei port.

USIAMO IL DISPLAY LCD

Il display della demoboard è un modulo intelligente in grado di visualizzare due righe di 16 caratteri ciascuna. Cosa vuol dire “intelligente”? Che per pilotarlo bisogna inviargli, in sequenza su apposite linee, il codice dei caratteri desiderati (con codifica ASCII) e la locazione in cui visualizzarli, ed il suo controller interno si preoccuperà di gestire le operazioni necessarie al comando dei *pixel* dello schermo per comporre la visualizzazione richiesta. Il controller in questione è lo “storico” chip HD44780 di Hitachi o equivalente, su cui si basano tutti i display LCD alfanumerici con 8, 16 o 20 caratteri per riga e 1, 2 o 4 righe. Questo tipo di display è da tempo molto diffuso, pertanto sulla Rete vi si trova qualunque tipo di documentazione ed esempi di programmazione in Assembly, C e quant’altro; in ogni caso discuteremo brevemente cosa bisogna fare per renderlo operativo e svilupperemo un semplice programma per visualizzare caratteri e numeri.
Per poter funzionare, il display deve ricevere i dati da visualizzare, ed i comandi per essere impostato (ad esempio per specificare la locazione in cui scrivere); la comunicazione avviene tramite otto linee di dato D<7:0>, il segnale RS (Register Select) per specificare al modulo se sulle linee D<7:0> vi è un comando (RS = 0) oppure un dato da visualizzare (RS = 1), ed un segnale di abilitazione E la cui transizione 1 → 0, analogamente a quanto visto per il segnale LE del *D-latch* 74HC573, dice al modulo di “incassare” il dato presente in quell’istante su D<7:0>. La **Fig. 2** mostra questa fase, evidenziando anche il tempo t_{SU} in cui il dato dev’essere stabile prima della transizione (detto tempo di setup) ed il tempo t_H in cui deve mantenersi stabile dopo la transizione (detto tempo di hold).
Pertanto per trasmettere un’informazione al display dovremo usare il seguente algoritmo:
1) impostare le linee D<7:0> del display con l’informazione da trasmettere;
2) mettere RS a ‘0’ oppure a ‘1’ a seconda che l’informazione sia un comando oppure un dato;
3) alzare E;
4) attendere il tempo di setup;
5) abbassare E;
6) attendere il tempo di hold.

È pure possibile programmare il display affinché accetti informazioni solo su quattro delle otto linee di dato, (D<7:4>), mentre le altre (D<3:0>) possono essere collegate a massa (forzate a ‘0’). Per fare questo, il microcontrollore deve dividere l’informazione (che è sempre a 8 bit) in due dati di 4 bit (*nibble*) e trasmettere

Il codice ASCII

Questo codice (che è l'acronimo di *American Standard Code for Information Interchange*, ovvero *Codice Standard Americano per lo Scambio di Informazioni*), è nato nei primi anni '60 per uniformare la codifica binaria dei caratteri dell'alfabeto e dei numeri contenuti nelle tastiere dei calcolatori, ovvero per stabilire un'associazione univoca di ogni carattere con un preciso numero binario. Esso è divenuto *de facto* uno standard mondiale di codifica, ed ancora oggi pigiando un tasto sulla tastiera noi trasferiamo al PC la codifica ASCII di tale tasto. Inizialmente la codifica avveniva su 7 bit (128 combinazioni), più che sufficienti per rappresentare tutti i caratteri di una normale tastiera (lettere maiuscole

dec	hex	ascii
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9

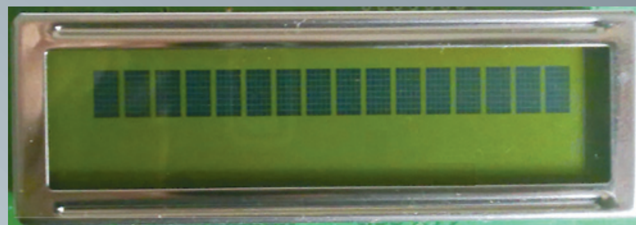
dec	hex	ascii
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
	⋮	
121	79	y
122	7A	z

Fig. A

e minuscole e i numeri), compresi i cosiddetti "codici non stampabili" come DEL, ALT, Control, eccetera. In **Fig. A** mostriamo come esempio le codifiche ASCII dei numeri da zero a nove e delle lettere dell'alfabeto minuscole.
In seguito si estese la codifica su 8 bit per raddoppiare le combinazioni (ora 256) e poter inserire nuovi caratteri "speciali", come i simboli grafici per rappresentare caratteri greci, cirillici, e così via. Va detto però che l'estensione a 8 bit, ovvero la codifica dei caratteri da 128 a 255, *non* è standard (un video terminale potrebbe visualizzare un carattere diverso da quello pigiato con la tastiera ad esso collegata, ma di marca o Paese diversi). I moduli LCD alfanumerici, che ovviamente possono visualizzare solo i caratteri "stampabili", sono aderenti alla codifica ASCII per i codici da 0x20 (il carattere spazio ' ') a 0x7D (il carattere '}'), mentre possono differire per i caratteri che visualizzano con i codici superiori (fino a 0xFF); per sapere quali sono, basta consultare la tabellina normalmente chiamata "Character pattern" sul relativo datasheet.
Da qualche anno è anche comparsa una nuova codifica: l'*Unicode*, pensato con l'ambizione di poter uniformare sotto un unico codice i caratteri rappresentabili di *tutte le lingue del mondo* (ma è ancora in evoluzione), ed associa ogni carattere con un numero binario di ben 21 bit (anche se era nato a 16 bit, dimostratisi subito insufficienti con "soli" 65536 caratteri rappresentabili). Le piattaforme informatiche (linguaggi di programmazione od anche sistemi operativi – Windows CE 6 utilizza solo la codifica Unicode) si stanno adeguando a questo nuovo formato, ma non temete: nel mondo dei microcontrollori e dei moduli LCD il codice ASCII dominerà incontrastato ancora a lungo!
Per maggiori informazioni sul codice ASCII nel web, basta digitarlo in un motore di ricerca e compare "il mondo", a partire dalla sempre apprezzata Wikipedia; per conoscere meglio l'Unicode, potete partire dal sito *www.unicode.org/standard/translations/italian.html*.

sulle linee D<7:4> del modulo prima il nibble superiore (che corrisponde ai bit <7:4> del dato di partenza) e poi quello inferiore (i bit <3:0>); pertanto l'algoritmo visto sopra raddoppia.
Anche nella nostra demoboard usiamo questa modalità poiché, a fronte di un leggero incremento del codice, ci fa risparmiare quattro linee del microcontrollore: i pin di I/O utilizzati sono 4 (dati) + 2 (RS, E) = 6, invece dei 10 necessari per la comunicazione a 8 bit.
Per vedere in concreto come va gestito, scarichiamo dal sito della rivista il codice sorgente di un programma applicativo chiamato "ProvaDisplay.c", che andremo a copiare-incollare dentro l'editor sorgente di un progetto creato in AVR Studio con la consueta procedura analizzata la scorsa puntata; il nome del progetto (e del file sorgente su cui copiamo il contenuto di "ProvaDisplay.c") può essere qualsiasi.
Dopo le operazioni di compilazione e di programmazione, dovremo vedere visualizzati sulle due righe del display i messaggi visibili in **Fig. 3** (se non appare nulla, controllate il trimmer R9 come spiegato nel paragrafo "Il contrasto del display" più avanti in questo articolo).
Il numero della seconda riga, contenuto nella variabile "Contatore", può essere incrementato o decrementato premendo rispettivamente i pulsanti P1 e P2; poiché "Contatore" è definita come `uint8_t` (quindi un byte), si possono rappresentare i numeri da 0 a 255.
Nella **Fig. 4** è mostrato il diagramma di flusso del programma: vi è la solita fase di inizializzazione delle periferiche utilizzate e anche del display: per renderlo operativo dopo che è stato alimentato, infatti, bisogna trasmettergli precise sequenze di numeri per istruirlo sulla modalità di funzionamento, ad esempio per informarlo che l'interfaccia verso il micro è a 4 bit anziché 8 (che sarebbe il valore predefinito).
Nel main-loop viene continuamente controllato lo stato dei pulsanti P1 e P2 per sapere se incrementare o decrementare la variabile "Contatore", ed aggiornare conseguentemente il suo valore sul display.
Poiché lo scopo del programma è abbastanza semplice, ci focalizzeremo sulle funzioni relative alla gestione del display; una volta capito il loro funzionamento, potrete copiare il loro codice per riutilizzarlo in tutte le applicazioni volute; non dimenticatevi anche di inserire pri-

Fig. 5



ma del main() la definizione dei loro prototipi, che riportiamo di seguito:

```
void LCD_Init(void);/* Inizializzazione display */
void LCD_Command (uint8_t dato); /* Trasmissione comando */
void LCD_DisplayChar (uint8_t dato); /* Visualizza carattere ASCII */
void LCD_DisplayString(char *str_ptr, uint8_t Nchar); /* vis. Stringa */
void LCD_DisplayChar3Num(uint8_t numero);
/* Visualizza un byte su tre cifre */
void ImpulsoEnable(void);
void RitardoCorto(void);
void RitardoLungo(void);
```

L'INIZIALIZZAZIONE DEL DISPLAY

Il codice necessario all'inizializzazione è contenuto nella funzione `LCD_Init()`, che va necessariamente chiamata prima di poter utilizzare il display. Peraltro è facile sapere se il display non è stato inizializzato (o comunque non correttamente), poiché esso mostrerà (se il contrasto è stato regolato, come spieghiamo più avanti) la prima riga con tutti i pixel accesi e la seconda con tutti i pixel spenti (**Fig. 5**).
In sintesi, nella funzione `LCD_Init()` vi sono le istruzioni per :

- impostare l'interfaccia dati a 4 bit;
- *pulire* il display (spegnere tutti i pixel e portare il cursore sul primo carattere della prima riga (*home*);
- impostare la modalità di visualizzazione su due righe, con cursore invisibile e incremento posizione dopo scrittura di ogni carattere.

Il *cursore*, analogamente a quello lampeggiante che viene visualizzato quando editiamo un documento di testo sul PC, indica la posizione in cui verrà scritto il prossimo carattere. Esso può essere impostato per venire incrementato dopo la scrittura (come quello del PC), decrementato (la scrittura avviene al contrario da destra verso sinistra), oppure rimanere fermo (il nuovo carattere compare sempre nella stessa posizione, ed è l'intera riga a scorrere). Inoltre il cursore può essere visibile come linea (*underscore*), lampeggiare, oppure essere lasciato spento (o invisibile, come nel nostro programma). Per conoscere tutte le modalità operative e come impostarle, invitiamo a fare riferimento al datasheet del display in questione, oppure anche di un qualsiasi altro modulo che utilizzi il protocollo HD44780-compatibile (la quasi totalità dei display alfanumerici). In esso vi troverete una tabella con la lista dei comandi di controllo e di visualizzazione; per il datasheet del nostro modulo (serie 162B della *Displaytech*) questa tabella è chiamata "Control and Display Command", e ve ne riproponiamo la parte con i comandi essenziali in **Fig. 6**. Ogni riga mostra la descrizione di un comando ed il relativo valore da impostare sui bit di dato e del segnale RS (per tutti, 'L' significa *Low* quindi '0', 'H' sta per High, quindi '1', e 'X' significa che il valore è ininfluente). Vi è anche la colonna di un segnale chiamato R/W (Read/Write) che, se posto a '1', permette di leggere dal display i dati (come fosse una memoria) ed il flag di '*Busy*' (indica se il display è ancora "occupato" ad eseguire qualche operazione). Nella nostra demoboard il segnale R/W non viene gestito ed è fissato a zero, in modo che si possa solo scrivere sul display; siccome in tal modo non si sarà in grado di conoscere lo stato del flag 'Busy', inseriamo dopo ogni istruzione un adeguato ritardo per essere sicuri che il display non sia ancora occupato quando trasmettiamo l'istruzione successiva. Tornando alla tabella, essa mostra anche il tempo necessario all'esecuzione di ogni istruzione (colonna *Execution time*), e nel nostro programma abbiamo inserito ritardi "generosi" in modo da esser certi di non violare la tempistica del display in nessuna condizione. A tale proposito, questi ritardi sono stati calcolati per una frequenza massima di clock di 16MHz (che è anche la frequenza massima dell'ATMEGA16); se fate funzionare il micro con frequenze inferiori, potete ridurli in proporzione: in caso contrario il display funziona ugualmente ma per scrivere una riga intera (16 caratteri), se il clock è 1MHz ci vogliono oltre 200ms, che si cominciano a notare: la scritta non appare più "in un colpo solo" ma si intravede l'avanzamento dei caratteri lungo la riga.

LE FUNZIONI DI COMANDO E VISUALIZZAZIONE

Per mettervi in grado di utilizzare la tabella

Control and Display Command

Fig. 6

Command	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Execution Time (f_{osc} = 250kHz)	Remark
DISPLAY CLEAR	L	L	L	L	L	L	L	L	L	H	1.64ms	
RETURN HOME	L	L	L	L	L	L	L	L	H	X	1.64ms	Cursor move to first digit
ENTRY MODE SET	L	L	L	L	L	L	L	H	I/D	SH	42µs	• I/D : Set cursor move direction (I/D: H Increase; L Decrease) • SH : Specifies shift of display (SH: H Display is shifted; L Display is not shifted)
DISPLAY ON/OFF	L	L	L	L	L	L	H	D	C	B	42µs	• Display (D: H Display on; L Display off) • Cursor (C: H Cursor on; L Cursor off) • Blinking (B: H Blinking on; L Blinking off)
SET DD RAM ADDRESS	L	L	H	DD RAM address							42µs	DD RAM Data is sent and received after this setting
WRITE DATA	H	L	Write Data								46µs	Write data into DD or CG RAM

X : Don't care

RS = 0: usare LCD_Command()

RS = 1: usare LCD_DisplayChar()

per personalizzare subito la visualizzazione del vostro display, supponiamo ad un certo punto del programma di volere accendere il cursore per vedere dove si trova, e di farlo lampeggiare; nella quarta riga della tabella di **Fig. 6**, chiamata "Display ON/OFF", troviamo i bit che gestiscono questa opzione: il bit C (*Cursor*) se posto a '1' accende il cursore e il bit B (*Blinking*) ne abilita il lampeggio. Il bit D deve naturalmente essere a '1' poiché accende il display. Pertanto gli otto bit di dato devono essere impostati come 0000 1111 (0x0F in esadecimale). Per trasmettere l'informazione al display usiamo le funzioni *già fatte* del nostro programma "ProvaDisplay.c"; poiché si tratta di un comando (segnale RS = 0, da tabella), dobbiamo usare la funzione `LCD_Command()` e pertanto scriveremo:

```
LCD_Command(0x0F);
```

Per spegnere nuovamente il cursore (ma lasciando acceso il display) scriveremo invece:

```
LCD_Command(0x0C);
```

Se volessimo poi "pulire" il display useremmo il comando della prima riga della tabella ("Display Clear"), in cui si vede che è a '1' solo il bit DB0, quindi l'istruzione sarebbe:

```
LCD_Command(0x01);
```

e così via.

Infine, per muovere il cursore in una data posizione (sulla quale visualizzare poi il carattere voluto), dobbiamo usare il comando "Set DD RAM address", caratterizzato dall'avere DB7 alto ed i restanti 7 bit che specificano la locazione voluta; per semplificare le cose, mostriamo in **Fig.** 7 il numero esadecimale da trasmettere al display per impostare il cursore in una delle 32 posizioni possibili (16 caratteri su due righe).

Ad esempio, per posizionarlo sul terzo carattere della prima riga dovremo scrivere la seguente riga di codice:

```
LCD_Command(0x82);
```

per metterlo sulla prima posizione della seconda riga:

```
LCD_Command(0xC0);
```

e così via.

Notate che se impostate il cursore in locazioni diverse da quelle specificate in **Fig. 7** (ad esempio tra 0x90 e 0xBF), il carattere successivamente scritto sarà "fuori schermo" e dunque non visualizzato.

Per scrivere il carattere da visualizzare (nella posizione corrente del cursore) dobbiamo inviarlo, codificato ASCII, al display mentre il segnale 'RS' è alto, per cui useremo la funzione `LCD_DisplayChar()`. Per scrivere la lettera 'a' l'istruzione è la seguente:

```
LCD_DisplayChar('a');
```

siccome il C trasforma il carattere tra singoli apici nel suo equivalente ASCII. Alternativamente dovremmo scrivere

```
LCD_DisplayChar(0x61);
```

poiché 61 esadecimale è infatti la codifica ASCII del carattere 'a'. Possiamo anche passare alla funzione una variabile:

```
LCD_DisplayChar(MiaVar);
```

stampa la codifica ASCII del valore della variabile `MiaVar`.

Utilizzando la funzione `LCD_DisplayChar()` abbiamo anche costruito la funzione `LCD_Di-`

splayString(), che visualizza una stringa (ovvero una sequenza di caratteri), a partire dalla posizione corrente del cursore; in tale funzione dobbiamo passare anche il numero di caratteri da scrivere: pertanto per visualizzare la parola "Ciao" dobbiamo scrivere:

```
LCD_DisplayString("Ciao", 4);
```

Al posto della parola esplicita, la funzione può anche accettare un puntatore ad una stringa di caratteri (sui cui chiarimenti rimandiamo ad un qualsiasi manuale del C standard).
Infine, per visualizzare il valore numerico di una variabile abbiamo la funzione DisplayChar3Num(): essa converte il numero (da 8 bit) passato come argomento in tre caratteri ASCII corrispondenti al codice dei numeri che lo compongono, e li manda al display. Ad esempio con l'istruzione:

```
DisplayChar3Num(57);
```

inviamo in sequenza al display i codici ASCII '0', '5', e '7' ed esso visualizzerà pertanto 057.
Nel programma utilizziamo questa funzione per visualizzare il valore della variabile "Contatore"; naturalmente possiamo modificarla per visualizzare solo due cifre (da 00 a 99) oppure al contrario farle accettare variabili più grandi (16 o 32 bit) adeguando il numero di cifre che deve visualizzare.

IL CONTRASTO DEL DISPLAY

Tutti i moduli LCD hanno un piedino (VO) per mezzo del quale, applicando una opportuna tensione, si può regolare il contrasto dei cristalli liquidi del display. Alcuni moduli necessitano di una tensione negativa rispetto a massa, mentre il nostro modello richiede, a temperatura ambiente (in tutti gli LCD il contrasto varia con la temperatura), di applicare su VO una tensione di circa 200mV.
Tale tensione è ricavabile agendo sul trimmer R9: regolate tale trimmer sino a quando non vedrete apparire la videata di **Fig. 3** (se avete caricato nel micro il programma "ProvaDisplay"), oppure quella di **Fig. 5** (se il display non è stato inizializzato, magari perché il programma caricato non lo prevede). Se invece il contrasto non è regolato bene il display avrà:

- tutti i pixel accesi se la tensione di VO è troppo bassa;
- tutti i pixel spenti se la tensione di VO è troppo alta.

Fig. 7

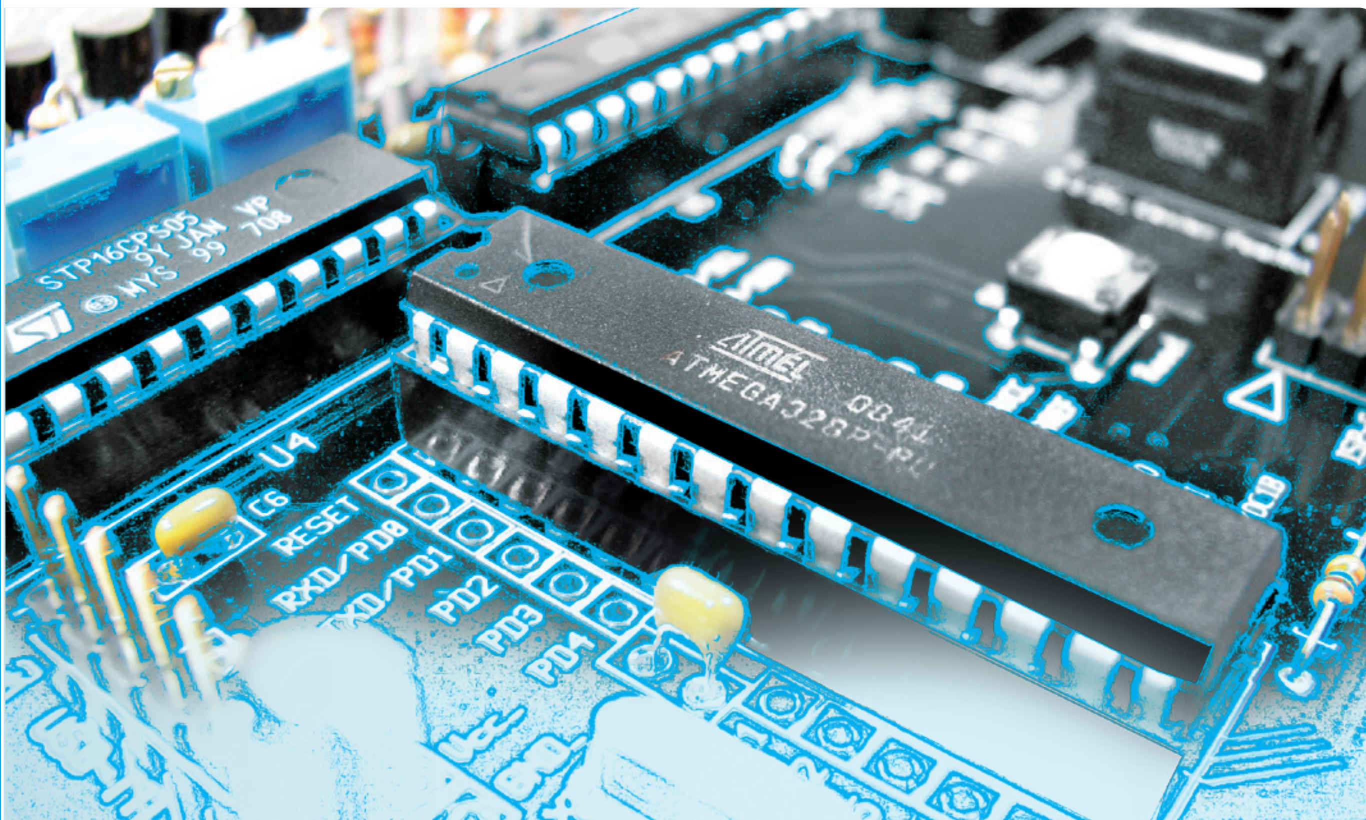
0	1		15		← Posizione display
80	81	- - - - - - - -	8E	8F	← Locazione riga 1
C0	C1	- - - - - - - -	CE	CF	← Locazione riga 2

LETTURA DEI TASTINI

Come potrete verificare eseguendo il programma "ProvaDisplay.c", per quanto velocemente premiate e rilasciate un tastino, la cifra visualizzata incrementa o decrementa di diverse unità, il che non è sbagliato poiché nonostante i ritardi introdotti per la visualizzazione dei caratteri, il processore esegue molte volte al secondo la verifica dei tastini e l'eventuale incremento/decremento della variabile.
Pertanto sarebbe meglio rendere il programma sensibile alle variazioni di stato del tastino, ovvero (consideriamo per brevità solo il tastino P1 collegato a PD2):

- se leggo '1' -> tastino non premuto: non fare nulla;
- se leggo '0' -> tastino premuto: aspetta che venga rilasciato;
- se leggo '1' -> tastino rilasciato: avanzamento contatore.

Quindi la cifra verrà incrementata solo quando il tasto è rilasciato. Nella pratica tuttavia si osserva che la cifra incrementa sempre più di una unità ad ogni rilascio del tastino, poiché esso è formato da un contatto tra due superfici metalliche che rimbalzano tra loro ogni volta che lo premiamo o lo rilasciamo: tali rimbalzi vengono interpretati dal programma (che è eseguito molto velocemente) come successive pressioni del tastino e quindi correttamente conteggiate.
Per evitare anche questo problema si inserisce un ritardo di (almeno) qualche decina di millisecondi tra le fasi sopra descritte, realizzando un semplice ma efficace meccanismo detto di antirimbalzo (*debouncing*) del tasto. Abbiamo così modificato "ProvaDisplay.c" nel programma "ProvaDisplay_debounce.c", che potete scaricare e sovrascrivere al primo. Siccome i tasti da verificare sono due, abbiamo raccolto il codice di controllo in una funzione chiamata ControllaTasto(), alla quale passiamo il nome del pin da verificare ed essa ci ritorna '1' se il tasto è stato premuto e rilasciato, *filtrando* tutti i rimbalzi. Alla funzione passiamo come argomento anche il port relativo al pin da testare; nel nostro programma è superfluo in quanto è sempre il PORTD, tuttavia l'abbiamo inserito nel caso voleste utilizzare questa funzione in vostri programmi in cui i pin da testare appartengano a port diversi.



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

Analizziamo il funzionamento e la gestione di un particolare sensore di temperatura che utilizzeremo per realizzare un termometro e un termostato con la nostra demoboard. Ottava puntata.

Dopo avere approfondito il funzionamento di alcune periferiche di input/output presenti sulla scheda demoboard come buzzer, LED, tasti e display alfanumerici, in questa puntata utilizzeremo sempre la nostra demoboard per realizzare un termometro e un termostato con un particolare sensore di temperatura.

SENSORE DI TEMPERATURA

La demoboard è predisposta per ospitare un sensore di temperatura, detto "termometro digitale", in quanto è un componente in grado di misurare e convertire il valore di temperatura in un numero digitale che può essere facilmente letto da qualsiasi microcontrollore. Il componente in questione è siglato DS18B20 ed è prodotto dall'azienda *Dallas Semiconductor*; si presenta esternamente come un normale transistor in contenitore TO-92 in quanto possiede tre soli piedini, dei quali due sono per l'alimentazione e uno serve per lo scambio dei dati digitali, un bit per volta (quindi in formato seriale).

Le sue caratteristiche salienti sono:

- comunicazione su singola linea, impegna solo un piedino di I/O del microcontrollore;
- nessuna necessità di componenti aggiuntivi per funzionare;
- misura temperature da -55 a +125 °C con una precisione di ±0,5 gradi da -10 a +85°C;
- risoluzione massima di conversione: 12 bit;
- EEPROM interna per memorizzare le impostazioni e due soglie di allarme.

Comando	Descrizione	Codifica
Convert T	Inizia una nuova fase di acquisizione della temperatura	44h
Read Scratchpad	Leggi i registri dello scratchpad	BEh
Write Scratchpad	Scrivi dati nei registri 2, 3 e 4 dello scratchpad	4Eh

Fig. 1

I lettori abituali di *Elettronica In* lo conosceranno sicuramente già, in quanto grazie alle sue ottime caratteristiche è stato adottato in numerosi progetti pubblicati dalla rivista. In questa sede analizzeremo nel dettaglio il suo funzionamento e soprattutto il codice necessario per trasformare la demoboard in un termometro, sulla base del quale potrete sviluppare nuove applicazioni a vostro piacimento.
Il DS18B20 viene gestito trasmettendo dei comandi di richiesta informazioni, per poi leggerne la risposta; ad esempio, le fasi per poter leggere il valore di temperatura sono:
1) Trasmissione comando di inizio conversione (micro → DS18B20);
2) Attesa tempo di conversione;
3) Trasmissione comando di lettura temperatura (micro → DS18B20);
4) Lettura valore temperatura (DS18B20 → micro).

In pratica il sensore rimane normalmente inattivo in attesa di un comando; quando riceve quello di inizio conversione, acquisisce il valore di temperatura e lo trasforma in un numero digitale che mantiene in due registri interni. Il microcontrollore, per leggere il risultato, deve mandare al sensore un comando di lettura registri, in risposta al quale il DS18B20 trasferirà al micro il numero binario corrispondente al valore di temperatura acquisito.
Ogni comando è codificato con un apposito byte, in modo che il sensore "capisca" cosa gli si stia chiedendo di fare. La **Fig. 1**, ripresa dal datasheet del sensore (che invitiamo a scaricare da [1]), mostra i tre comandi necessari per poter comunicare col sensore ed eseguire le operazioni elencate sopra. Per inciso, il DS18B20 ha diversi altri comandi (lettura stato alimentazione, trasferimento dati da e per EEPROM interna, ecc.) che però non utilizzeremo in questa sede; con le funzioni di comunicazione che mostreremo più avanti, il lettore interessato sarà comunque in grado di provare tutte le possibilità offerte da questo sensore.

Lo *scratchpad* nominato in **Fig. 1** è l'insieme dei registri presenti nel sensore: essi sono locazioni di memoria RAM, ciascuno da 1 byte; la **Fig. 2** mostra la composizione dello scratchpad.
- Byte 0 e 1: contengono il numero binario (su 16 bit totali) corrispondente al valore di temperatura acquisito (T). Questi registri vengono aggiornati dopo l'operazione di conversione;
- Byte 2 e 3: in essi si possono memorizzare due numeri T_L e T_H (a 8 bit ciascuno) tali che se la temperatura acquisita (T) va all'esterno di tali valori ($T < T_L$ oppure $T > T_H$) viene attivato un flag di allarme interno, che si può poi leggere dal microcontrollore;
- Byte 4: è un registro di configurazione, e serve solo per impostare la risoluzione di conversione del sensore;
- Byte 5-7: sono *riservati*, nel senso che il costruttore non specifica a cosa servano e consiglia di "non toccarli" (in particolare non bisogna scrivervi nulla);
- Byte 8: *checksum* dei dati presenti nello scratchpad, permette di verificare se essi vengono letti correttamente dal microcontrollore.

I byte 2, 3 e 4 sono collegati alla EEPROM, che come vi ricorderete è una memoria non volatile (non perde i dati togliendo l'alimentazione al sensore), quindi il loro contenuto può esservi scritto e poi recuperato, anche se fra le due operazioni viene spento il sistema. Il salvataggio di quei tre byte in EEPROM non è tuttavia automatico, in quanto bisogna trasmettere al sensore il comando "*Copy scratchpad*"; noi comunque non useremo questa caratteristica.
Nella **Fig. 2** sono indicati tra parentesi i valori che assumono i registri dopo l'accensione del sensore; in particolare si nota che i primi due, dove viene scritta la temperatura acquisita, caricano i valori 0x50 e 0x05, corrispondenti ad

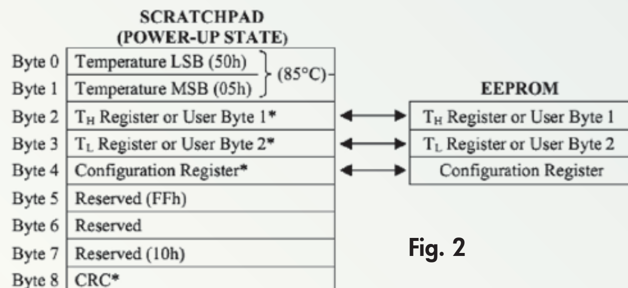


Fig. 2

una temperatura di 85°C. Tale valore permane finché non viene attivato dal micro un comando di inizio acquisizione ("*Convert T*" in base alla notazione di **Fig. 1**): quando la conversione terminerà, i due registri conterranno il valore reale della temperatura a cui si trova il sensore; tale valore rimarrà immutato fino all'acquisizione successiva.
Abbiamo così descritto i passi necessari "ad alto livello" per fare leggere dal sensore la temperatura a cui si trova e trasferire il risultato al microcontrollore. Scendendo verso il "basso livello", bisogna ancora capire come *viaggiano* le informazioni tra il micro ed il sensore stesso, che come abbiamo visto dispone di un'unica linea per i dati.

IL PROTOCOLLO 1-WIRE

Il protocollo di comunicazione è stato battezzato "1-Wire" ad intendere che la trasmissione dei dati (anche bidirezionale) tra due dispositivi avviene utilizzando un'unica linea. Per fare questo è necessario che i due dispositivi siano in grado di forzare la linea a massa per scrivervi lo '0' logico, mentre per avere l''1' logico i dispositivi rilasciano la linea (come se si scollegassero fisicamente), ed in essa viene a stabilirsi il livello alto grazie ad un resistore di *pull-up* esterno (sulla demoboard è R11).
Il disegno dei due dispositivi nella **Fig. 3** riassume questi concetti: la linea è a '0' se anche uno solo degli interruttori è chiuso, mentre per portarsi a '1' (lo stato di *riposo*) entrambi gli interruttori debbono essere aperti.
In tal modo viene anche scongiurato il pericolo di avere una condizione di *conflitto* che si potrebbe invece verificare se i dispositivi fossero anche in grado di applicare il livello logico '1' alla linea collegandola direttamente al +5V (invece di affidarsi al pull-up esterno): se uno di essi forzasse tale stato mentre l'altro dispositivo pilota nel contempo la linea verso massa, si verificherebbe un cortocircuito sull'alimentazione!
Dalle forme d'onda (a, b, c, d) della **Fig. 3** si osserva che l'inizio della trasmissione di ogni singolo bit avviene sempre ad opera di uno dei due dispositivi, detto, per questo motivo, *master* (che è il microcontrollore, mentre il sensore è detto *slave*): esso porta la linea dallo stato di riposo (alto), in cui si trova normalmente, allo stato '0' (linea rossa), e ve la mantiene per un breve tempo. Trascorso tale tempo, vi sono due possibilità in base alla direzione della comunicazione:

1) micro → DS18B20. Il micro rilascia subito la linea (che torna alta) se vuole trasmettere un '1' (**Fig. 3** – b), oppure la tiene ancora bassa per un certo tempo sufficiente affinché il sensore possa leggere lo '0' logico (**Fig. 3** – a);
2) DS18B20 → micro. Il micro rilascia subito la linea e si pone in lettura, poiché la linea può ancora essere bassa in quanto è il sensore che nel frattempo è passato a forzarla in quello stato, per trasmettere uno '0' logico (**Fig. 3** – c). Se invece, appena rilasciata dal micro la linea torna subito alta, il sensore vuole trasmettere un '1' (**Fig. 3** – d).

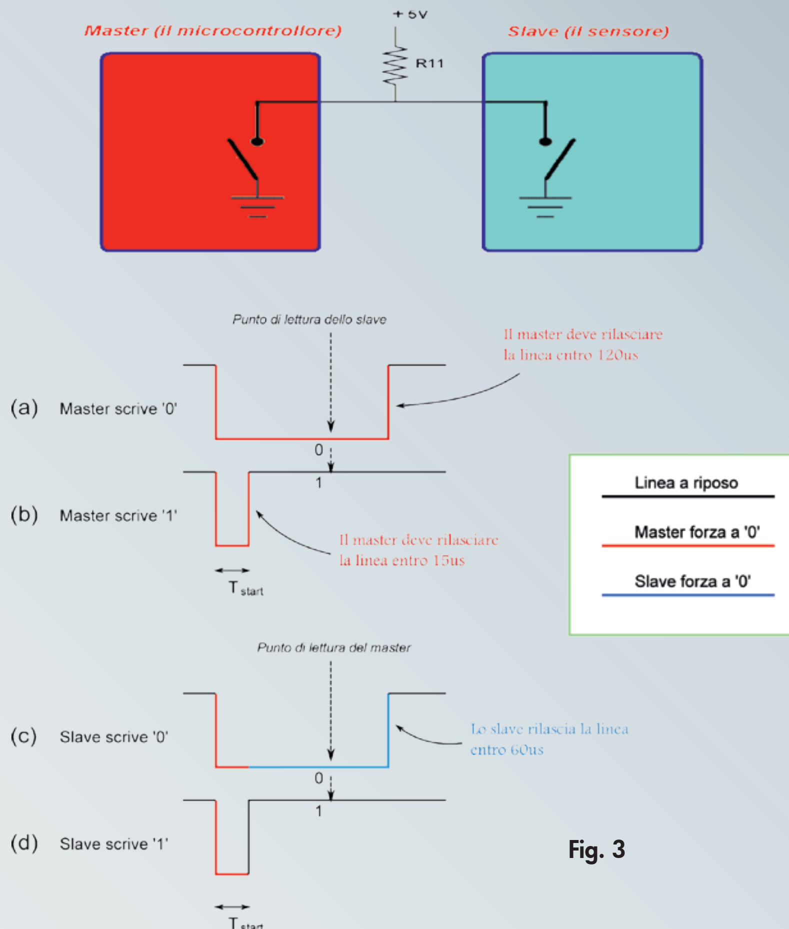


Fig. 3

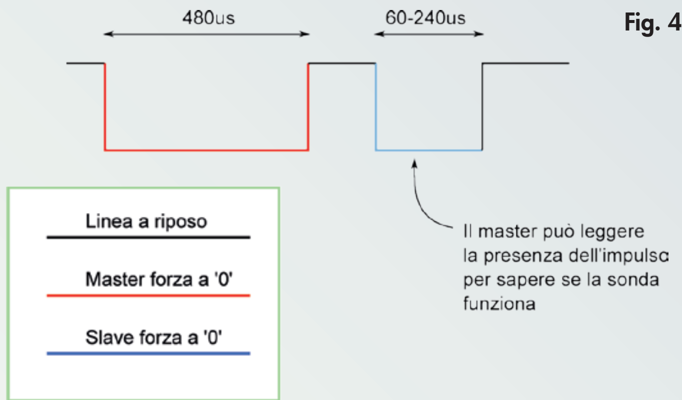


Fig. 4

Si comprende pertanto come sia possibile comunicare un dato in maniera bidirezionale; l'unico punto critico è il rispetto delle tempistiche da parte del master: esso infatti deve fornire un impulso di "inizio bit" di breve e precisa durata (T_{start} in **Fig. 3**), in quanto il datasheet del sensore impone che debba essere maggiore di 1µs e minore di 15µs, in quanto dopo tale tempo sulla linea viene imposto il valore vero del bit dal dispositivo che in quella fase deve scrivere. Anche la durata complessiva del ciclo ha limiti precisi: nel caso in cui il master scriva '0', entro 120µs deve comunque rilasciare la linea. Queste considerazioni sono da tenere presenti quando scriviamo il programma, in quanto dobbiamo preventivamente conoscere la frequenza di lavoro della CPU per regolarci nel calcolo delle funzioni di ritardo in modo da rispettare le tempistiche sopra descritte. Se scegliamo ad esempio di utilizzare il quarzo esterno da 16MHz (e dimensioniamo di conseguenza i ritardi), il programma non potrà poi comunicare con la sonda se mettiamo sulla scheda un quarzo da 8MHz!
Adesso che abbiamo conosciuto il meccanismo base di comunicazione dei bit nel protocollo *1-Wire*, si intuisce facilmente che per trasmettere un intero comando oppure un dato, lunghi ciascuno otto bit (un byte) basta ripetere otto volte le fasi sopra descritte, aggiungendo solo una breve pausa (in cui la linea rimane a riposo, cioè '1') tra un bit ed il successivo (il datasheet richiede almeno 1µs).

LA SEQUENZA D'INIZIALIZZAZIONE

Il riconoscimento della fase di scrittura o lettura dei byte è automatico, in quanto il master inizia sempre il dialogo trasferendo un comando (fase di scrittura) allo slave: se questi è un comando di lettura, come ad esempio il "Read Scratchpad" in **Fig. 1**, dal byte successivo la comunicazione avviene dallo slave verso il master, sino a quando tutti i byte dello scratchpad non sono stati trasferiti, oppure il master non inserisce un reset sulla linea.
In questo protocollo un segnale di reset può essere inviato dal master e consiste nel forzare bassa la linea per almeno 480µs e poi rilasciarla: si tratta in pratica di un impulso a '0' molto più lungo di quelli di inizio bit durante una comunicazione. Lo slave pertanto lo riconosce, interrompe una eventuale comunicazione in corso e si pone in stato di attesa di un nuovo comando dal master; pertanto il datasheet consiglia di inserirlo prima di ogni nuova comunicazione. Peraltro l'impulso di reset è utile anche per sapere se lo slave è "vivo", poiché in risposta a tale impulso esso deve rispondere con il cosiddetto "impulso di presenza" (*presence pulse*) ovvero forzando a sua volta la linea a '0', dopo che il master l'ha rilasciata, per un breve periodo (60-240µs); pertanto se dopo l'impulso di reset il master non vede tornare bassa la linea, vuol dire che lo slave non risponde perché guasto o scollegato dal bus. In **Fig. 4** è mostrata l'intera sequenza, cui è dato il nome di sequenza d'inizializzazione, consistente degli impulsi di reset e del conseguente impulso di presenza.
Dopo aver analizzato le caratteristiche di base del sensore DS18B20 (con riferimento al datasheet per la spiegazione delle sue funzioni supplementari ed il dettaglio del funzionamento e delle tempistiche del bus 1-Wire), è ora di "sporcarsi le mani" con la scrittura di un programma pratico per poterlo utilizzare.

IL PROGRAMMA DEL TERMOMETRO

Dal sito di *Elettronica In* potete scaricare il programma sorgente chiamato "Termometro.c", che legge la temperatura ambiente tramite il sensore DS18B20 e la visualizza sul display LCD (**Fig. 5**).
In questo programma viene verificato anche l'impulso di presenza del sensore, per sapere se è collegato e funzionante: in caso contrario, al posto del valore di temperatura viene visualizzata la stringa "---".
Come abbiamo già visto il sensore impiega un certo tempo, dopo aver ricevuto il relativo comando, ad effettuare la conversione di temperatura; in **Fig. 6** mostriamo la relazione tra la risoluzione della sonda (e l'impostazione relativa da assegnare ai bit del *Configuration*

register), ed il tempo di conversione corrispondente. Nel nostro termometro scegliamo di visualizzare solo i gradi interi, quindi non ci servono i bit decimali della conversione; dalla **Fig. 6** si osserva che possiamo impostare il sensore per la minima risoluzione (entrambi i bit R1, R0 del *Configuration register* a 0) e anche in questo caso il DS18B20 ci fornisce comunque un bit decimale (che potrebbe essere utile per visualizzare il mezzo grado), ed in questo caso il tempo di conversione è di "soli" 94ms, che per il microcontrollore è comunque un intervallo enorme. Pertanto, invece di tenere ferma la CPU per tutto quel tempo con un consueto ciclo di ritardo, possiamo utilizzare il Timer/Counter 1 impostandolo per generare una interruzione ogni 100ms, e nella isr relativa andiamo a gestire il sensore, mandandogli per ultimo il comando di nuova conversione: alla isr successiva (quindi dopo 100ms), come prima cosa possiamo quindi prelevare dal sensore il valore di temperatura che sarà ormai disponibile; fatto ciò, prima di uscire dalla isr, gli inviamo un nuovo comando di conversione, e così via. Nel programma, in realtà, abbiamo impostato il T/C1 per avere una interruzione al secondo (impostando OCR1A = 15000), poiché potrete verificare che il sensore, possedendo una certa inerzia termica, non è in grado di apprezzare veloci variazioni, ed un aggiornamento del valore ogni secondo è più che sufficiente; inoltre in un normale ambiente è realisticamente molto improbabile avere variazioni di temperatura di un grado al secondo!

LE FUNZIONI DI GESTIONE DEL SENSORE

In modo simile a quanto fatto per il display, abbiamo approntato delle funzioni di "basso livello" per gestire le operazioni di comunicazione 1-Wire con la sonda che sono essenzialmente la scrittura, la lettura e l'impulso di reset della sequenza d'inizializzazione:

```
void DS_ScriviByte(uint8_t Dato);
/* Scrittura di un byte alla sonda */
uint8_t DS_LeggiByte(void);
/* Lettura di un byte dalla sonda */
uint8_t DS_Reset(void);
/* Inizializzazione sonda */
```

Nei loro nomi abbiamo volutamente inserito il prefisso "DS" per rimarcare che esse sono relative al sensore DS18B20. Nel programma troverete il codice sorgente di queste funzioni ben commentato, quindi non ci addentreremo nella loro descrizione. È utile invece soffermarsi sulle funzioni di ritardo necessarie a gestire le tempistiche della comunicazione 1-Wire (e non solo) che, come abbiamo visto, sono abbastanza stringenti: nel programma utilizziamo tre funzioni di cui riportiamo la definizione:

```
void Ritardo_1us(void);
/* ritardo di 1us a 16MHz */
void Ritardo_15us(void);
/* ritardo di 15us a 16MHz */
void Ritardo_100us(void);
/* ritardo di 100us a 16MHz */
```

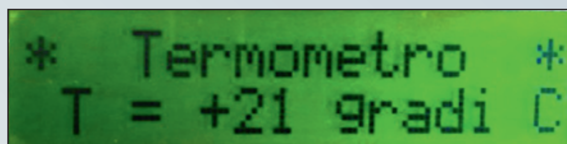
La funzione "base" è quella del ritardo di 1µs; le altre, di ritardo maggiore, sono ottenute chiamando più volte la funzione base. Questa funzione è composta da tre righe di codice del tipo:

```
asm volatile("nop\n\t""nop\n\t""nop\n\t""nop\n\t":);
```

Per il compilatore *avr-gcc*, la direttiva "`asm volatile()`" indica che il codice contenuto tra parentesi contiene istruzioni in linguaggio Assembly che come tali andranno tradotte nel corrispettivo codice oggetto (come farebbe un programma *Assembler*) ed inserite direttamente nell'immagine eseguibile del programma, quindi senza essere processate dal compilatore stesso: nel nostro caso, questa riga di codice ci permette di inserire nel codice quattro istruzioni dette in Assembly "NOP" (per approfondire aspetti sulla sintassi e l'utilizzo della direttiva "`asm`" rimandiamo sempre al documento *avr-libc-user-manual.pdf*.

L'istruzione NOP (che significa No OPerate) tiene semplicemente "occupato" il processore

Fig. 5



BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0	R1	R0	1	1	1	1	1

Configuration register

R1	R0	Risoluzione (bits)	Numero bits decimali	Tempo conversione
0	0	9	1	93.75ms
0	1	10	2	187.5ms
1	0	11	3	375ms
1	1	12	4	750ms

Fig. 6

Listato 1

```
if (RegistroFlag & (1 << STATO_USCITA)) {   /* uscita attiva: verifico se
devo spegnerla */
if (ModuloTemperatura > SetPoint)
        RegistroFlag &= ~(1 << STATO_USCITA); /* STATO_USCITA = 0 */
}
else {   /* uscita disattiva: verifico se devo accenderla */
if (ModuloTemperatura < SetPoint)
                RegistroFlag |= (1 << STATO_USCITA); /* STATO_USCITA = 1 */
}
```

per un ciclo di clock (che è il tempo che impiega ad eseguirla), ed è quindi in pratica il più piccolo ritardo che possiamo ottenere: se il clock della CPU è prelevato dal quarzo esterno a 16MHz, otterremo un ritardo esatto di 1/16 MHz = 0,0625µs. Capite quindi come anche un'istruzione che apparentemente non fa nulla possa essere utile: non a caso quasi tutti i microcontrollori annoverano tra il loro set di istruzioni anche la NOP.
Pertanto la riga di codice vista sopra, che inserisce quattro NOP, produce un ritardo complessivo di 0,0625 x 4 = 0,25µs. Nella funzione `Ritardo_1us()` vi sono tre di queste righe, dunque il ritardo complessivo vale teoricamente 0,25 x 3 = 0,75µs; tuttavia anche la chiamata ad una funzione ed il suo ritorno al chiamante portano via del tempo, non trascurabile quando parliamo di microsecondi; pertanto, sommando tutto otteniamo che il ritardo complessivo di `Ritardo_1us()` vale in realtà 1,875µs. Peraltro è anche chiara la dipendenza dalla frequenza di clock: se mettessimo un quarzo a 8MHz (oppure volessimo utilizzare l'oscillatore interno del microcontrollore a 8MHz), il tempo di esecuzione dei codici NOP raddoppierebbe e quindi anche il ritardo della funzione `Ritardo_1us()` diventerebbe matematicamente di 3,75µs: per ridurlo dovremmo dunque "rimodulare" la funzione diminuendo il numero di NOP inserite.
Nota: in questa applicazione non consigliamo l'utilizzo dell'oscillatore interno in quanto a differenza del quarzo ha una tolleranza molto più ampia (±3% ma è variabile anche con la temperatura), e questo provocherebbe una conseguente imprecisione sui ritardi del programma.

DAL TERMOMETRO AL TERMOSTATO

Dopo aver appreso come leggere una temperatura con il sensore DS18B20 possiamo fare un passo in più e, aggiungendo qualche riga di codice al programma, trasformare la demoboard in un termostato elettronico. Il codice aggiuntivo rispetto al termometro serve nella pratica per confrontare il valore di temperatura acquisito dal sensore con un riferimento prefissato (detto *setpoint*), per decidere se attivare o meno un'uscita che può ad esempio pilotare la caldaia del riscaldamento; se chiamiamo `SetPoint` e `Temperatura` rispettivamente il valore prefissato e la temperatura attuale, il nostro algoritmo è il seguente:

- Se (`Temperatura > SetPoint`) spegni l'uscita
- altrimenti accendila.

Questo algoritmo è semplice ma ha un problema: supponendo di avere impostato il setpoint a 20°C, se la temperatura ambiente arriva a 21°C l'uscita viene spenta, per poi riaccendersi quando la temperatura scende a 20°C; nella realtà la temperatura potrebbe trovarsi a 20,99°C ed il sensore, causa il suo errore di misura (piccolo ma non nullo) e l'arrotondamento, produrrebbe in uscita alternativamente i valori 20-21-20-21-… ad ogni conversione (ogni secondo nel nostro programma). Pertanto l'uscita del termostato potrebbe trovarsi a commutare continuamente il suo stato (on-off-on-off-…), il che sarebbe senz'altro poco gradito dal dispositivo ad essa collegato. Per ovviare a questo problema si utilizzano in genere tre metodi, anche combinati tra loro:

1) Mediare la temperatura letta dalla sonda, facendo la somma di N acquisizioni e poi dividendo il risultato per N; in tal modo il risultato sarà tanto più stabile quanto più campioni utilizzeremo per la somma;
2) Inserire un ritardo dopo lo spegnimento: l'uscita non può tornare accesa fin tanto che non è trascorso un certo periodo dopo il precedente spegnimento, anche se la temperatura lo richiedesse. Questo metodo è adottato nei frigoriferi, nei quali la ripartenza non è possibile se il gas dell'impianto refrigerante si trova ancora in pressione dal ciclo precedente;
3) Adottare un'isteresi, ovvero differenziare le soglie di accensione e di spegnimento.

Nel nostro programma abbiamo adottato quest'ultima soluzione, che può comunque essere integrata con le altre due: il lettore

interessato può provare ad implementare le modifiche necessarie!

L'ISTERESI

L'utilizzo dell'isteresi consiste nel modificare l'algoritmo "decisionale" del termostato come segue:

- L'uscita è attiva: allora spegnila se (`Temperatura > SetPoint + X`);
- L'uscita è disattiva: allora accendila se (`Temperatura < SetPoint - X`).

Ad esempio, se X = 1 e setpoint = 20, l'uscita si spegnerà quando la temperatura raggiunge i 22°C (ovvero diventa maggiore di (`SetPoint + X`) = 21) e si riaccenderà quando la temperatura scende a 18°C (ovvero diventa minore di (`SetPoint - X`) = 19).
Naturalmente il valore X può essere aumentato tanto più è grande l'imprecisione di lettura della temperatura, tenendo presente che questo degraderà l'accuratezza della termostatazione: nell'esempio sopra, con setpoint=20 e X=1, la temperatura ambiente sarà sempre oscillante tra i 18 e i 22°C. Nel nostro caso, siccome la lettura della sonda non avrà mai un'imprecisione superiore al grado, poniamo X=0 per cui, sempre considerando setpoint=20, l'uscita si spegne a 21 e si riaccende a 19°C. Nel listato 1 riportiamo il brano di codice che realizza quanto discusso.
La variabile `ModuloTemperatura` contiene il valore di temperatura, espresso in gradi, senza segno, in quanto supponiamo di gestire solo temperature positive, come avviene normalmente in un termostato domestico; potete comunque perfezionare il programma per poter lavorare anche con valori negativi. Questa variabile viene confrontata con il valore di temperatura prefissata (variabile `SetPoint`), e lo stato dell'uscita (che manteniamo in un bit chiamato '`STATO_USCITA`', contenuto nella variabile `RegistroFlag`) viene aggiornato: se (`ModuloTemperatura > SetPoint`) spegniamo l'uscita, mentre se (`ModuloTemperatura < SetPoint`) la accendiamo.
Volendo si possono scambiare gli operatori '<' e '>' per invertire la logica di funzionamento, ovvero accendere l'uscita quando la temperatura sale oltre al setpoint, e utilizzare il termostato per controllare un frigorifero oppure un condizionatore.
Per completare il nostro termostato dobbiamo ora decidere il modo con cui immettere nel microcontrollore il valore del setpoint desiderato, in quanto non deve essere una costante dichiarata nel programma (che quindi richiederebbe una ricompilazione e riprogrammazione per poterla modificare) ma una variabile che l'utente possa cambiare a proprio piacimento mentre la scheda è in funzione. I metodi con cui possiamo passare questa informazione al micro sono fondamentalmente due: analogico e digitale.

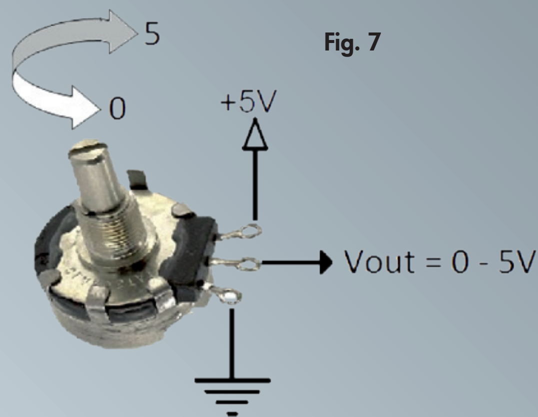
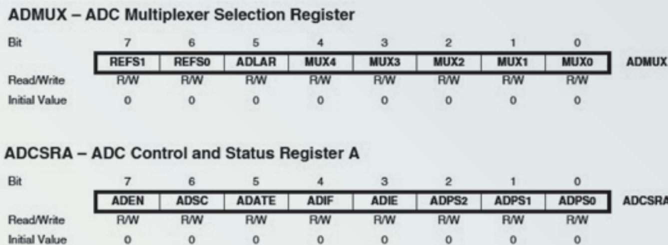


Fig. 7

IL SETPOINT "*ANALOGICO*" E IL CONVERTITORE A/D

Con il metodo analogico il setpoint si regola agendo su di un potenziometro, in modo da ricavare una tensione proporzionale al valore di temperatura desiderato. Se colleghiamo infatti un qualsiasi potenziometro (purché sia lineare e di valore compreso da 4,7KΩ a 1MΩ) come mostrato in **Fig. 7**, ruotando il suo alberino tra un estremo e l'altro ricaveremo sul suo piedino centrale (il cursore) una tensione compresa tra 0 e 5V, a cui possiamo associare un intervallo di valori del setpoint, ad esempio 0°C con 0V e 30°C con 5V.
Come facciamo però a trasformare il valore di tensione (che contiene l'informazione di temperatura desiderata) in un numero binario che il programma possa elaborare (la variabile `SetPoint`)? La risposta sta nell'utilizzo di una periferica del microcontrollore chiamata *convertitore analogico/digitale* (abbreviato in *convertitore A/D* oppure *ADC*) che, come si può desumere dal nome, fa proprio questo "mestiere": la tensione posta al suo ingresso viene convertita in un proporzionale numero binario di 10 bit (che può rappresentare numeri decimali da 0 a 1023). Più esattamente, questo numero va da 0 (se la tensione in ingresso è 0V) a 1023 se la tensione in ingresso è pari ad un valore V_{REF} (tensione di riferimento), impostabile da



ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig. 8

programma oppure applicabile su un apposito piedino del microcontrollore, che fissa il *fondo scala* del convertitore, ovvero la massima tensione che può ricevere in ingresso e che produce il numero massimo in uscita (1023 nel nostro convertitore). Più in generale, la relazione tra la tensione d'ingresso V_I ed il numero N (da 10 bit) prodotto in uscita è la seguente:

$$N = (1024 \times VI) / V_{REF}$$

Poiché il potenziometro collegato come in **Fig. 7** può fornire una tensione massima di 5V, scegliamo tale valore anche come riferimento: VREF = 5V (nota: V_{REF} non può mai essere superiore alla tensione di alimentazione del microcontrollore). Il numero N prodotto all'uscita del convertitore A/D viene salvato in due registri (chiamati ADCH e ADCL) dai quali il programma può leggerlo ed utilizzarlo, ed in base a quanto esposto sopra sarà un valore compreso tra 0 se il potenziometro è ruotato tutto da una parte (0V sul piedino centrale) e 1023 se il potenziometro è ruotato dalla parte opposta (5V). Per convertirlo in un valore coincidente con la temperatura che vogliamo rappresenti (0 → 0 gradi … 1023 → 30 gradi) è sufficiente dividerlo per un numero X tale che a fondo scala si ottengano 30°C ovvero: 1023 / X = 30, da cui ricaviamo X = 34,1.
Detto questo vediamo nel dettaglio come attivare il convertitore A/D, che potrete naturalmente utilizzare in ogni vostra applicazione in cui vi sia la necessità di leggere delle tensioni analogiche. Come già visto per le altre periferiche viene anch'esso controllato da alcuni registri del microcontrollore da inizializzare con valori opportuni che qui andremo a illustrare brevemente, mentre per i dettagli rimandiamo come di consueto al datasheet dell'ATMEGA16, al capitolo "Analog to Digital Converter"; inoltre le sue caratteristiche elettriche (tensione massime d'ingresso, precisione, ecc.) sono mostrate nel paragrafo "ADC Characteristics" sotto il capitolo "Electrical Characteristics".
Fondamentalmente i registri da impostare sono due: ADMUX e ADCSRA, riportati in **Fig. 8**.
Il primo stabilisce:
- Il canale d'ingresso da convertire (con i bit MUX<4:0>);
- la tensione di riferimento da utilizzare (con i bit REFS<1:0>).

L'ingresso del convertitore A/D dell'ATMEGA16 fa capo all'uscita di un multiplexer (un selettore) collegato a sua volta agli 8 pin di I/O del PORTA; il multiplexer, sulla base del numero scritto in MUX<4:0>, collega uno di questi pin con l'ingresso del convertitore: pertanto si può scegliere (da programma) su quale di tali pin effettuare la conversione. Nel datasheet è riportata una tabellina che mostra le associazioni tra il valore di MUX<4:0> ed il canale d'ingresso; nel nostro circuito abbiamo collegato il potenziometro su PA0, quindi poniamo MUX<4:0> = 00000, ma potreste anche scegliere un altro canale e cambiare MUX<4:0> di conseguenza. Notare che se MUX<4:0> è compreso tra a 01000 (8 decimale) e 11101 (29) il convertitore permette di selezionare *canali differenziali* (la tensione da convertire viene ricavata dalla differenza delle tensioni su due differenti piedini), però tale caratteristica non è stranamente presente (o testata) sui microcontrollori in contenitore DIP40 (quello utilizzato sulla demoboard) ma solo su quelli SMD, quindi non potete utilizzarla.
I bit REFS<1:0> permettono di selezionare la sorgente della tensione di riferimento (VREF), che in base alla relativa tabellina sul datasheet "Voltage Reference Selections for ADC" può derivare da:
- REFS<1:0> = 00 → piedino esterno AREF;
- REFS<1:0> = 01 → alimentazione del convertitore A/D (piedino AVCC);
- REFS<1:0> = 11 → riferimento interno a 2,56V.

Siccome in base a quanto detto sopra la V_{REF} dev'essere impostata a 5V, selezioniamo AVCC che nella demoboard è appunto collegato a tale tensione.
Notare che, per potere impostare V_{REF} a tensioni diverse, nella demoboard il piedino AREF è stato collegato ad un partitore composto dai resistori R14 e R15, tramite i quali si può creare qualunque tensione da 0 a 5V in base alla seguente formula:

$V_{REF} = 5 \times R15 / (R14 + R15)$

Ricordatevi anche che per utilizzare questa tensione bisogna porre REFS<1:0> = 00. Infine il registro ADMUX contiene il bit ADLAR che, se forzato a '1', ci permette di leggere gli 8 bit più significativi del risultato dal registro ADCH, e i rimanenti 2 bit (ricordiamo che il convertitore è a 10 bit) sul registro ADCL; nel nostro programma scartiamo questi ultimi e consideriamo solo il contenuto di ADCH, riducendo di fatto la risoluzione di conversione a 8 bit, che però sono ampiamente sufficienti: abbiamo 256 livelli per un setpoint che va da 0 a 30°C, dunque la minima differenza apprezzabile del numero in ADCH vale circa 0,11 gradi. Poiché consideriamo nel nostro termostato solo gradi interi, per convertire il valore della conversione (su 8 bit: 0…255) nel setpoint (scala 0…30) dobbiamo fare, come già visto, una divisione:

```
SetPoint = ADCH / 8.5;
```

Passiamo al registro ADCSRA; esso contiene i seguenti bit:

- ADEN: abilitazione convertitore: va posto a '1' per poterlo utilizzare;
- ADSC: quando vi viene scritto '1' fa iniziare una nuova conversione;
- ADATE: serve per fare partire una conversione da segnali esterni (non lo utilizziamo);
- ADIF: è il flag che segnala la fine di una conversione precedentemente iniziata (con ADSC). Infatti il convertitore impiega un certo tempo per elaborare il risultato, e bisogna testare questo flag per sapere quando ha finito. Volendo può anche essere generato un interrupt, e la lettura del risultato viene fatta dalla isr relativa. Come gli altri flag dell'ATMEGA16, anche questo può essere riazzerato scrivendovi un '1' (se invece si utilizza l'interrupt l'azzeramento è automatico);
- ADIE: se '1' il flag ADIF genera una interruzione e bisogna scrivere nel programma la relativa isr di gestione;
- ADPS<2:0>: selezionano il fattore di divisione di un prescaler, dal quale il convertitore ricava la frequenza di funzionamento (che il datasheet consiglia di tenere compresa tra 50 e 200KHz) a partire dal clock di sistema. Nel nostro programma decidiamo di dividere per 128 (ADPS<2:0> = 111) in modo da ottenere la frequenza di 125KHz con il quarzo esterno da 16MHz.

In definitiva, per impostare il convertitore A/D inizializziamo i suddetti registri come segue:

```
ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 <<
ADPS1)| (1 << ADPS0);
ADMUX = (1 << REFS0) | (1 << ADLAR);
```

Dopo aver "lanciato" una nuova conversione, attendiamo il suo completamento testando continuamente il flag ADIF del registro ADCSRA con la seguente espressione, dalla quale il programma non esce fino a che non diventa '1':

```
while (!(ADCSRA & (1 << ADIF))); /* attendi che
ADIF diventi '1' */
```

Come sappiamo si poteva scegliere una via più efficiente (che non dovesse tenere il programma fermo a controllare il flag), ovvero abilitare l'interrupt del convertitore A/D per eseguire automaticamente la sua isr, al termine della conversione, in cui andare a leggere il risultato; tuttavia ogni singola conversione dura al più 25 cicli di clock (del convertitore) che si traducono, a 125KHz, in un tempo di (1/125000) x 25 = 200µs: dunque il programma deve "attendere poco" e, almeno in questa applicazione, possiamo permettercelo!

IL SETPOINT "*DIGITALE*" E LA MEMORIA EEPROM

Con il metodo *digitale* utilizziamo i due tastini della demoboard per modificare il valore della variabile `SetPoint`, in modo analogo a quanto facevamo con la variabile `Contatore` nel programma "ProvaDisplay" analizzato la scorsa puntata. Questo metodo ha un unico problema: ogni volta che il sistema viene spento si perde l'impostazione del setpoint, in quanto la variabile in questione risiede come le altre nella RAM del micro e dunque non conserva le informazioni senza tensione di alimentazione. Quindi alla successiva riaccensione essa potrebbe avere un valore qualsiasi (variabile sporca) oppure essere inizializzata dal programma ad un valore prefissato, che dunque non coinciderebbe con l'ultima impostazione desiderata.
Per fortuna il nostro microcontrollore possiede

Il calcolo "preciso" dei ritardi

Per ciascuna funzione dei nostri programmi (compresa la `main()`), è possibile conoscere il tempo esatto che il processore impiegherà ad eseguirla: per fare ciò bisogna sapere esattamente da quali e quante istruzioni in linguaggio macchina è composta, ed il tempo impiegato per eseguire ciascuna. Per fortuna, oltre a generare i file binari utilizzati dai programmatori (come l'"Intel-hex"), il compilatore crea anche un listato (*listing*) del programma in cui troviamo per ogni funzione la *traduzione* in codici macchina affiancata dal codice mnemonico (Assembly) corrispondente. Questo documento è un file di testo,

```
DDRA = 0xFE; /* PA<7:1> sono uscite (pin non collegati), PA<0> è
l'ingresso per il convertitore A/D */
 9a:  ea e3          ldi   r30, 0x3A  ; 58
 9c:  f0 e0          ldi   r31, 0x00  ; 0
 9e:  8e ef          ldi   r24, 0xFE  ; 254
 a0:  80 83          st    Z, r24
DDRB = 0xFF;  /* Tutti i pin di PORTB sono impostati come uscite */
 a2:  e7 e3          ldi   r30, 0x37  ; 55
 a4:  f0 e0          ldi   r31, 0x00  ; 0
 a6:  8f ef          ldi   r24, 0xFF  ; 255
 a8:  80 83          st    Z, r24
```

Fig. A

chiamato <nome progetto>.lss, che si trova nella sottocartella "default" del progetto (la stessa in cui vi è il file .hex per la programmazione del micro). Se non vi fosse un file con tale estensione accertatevi in AVR Studio che la casellina "Generate List File", contenuta nella finestra delle opzioni progetto (sotto Project → Configuration options → General), sia spuntata.
Come esempio pratico analizziamo il file di listing generato dalla compilazione del programma "Termostato_potenziometro.c", aprendolo con AVR Studio (File → Open file) oppure con qualunque editor di testo, e cercando la scritta "Disassembly of section .text" che indica l'inizio della sezione in cui è listato il codice oggetto (text identifica infatti la sezione di codice del programma, ovvero l'immagine eseguibile da caricare nella memoria flash del microcontrollore). Di lì in seguito si troverà il codice relativo a qualsiasi funzione utilizzata dal programma, preceduto da una riga con l'indirizzo di partenza ed il suo nome utilizzato nel sorgente (o nella libreria se la funzione non fa parte del programma C ma è inserita in fase di linking). Ad esempio, cercando la parola "main" troveremo la riga:

```
00000092 <main>:
```

ad indicare che vi è una funzione con tale nome ed in memoria viene allocata dall'indirizzo 92 (espresso in esadecimale). Seguono, intercalate, le righe ricopiate dal sorgente (compresi i commenti) e quelle con il codice vero e proprio; in figura riportiamo alcune di queste righe riprese dalla funzione `main()` che abbiamo colorato per evidenziarne le differenze di significato: le righe rosse sono ricopiate dal file sorgente, seguite dalla loro "traduzione" in linguaggio macchina ovvero i codici oggetto in verde (in esadecimale) e dal loro equivalente mnemonico in nero. Dopo il carattere ';' (che nei file in linguaggio Assembly indica l'inizio di un commento) il

```
0000080c <Ritardo_1us>:
void Ritardo_1us(void) { /* ritardo di 1us a 16MHz */
     80c:  df 93          push  r29                          2
     80e:  cf 93          push  r28                          2
     810:  cd b7          in    r28, 0x3d                    1
     812:  de b7          in    r29, 0x3e                    1
     ...
     asm volatile("nop\n\t""nop\n\t""nop\n\t""nop\n\t":);    4
     asm volatile("nop\n\t""nop\n\t""nop\n\t""nop\n\t":);    4
     asm volatile("nop\n\t""nop\n\t""nop\n\t""nop\n\t":);    4
}
     82c:  cf 91          pop   r28                          2
     82e:  df 91          pop   r29                          2
     830:  08 95          ret                                4
```

ret: fine funzione

cicli totali: 26

Fig. B

compilatore scrive l'equivalente decimale dei numeri (esadecimali) dell'argomento delle istruzioni, qualora presente. I numeri blu ad inizio riga indicano invece l'indirizzo (esadecimale) in cui tali istruzioni vengono allocate in memoria.
Dalla **Fig. A** notiamo che l'assegnazione di 0xFE a DDRA richiede quattro istruzioni in linguaggio macchina per essere eseguita: tre 'LDI' e un 'ST'. Per conoscere il significato di questi codici mnemonici bisogna fare riferimento al manuale "AVR Instruction Set" (reperibile in *http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf*), nel quale sono descritte in modo dettagliato tutte le istruzioni dei processori AVR, della cui famiglia fa parte anche l'ATMEGA16; in fondo alla loro descrizione sono specificati anche i cicli di clock necessari per l'esecuzione di ciascuna, alla voce "Cycles". Possiamo così scoprire che ogni 'LDI' impiega un ciclo e 'ST' ne impiega due, quindi per scrivere il valore 0xFE in DDRA occorrono 1 + 1 + 1 + 2 = 5 cicli di clock del processore, che con il clock prelevato dal quarzo esterno a 16MHz fanno 0,3125*µs*.
Con questo stesso sistema si può quindi anche calcolare o verificare il tempo esatto impiegato dalle funzioni di ritardo utilizzate nel nostro programma, come `Ritardo_1us()`. Questa funzione è chiamata in diversi punti del programma, che nel file di *listing* si identificano con le righe contenenti il seguente testo:

```
call    0x80c    ; 0x80c <Ritardo_1us>
```

Infatti per i processori AVR la chiamata a funzione si può eseguire con l'istruzione 'CALL' (*chiamata* a funzione) seguita dall'indirizzo di partenza della funzione stessa (80C è appunto l'indirizzo in cui la funzione `Ritardo_1us()` è stata allocata); il processore ritornerà dalla funzione quando incontrerà una successiva istruzione "RET" (*ritorno* da funzione) che dunque dev'essere l'ultima istruzione della funzione stessa.
Nella **Fig. B** abbiamo evidenziato nel listato della funzione `Ritardo_1us()` il numero di cicli impiegati da ciascuna istruzione, ai quali vanno aggiunti i 4 impiegati dalla istruzione 'CALL' per un totale di 30 cicli di clock, pari (con un clock di 16MHz) a 1,875*µs* di tempo impiegato dalla chiamata alla funzione al suo ritorno.

anche una memoria EEPROM che sappiamo avere la caratteristica di mantenere i dati in essa memorizzati anche in mancanza di alimentazione. Avevamo già accennato che questa memoria non è accessibile direttamente dalla CPU (attraverso i bus dati e indirizzi) come invece lo è la RAM, pertanto non possiamo definire una variabile in EEPROM e poi utilizzarla nel modo consueto (assegnare un valore, leggerla, ecc.): per accedervi in lettura o scrittura bisogna infatti utilizzare particolari procedure che agiscono su appositi registri (effettivamente possiamo pensare alla EEPROM come a una periferica, piuttosto che a una memoria).
In nostro aiuto viene la libreria fornita col

pacchetto WinAVR (*avr-libc*), che è fornita anche di alcune funzioni "pronte all'uso" per lavorare con la EEPROM, consentendoci di esulare dai dettagli di programmazione: quelle che ci interessano in questa applicazione sono le seguenti:

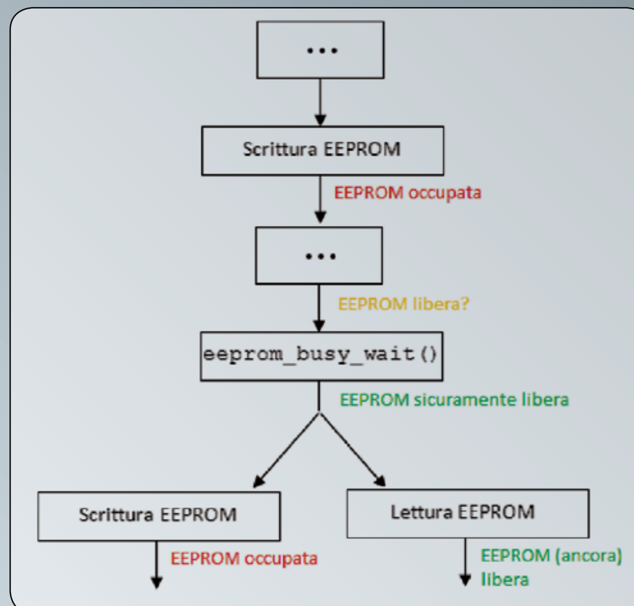
```
- eeprom_write_byte(Indirizzo, Valore); /* scrit-
  tura di un byte nella cella Indirizzo */
- eeprom_read_byte(Indirizzo); /* lettura del
  byte dalla cella Indirizzo */
```

La prima scrive il byte `Valore` nella cella di EEPROM indirizzata dalla variabile `Indirizzo` (che dev'essere definita come puntatore a byte); la seconda invece permette di accedere alla memoria in lettura per recuperare i byte precedentemente scritti.
Nel nostro programma utilizziamo queste funzioni per salvare la variabile `SetPoint`, ed anche il suo complemento (`~SetPoint`); all'accensione del sistema, dopo averle recuperate entrambe, verifichiamo appunto che una sia il complemento dell'altra per ritenere valido il valore.
Quando si leggono i dati da una EEPROM infatti dobbiamo considerare che potrebbero non essere corretti per due motivi:

1. se non si è mai scritto nulla precedentemente (come nel caso di un microcontrollore appena acquistato oppure riprogrammato), la EEPROM è "vergine" e contiene tutti i bit a '1';
2. se vi è stata un'interruzione dell'alimentazione proprio durante la fase di scrittura, nel qual caso non è prevedibile quale sarà il numero effettivamente memorizzato.

Facendo la verifica del setpoint e del suo complemento, il programma può capire se ci si trova in una di tali condizioni (valore comunque sbagliato, per un motivo o per l'altro) ed in tal caso manterrà il valore caricato di *default* (SETPOINT_DEFAULT).
Parlando ancora della EEPROM, per utilizzarla correttamente bisogna ancora osservare che mentre l'operazione di lettura è "istantanea", ogni fase di scrittura impiega un certo tempo per essere eseguita (circa 10ms) durante il quale la memoria non è accessibile né in scrittura, né in lettura: pertanto nel programma dobbiamo verificare il suo stato prima di fare una nuova operazione, se non siamo sicuri che il tempo trascorso dall'ultima scrittura sia stato sufficientemente lungo. Nel nostro programma

Fig. 9



utilizziamo un'altra funzione di libreria, chiamata `eeprom_busy_wait()`, che semplicemente non termina (quindi il programma non prosegue) fin tanto che la EEPROM è "occupata": pertanto conviene chiamarla prima di ogni operazione su EEPROM che segue una scrittura, come schematizzato in **Fig. 9**.
Un'ultima cosa: ricordatevi di includere l'header *<avr/eeprom.h>*, che rende visibili al vostro programma queste ed altre funzioni relative alla gestione della memoria EEPROM che potrebbero venirvi utili nei vostri programmi, delle quali troverete tutta la documentazione sul solito file "avr-libc-user-manual.pdf", nella sezione *EEPROM handling*.

IL PROGRAMMA DEL TERMOSTATO

Nelle **Fig. 10** e **Fig. 11** vengono mostrati i diagrammi di flusso relativi rispettivamente al programma principale e alla isr del T/C1; dal sito di *Elettronica In* potete invece scaricare i due programmi sorgente, chiamati "Termostato_potenziometro.c" (che implementa la soluzione con setpoint analogico) e "Termostato_tasti.c" (con il setpoint digitale).
Partendo dalla descrizione della isr (**Fig. 11**), in essa si interroga il sensore per leggere la temperatura precedentemente acquisita, e quindi si manda il comando per iniziare una nuova conversione, il cui risultato verrà letto alla chiamata successiva della isr. Prima di uscire da essa si mette infine a '1' il flag "LETTURA_SENSORE" (contenuto, insieme ai flag denominati "ERRORE_SONDA" e "STATO_USCITA", nella variabile chiamata `RegistroFlag`) per comunicare al programma principale (nel `main()`) che

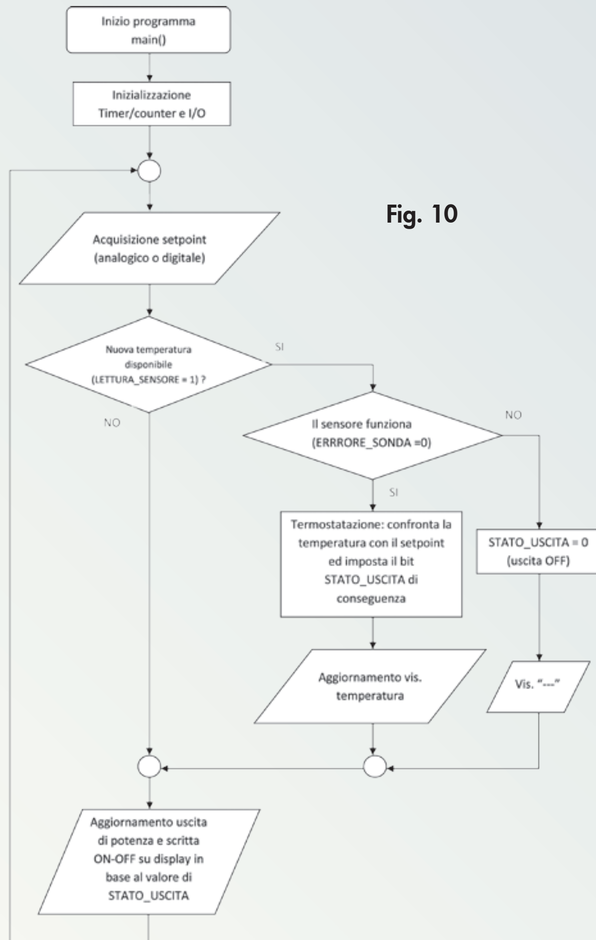


Fig. 10

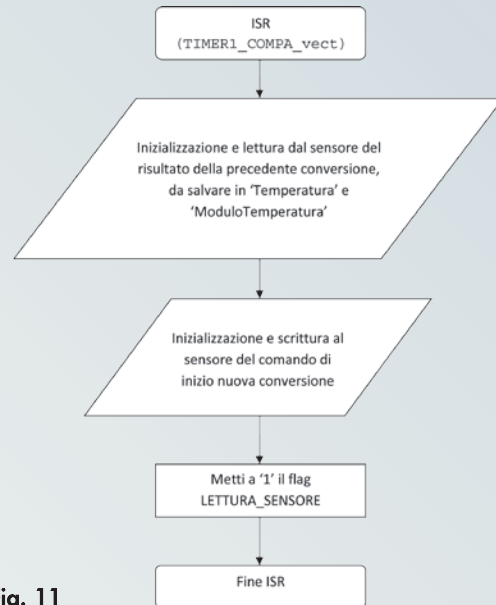


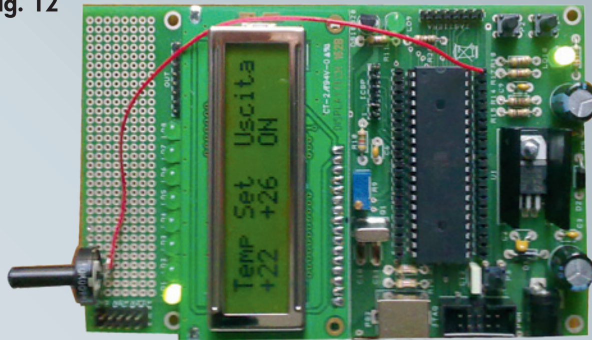
Fig. 11

nelle variabili `Temperatura` e `ModuloTemperatura` è disponibile un nuovo valore da elaborare.
Nel sorgente del programma si può notare che prima di ogni nuova comunicazione col sensore viene effettuata la sequenza d'inizializzazione (implementata nella funzione `DS_Reset()`), nella quale si verifica anche che il sensore risponda con l'impulso di presenza: in caso contrario viene messo a '1' il flag chiamato "ERRORE_SONDA" che permette al programma principale di sapere se la sonda funziona o meno e regolarsi di conseguenza.
Passando al diagramma di flusso del `main()` di **Fig. 10**, si evince che dopo il processo di acquisizione del setpoint (analogico o digitale), viene testato il flag "LETTURA_SENSORE" per sapere se è disponibile un nuovo valore di temperatura: in tal caso si esegue il codice precedentemente discusso (**Listato 1**) per aggiornare il valore che deve assumere l'uscita, contenuto nel flag "STATO_USCITA", e viene inoltre complementato lo stato del pin PD7, producendo il lampeggio del LED LD9 a conferma visiva che il programma sta "funzionando" anche se non si osservano a display variazioni di temperatura.
Il flag "STATO_USCITA" viene successivamente verificato nel brano di codice riportato nel **Listato 2** per attivare o meno l'uscita di potenza 'O0', alla quale come visto nella scorsa puntata possiamo collegare un relè per pilotare il carico che ci interessa (caldaia, condizionatore, ecc). In particolare, utilizziamo la funzione "`ImpostaLineePotenza`()" cui passiamo come argomento 0x01 (uscita attiva) oppure 0x00 (uscita disattiva). Parallelamente andiamo anche a visualizzare lo stato dell'uscita sul display (ON oppure OFF).
In **Fig. 12** potete notare la visualizzazione del display con il programma "Termostato_potenziometro.c" in azione; si noti il potenziometro montato sull'area millefori,

Listato 2

```
/* Impostazione uscita 'O0' di U3 sulla base del flag 'STATO_USCITA' */
        LCD_Command(0xCA);/* Posizionamento cursore display */
        if (RegistroFlag & (1 << STATO_USCITA)) { /* flag 'STATO_USCITA' = 1 -> uscita attiva */
                ImpostaLineePotenza(0x01); /* attivazione uscita 'O0' */
                LCD_DisplayString("ON ", 3); /* aggiornamento visualizzazione display */
        }
        else {  /* flag 'STATO_USCITA' = 0 -> uscita disattiva */
                ImpostaLineePotenza(0x00); /* disattivazione uscita 'O0' */
                LCD_DisplayString("OFF", 3); /* aggiornamento visualizzazione display */
        }
```

Fig. 12



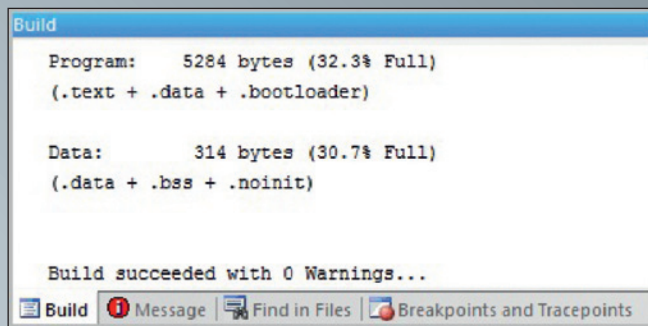


Fig. 13

ed il filo rosso utilizzato per collegare il suo terminale centrale con il piedino 40 (PA0) del microcontrollore. La tensione di +5V e la massa per il potenziometro sono invece prelevati sul vicino connettore.

L'OCCUPAZIONE DI MEMORIA

A proposito del "Termostato_potenziometro.c" vi invitiamo ad osservare, dopo averlo compilato, alcune delle informazioni mostrate nella finestra "Build" di AVR Studio, ed in particolare la voce "Program" (**Fig. 13**).
Essa riporta il numero di bytes complessivi richiesti dal codice e dalle variabili utilizzate, che vale 5284 per il nostro programma; sulla base di questa informazione il compilatore calcola e mostra pure la percentuale di risorse impegnate dal microcontrollore che si intende utilizzare (32,3% per il nostro ATMEGA16).
Anche se potrebbe sembrare strano che per un programma relativamente *semplice* (meno di 400 righe di codice sorgente) come quello proposto vi sia una tale esigenza di memoria, si ricordi che il compilatore aggiunge nella fase di *linking* il codice oggetto delle librerie necessarie ad implementare le funzionalità che richiediamo nel programma. Ad esempio, la divisione che usiamo per ottenere il setpoint dalla lettura della tensione sul potenziometro:

```
SetPoint = ADCH / 8.5;
```

coinvolge un numero decimale (nel divisore: 8,5) e pertanto il compilatore deve fare ricorso a funzioni in virgola mobile per svolgere la divisione, le quali "portano via" parecchio spazio in memoria. A riprova di ciò, ricompilate il programma dopo aver modificato il divisore in un numero intero:

```
SetPoint = ADCH / 8;
```

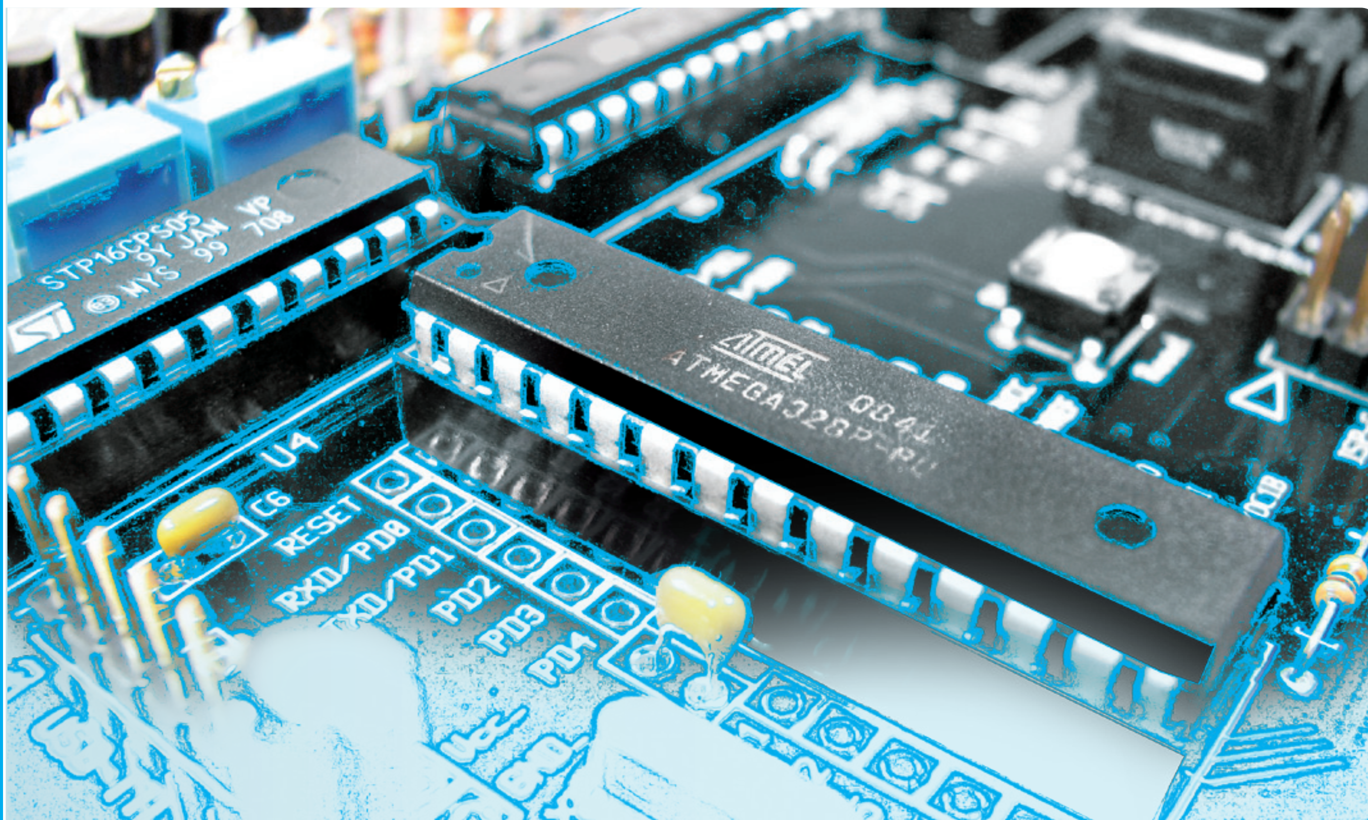
Questa volta vedrete alla voce "Program" della finestra "Build" che la memoria richiesta è scesa a 2122 bytes per un totale di occupazione del 13% nel microcontrollore! Dal lato pratico l'effetto di questa modifica è che il setpoint impostabile può arrivare fino a 31°C (invece di 30, in quanto 255 / 8 = 31,875), ma la memoria utilizzata con questo accorgimento si è praticamente dimezzata, lasciando spazio disponibile per inserire molto più codice nel programma prima di essere costretti a passare a un microcontrollore con maggiori capacità di memoria (come l'ATMEGA32).
Questo discorso è generalizzabile: scrivendo programmi per microcontrollori utilizzando linguaggi ad alto livello (come il C), bisogna considerare che non si hanno a disposizione le "risorse infinite" di memoria di un PC; nel concepimento di un nuovo programma bisognerebbe pertanto cercare di organizzarlo in modo da *evitare*, se possibile, di:

- lavorare con variabili più grandi di 8 bit o in virgola mobile (di tipo *float* e *double*);
- fare ricorso a funzioni di libreria proprie del linguaggio C come `printf()`, scanf(), ecc.

Infine si ricordi che le funzioni di libreria sono… funzioni, e come tali vengono inserite in un solo punto nel codice oggetto del programma indipendentemente da quante volte esse vengano richiamate dal programma stesso.
Questo significa ad esempio che ridurre il numero di operazioni con numeri decimali non è in proporzione efficace (ai fini della riduzione del codice) come l'eliminarle del tutto, in quanto ne basta una perché il compilatore inserisca tutto il codice necessario dalla libreria relativa: altre eventuali operazioni con numeri decimali presenti nel programma utilizzeranno (almeno in parte) lo stesso codice, e quindi la differenza di memoria occupata sarà più modesta.

Riferimenti

[1] http://datasheets.maxim-ic.com/en/ds/DS18B20.pdf



ATMEL OPEN SOURCE

dell'ing. OSVALDO SANDOLETTI

Concludiamo la serie di puntate del corso con la descrizione di come interfacciare la demoboard con un PC dotato di porta USB. Ultima puntata.

L'interfaccia "storicamente" più utilizzata per collegare un dispositivo ad un calcolatore, sia esso un PC, un grosso *mainframe* o un terminale, si basa sulla trasmissione seriale dei dati, utilizzando un protocollo asincrono: i bit della parola da trasmettere vengono trasferiti sequenzialmente lungo un unico filo e ciascuno ha una durata temporale fissa e stabilita, in modo tale che il ricevitore sappia distinguerli anche se hanno tutti lo stesso valore (in tal caso non vi è una variazione di livello che riveli la "fine" di un bit e l'inizio del successivo). Pertanto in linea di principio, se si stabilisce che il *tempo di bit* sia pari a un millisecondo, occorreranno otto millisecondi per trasmettere un intero byte di dato. Questo è indubbiamente uno svantaggio in termini di velocità di trasmissione, bilanciato tuttavia dalla semplicità della connessione tra i dispositivi, composta da un solo filo (oltre ovviamente a quello di "ritorno" di massa) per ogni direzione. Se la comunicazione è bidirezionale, occorreranno quindi due fili: uno di trasmissione (abbreviato in *Tx* o *TxD*) per i dati che escono dal dispositivo, e uno di ricezione (abbreviato in *Rx* o *RxD*) per i dati in ingresso. Naturalmente, effettuando il collegamento tra due dispositivi bisogna aver cura di collegare l'uscita (Tx) del primo all'ingresso (Rx) del secondo e viceversa.

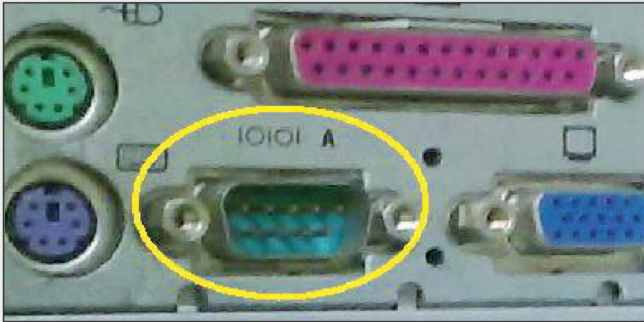


Fig. 1 - Connettore D-SUB dell'interfaccia RS-232.

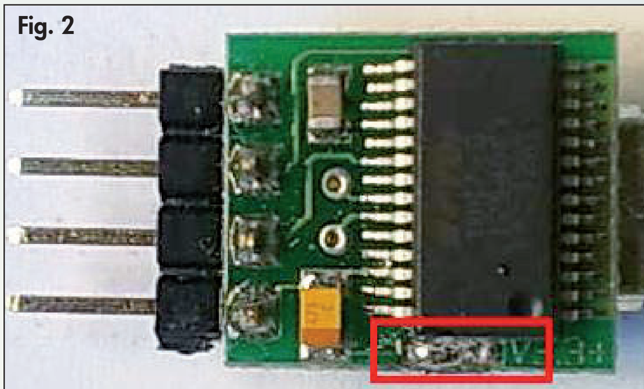


Fig. 2

Nota: per quantificare la velocità di trasmissione si indica il numero di bit che vengono trasmessi in un secondo; tale valore coincide con l'inverso della durata di ogni singolo bit (espressa in secondi), e viene identificato con i termini di bit per secondo (abbreviato in bps), bit-rate, oppure baud-rate (sebbene il baud identifichi più genericamente una grandezza che, in alcune modulazioni di trasmissione, può corrispondere a più bit ed in tal caso il baud-rate non coinciderebbe con il bit-rate).
Per via della sua semplicità, sin dagli anni '60 si è adottato prevalentemente questo metodo per mettere in comunicazione tra loro due dispositivi, standardizzandolo sia a livello elettrico (numero di bit per parola, ampiezza dei segnali e velocità di trasmissione) che meccanico (tipo di connettori e di cavo), sotto il nome di EIA RS-232. (del quale potete trovare approfondimenti su [1]). Alcuni PC di tipo "desktop" possiedono ancora questo tipo di interfaccia (basta verificare la presenza del connettore a vaschetta a nove poli di tipo "D-SUB") ed evidenziato in **Fig. 1**, ormai scomparsa invece dai notebook in quanto soppiantata dalla più recente USB (che pure utilizza un protocollo seriale, ma completamente differente).
Poiché non vogliamo precludere la possibilità di collegare la nostra demoboard ai PC sprovvisti dell'interfaccia seriale, abbiamo previsto l'inserimento del modulo FT782 (acquistabile sul sito *www.futurashop.it*) che fa uso di un integrato della FTDI per convertire i dati USB nel più "tranquillo" protocollo seriale asincrono, facilmente gestibile dal microcontrollore ATMEGA16. Quando entrate in possesso di tale modulo, ricordatevi come prima operazione di unire con una goccia di stagno le apposite piazzole che si trovano sul circuito stampato in basso, sotto l'integrato FTDI (se guardiamo il modulo dal lato dell'integrato e tenendo il connettore USB sulla destra, **Fig. 2**). Con queste piazzole possiamo selezionare i livelli di tensione da utilizzare sulle linee seriali Tx e Rx che vanno al microcontrollore: poiché l'ATMEGA16 viene alimentato a 5V, bisognerà cortocircuitare la piazzola centrale con quella sinistra (verso la strip), e lasciare scollegata la piazzola a destra (verso il connettore USB).
Nella demoboard la strip del modulo va inserita nell'apposita "controparte" a 4 piedini che si trova sotto al display (il quale dev'essere pertanto rimosso). Per mantenere il modulo orizzontale sulla scheda e poter rimontare il display, suggeriamo di saldare sulla demoboard una strip femmina a 90°. Nella **Fig. 3** si nota la collocazione di tale modulo sulla demoboard, con il cavetto USB già inserito; a proposito di quest'ultimo, il connettore B (lato demoboard) deve essere con spina mini-USB.
Il ponticello J1 sulla demoboard consente, se collegato, di prelevare i +5V di alimentazione della demoboard dal bus USB (e quindi dal PC ad esso collegato), piuttosto che attraverso U1; in tal caso, non bisogna applicare alcuna tensione sull'ingresso "PWR" per evitare il rischio di compromettere la circuiteria della demoboard e… del PC ad essa collegato! Salvo esigenze particolari, consigliamo pertanto di non ponticellare mai J1.

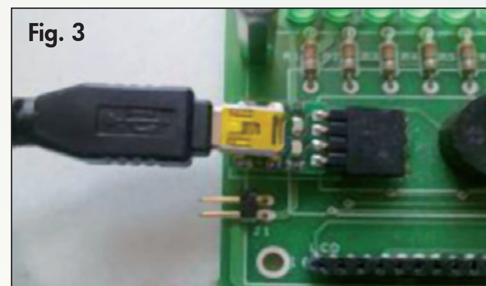


Fig. 3

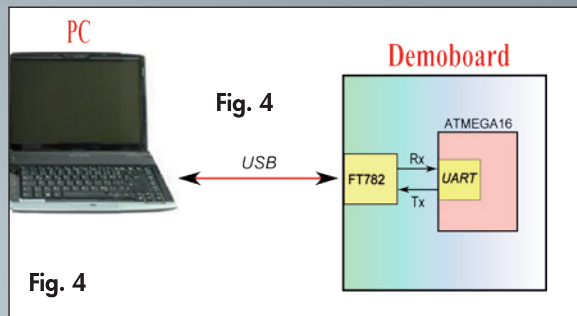


Fig. 4

LA GESTIONE DEL PROTOCOLLO SERIALE

In **Fig. 4** viene mostrato lo schema a blocchi del flusso di dati tra PC e demoboard, nel quale si nota il "traduttore" FT782, a cavallo tra il bus USB e le due linee seriali asincrone Tx e Rx, ciascuna delle quali trasferisce i dati in una direzione (per avere una comunicazione bidirezionale tra PC e microcontrollore). Tali linee sono direttamente collegate al microcontrollore, a due piedini (PD0 e PD1) che possono essere configurati per fare capo ad una periferica interna chiamata UART (*Universal Asynchronous Receiver Transmitter*), che si occupa della conversione dei dati da parallelo a seriale e viceversa: il dato ricevuto serialmente sul piedino Rx (PD0) viene salvato in un registro dal quale la CPU può in seguito prelevarlo; viceversa, un dato parallelo scritto in tale registro dalla CPU viene serializzato e trasferito, un bit alla volta, sul piedino Tx (PD1).
Va detto che in virtù della sua semplicità si potrebbe implementare il protocollo seriale asincrono anche via software, cioè senza l'ausilio di una periferica UART: si tratta in linea di principio di eseguire otto cicli, alla cadenza del tempo di bit voluto, in ognuno dei quali si imposta il valore di un pin di I/O scelto come "Tx" con lo stato del bit corrente; per la ricezione, in ciascuno degli otto cicli si legge lo stato di un piedino scelto come "Rx" e lo si ricopia nel relativo bit di un registro utilizzato come destinazione del dato. Nella pratica le cose sono leggermente più complesse, in quanto nel protocollo asincrono bisogna anche gestire i bit di start (inserito prima dei bit di dato) e di stop (inserito dopo) e, per quanto riguarda la lettura, riuscire a effettuare il campionamento nel modo più preciso possibile al centro di ogni bit in arrivo. Detto questo, noi sfrutteremo la periferica UART del microcontrollore e pertanto non ci preoccuperemo dei problemi appena esposti.

LA PERIFERICA U(S)ART

Il microcontrollore ATMEGA16 è dotato di una periferica chiamata USART, dove la 'S' aggiuntiva sta per "Synchronous" poiché è in grado di gestire anche il protocollo seriale sincrono, caratterizzato dal fatto che la cadenza dei bit sulle linee Tx e Rx è gestita da un segnale di clock esterno (richiede quindi un collegamento aggiuntivo tra le periferiche), piuttosto che essere ricavata automaticamente sulla base del bit-rate impostato come nel protocollo asincrono che utilizzeremo. Va detto che il protocollo sincrono, molto utilizzato per il trasferimento dati tra dispositivi su una stessa scheda (uno per tutti l'SPI, che utilizziamo per la programmazione dei micro, viene adottato ad esempio per comunicare con le memorie *SD-Card*), non è praticamente mai stato scelto per la comunicazione "via cavo" tra sistemi diversi, come un PC e una stampante.
Tornando alla periferica USART del nostro micro, ne troverete come al solito una dettagliata spiegazione sul datasheet nel capitolo omonimo; di seguito discuteremo i passi salienti da effettuare per essere rapidamente in grado di utilizzarla. La fase più importante è sempre l'inizializzazione nella quale, impostando opportunamente i bit dei registri relativi, si stabiliscono:

- il bit-rate della comunicazione – registri UBRRH e UBRRL;
- il tipo di protocollo (sincrono o asincrono) e il formato dei dati – registro UCSRC.

Per quanto riguarda il bit-rate, se vogliamo comunicare con un PC dobbiamo avere cura di scegliere uno dei valori che convenzionalmente (per motivi "storici" legati alle periferiche UART dei primi calcolatori) sono stati previsti per la comunicazione: 2400, 4800, 9600, 19200, 38400, 57600 oppure 115200 bps; essi sono i valori che *sicuramente* il sistema operativo e tutti i programmi di comunicazione permettono di impostare senza problemi.
Una volta stabilita la velocità, datasheet alla mano possiamo trovare immediatamente il numero a 16 bit UBRR (da caricare nei registri a 8 bit UBRRH e UBRRL) grazie a quattro tabelle chiamate "Examples of UBRR Settings for Commonly Used Oscillator Frequencies" nelle quali, data la frequenza di funzionamento del microcontrollore, si cerca la riga con la

Table 19-12. Examples of UBRR Settings for Commonly Used Oscillator Frequencies

Baud Rate (bps)	f_{osc} = 16.0000 MHz			
	U2X = 0		U2X = 1	
	UBRR	Error	UBRR	Error
2400	416	-0.1%	832	0.0%
4800	207	0.2%	416	-0.1%
9600	103	0.2%	207	0.2%
14.4k	68	0.6%	138	-0.1%
19.2k	51	0.2%	103	0.2%
28.8k	34	-0.8%	68	0.6%
38.4k	25	0.2%	51	0.2%
57.6k	16	2.1%	34	-0.8%
76.8k	12	0.2%	25	0.2%
115.2k	8	-3.5%	16	2.1%
230.4k	3	8.5%	8	-3.5%
250k	3	0.0%	7	0.0%
0.5M	1	0.0%	3	0.0%
1M	0	0.0%	1	0.0%
Max [1]	1 Mbps		2 Mbps	

1. UBRR = 0, Error = 0.0%

Fig. 5

velocità di trasmissione desiderata (*baud-rate*) e viene ricavato il numero da scrivere nei registri. Si noti che nelle tabelle sono contemplate tra l'altro tutte le velocità di trasmissione che abbiamo elencato sopra, proprio perché esse rappresentano uno standard *de facto*. Per tutte le altre velocità ottenibili dalla USART del micro e non presenti in tabella, bisogna calcolarsi manualmente il numero da inserire in UBRRH-L con le equazioni mostrate nella tabella "Equations for Calculating Baud Rate Register Setting". Nel fare ciò si consideri che difficilmente si otterrà un numero intero esatto corrispondente alla velocità desiderata; sicuramente non vi sono problemi se la velocità "effettiva" corrispondente al troncamento del numero non differisce più di un 1% dal valore voluto, ma in particolari casi sono *ben tollerate* percentuali maggiori (al proposito si veda il paragrafo "Asynchronous Operational Range" per maggiori dettagli).
Per fare un esempio concreto nel programma "ComunicaPC" di cui parleremo nel prossimo paragrafo abbiamo scelto la velocità di 38400 bps che, con l'oscillatore esterno da 16MHz della demoboard e in base alla tabella che riportiamo in **Fig. 5**, porta ad impostare UBRR = 25 (quindi UBRRH = 0 e UBRRL = 25). Sempre da tabella si nota che con tale scelta vi è un errore dello 0,2%, trascurabile ai fini della correttezza della comunicazione.
Come ultima annotazione a riguardo dell'impostazione della velocità di trasferimento, nel generatore di clock dell'USART vi è un bit chiamato U2X (contenuto nel registro UCSRA) che permette di raddoppiare la velocità di comunicazione gestibile dalla periferica. Sempre dalla figura 5 potete notare infatti che se tale bit vale '1', per avere 38400 bps dovremmo impostare UBRR (che è semplicemente il fattore di divisione della periferica) a 51 anziché a 25; inoltre la velocità massima ottenibile (con UBRR = 0) è di 1 Mbps se U2X = '0' e 2 Mbps se U2X = '1'. Tuttavia, se la velocità prescelta è ottenibile anche con U2X = '0', suggeriamo di lasciare questa impostazione in quanto la USART effettua più campionamenti durante la ricezione di un dato (con U2X = '0' ogni bit viene verificato ben 16 volte anziché 8) e dunque diminuisce la probabilità di errore dovuta anche al bit-rate non perfetto o ad eventuali disturbi.
Per quanto riguarda il formato dei dati (*data frame*) scegliamo il più semplice: 8 bit di dato, nessun bit di parità e un bit di stop: questo significa ogni dato trasmesso dalla USART sul piedino Tx sarà composto dalla sequenza di bit rappresentata in **Fig. 6**; analogamente, ogni dato ricevuto su Rx dovrà avere la stessa struttura per essere considerato valido dalla periferica.
Tale formato è il più classico tra quelli adottati nelle trasmissioni asincrone: la linea (Tx o Rx) si trova normalmente a '1' logico, che rappresenta lo stato "di riposo" (*idle state*). La trasmissione di un nuovo dato inizia con il bit di *start* che è sempre '0': la linea è forzata bassa per la durata esatta di un tempo di bit (l'inverso del bit-rate); seguono gli otto bit che portano il valore del byte effettivo che vogliamo trasmettere (il primo dei quali, D0, è il meno significativo) e un bit finale detto di *stop*, che è sempre a '1' logico. Successivamente la linea torna nello stato di riposo, sino al prossimo dato.
Si ricordi comunque che non dobbiamo assolutamente preoccuparci di aggiungere i bit di start e stop ai dati che vogliamo trasmettere, o di togliere tali bit dai dati ricevuti, in quanto lo fa automaticamente la USART: in un suo apposito registro (chiamato UDR), sia che lo scrivessimo o lo leggessimo, viene contenuto il dato vero e proprio, "ripulito" dai bit supplementari.

Fig. 6

RS232 contro USB

Se nel mondo dei PC la tendenza è chiaramente quella di eliminare totalmente l'interfaccia RS-232, nei dispositivi "*embedded*" (sistemi elettronici medio-piccoli a microcontrollore, come la nostra demoboard) la situazione è diversa e il protocollo seriale asincrono grazie alla sua semplicità viene ancora largamente utilizzato: la comunicazione *minima* è composta da un singolo pacchetto di bit del tipo mostrato in **Fig. 6**, mentre su USB bisogna inviare lunghe sequenze di bit anche per trasmettere un solo byte di informazione effettiva (in questo caso si parla di *overhead* di protocollo).
Ne deriva che il programma necessario alla gestione del protocollo USB avrà una certa complessità e andrà a occupare uno spazio non trascurabile nella memoria del microcontrollore, che come sappiamo è molto più limitata di quella disponibile in un PC; la sua esecuzione impatterà inoltre negativamente sulle prestazioni del processore con ricadute sulla velocità di esecuzione dell'applicazione principale durante i cicli di comunicazione.
Infine, mentre il protocollo seriale asincrono è "anonimo", in quello USB ogni periferica collegata deve inviare tra l'altro un codice di identificazione, composto da due numeri: VID (Vendor Identifier, identifica il costruttore del dispositivo) e PID (Product Identifier, che contraddistingue diversi dispositivi accomunati dallo stesso VID).
Queste informazioni sono necessarie in quanto il protocollo USB è *multipunto* (vi possono essere fino a 127 periferiche collegato sullo stesso bus) e consente il riconoscimento automatico della periferica che si collega in un certo istante (il famoso *plug and play*); per evitare che due costruttori scelgano fortuitamente lo stesso VID per i propri prodotti, esso non può essere liberamente *inventato* ma va richiesto sul sito "USB Implementers Forum" (*www.usb.org*) dietro pagamento di un corrispettivo (dai 2000$ in su, dipendentemente dal fatto che si voglia o meno anche l'autorizzazione ad apporre sui propri prodotti il logo USB). In conclusione, mentre la USB spopola universalmente su PC e memorie di massa, tra i quali si movimentano grosse quantità di dati come musica e video e l'esigenza primaria è la velocità di trasferimento (che con la nuova USB 3.0 dovrebbe teoricamente arrivare a 4,8 Gigabit al secondo), il protocollo seriale asincrono viene ancora adottato sulle piccole piattaforme a microcontrollore, per colloquiare con periferiche di comunicazione nelle quali velocità inferiori al MBit/s risultano più che sufficienti.
Comunque, per dotare di USB anche i piccoli sistemi in cui il microcontrollore sia sprovvisto internamente di tale interfaccia, alcune Case di semiconduttori (tra le quali la più conosciuta è forse FTDI) hanno incentrato il loro *core-business* nello sviluppo di dispositivi convertitori da protocollo USB a seriale asincrono (ma esistono anche convertitori USB-parallelo, USB-IIC, ed altri ancora), consentendo dunque l'interfacciamento tra un PC dotato di porta USB con un sistema a microcontrollore sprovvisto di tale interfaccia.
La schedina FT782 utilizzata nell'articolo, per esempio, fa proprio uso di uno di questi dispositivi: essi gestiscono perfettamente il protocollo USB senza richiedere alcuna informazione aggiuntiva e possiedono anche gli appropriati codici VID e PID da utilizzare per il colloquio sul bus, pertanto l'utilizzatore non deve preoccuparsi di doverli reperire a parte o tantomeno di comprarli. Inoltre, dal lato PC, il dispositivo può essere riconosciuto dal sistema operativo come una porta seriale asincrona RS232 (virtuale): ad esempio Windows la individua come "COMx", Linux come "ttySx"; pertanto per accedervi si possono usare programmi come *Hyperterminal*, oppure utilizzare le API (librerie di funzioni dedicate) di gestione del protocollo seriale che sono già incluse nel sistema operativo stesso (sia Windows, Linux, o altri) nel caso dello sviluppo di nuovi programmi.

DESCRIZIONE DEL PROGRAMMA

Che cosa fa?

Per imparare a sfruttare nel concreto la comunicazione seriale e l'interfaccia FT782 per poter comunicare con un PC, abbiamo pensato di scrivere un programmino chiamato "ComunicaPC", il cui sorgente è come di consueto scaricabile dal sito di *Elettronica In*. Il suo funzionamento è molto semplice: attende che arrivi un comando dal PC, lo esegue e ritorna in attesa del comando successivo. Come formato dei dati abbiamo scelto di comunicare tra PC e demoboard facendo uso dei soli caratteri ASCII stampabili (ovvero lettere e numeri, per renderli più intellegibili), ed abbiamo suddiviso i comandi in due sole categorie: Scrittura (lettera S) e Lettura (lettera L), entrambi seguiti da un altro carattere (*argomento*) che identifichi la periferica sulla demoboard oggetto dell'operazione. In scrittura, da PC dobbiamo fornire oltre al comando e all'argomento un parametro che specifichi il valore che vogliamo scrivere; in lettura è invece la demoboard a restituire il valore richiesto dall'argomento. Nel programma abbiamo previsto a titolo di esempio un solo comando di scrittura (identificato dalla lettera di argomento 'P', che va a pilotare le linee di potenza) e di lettura (argomento 'T', che ritorna al PC il valore di temperatura rilevata dal sensore DS18B20), pertanto le stringhe (sequenze di caratteri ASCII) che possiamo inviare sono: "L T" per leggere la temperatura e "S P xx" per scrivere il valore 'xx' sulle otto linee di potenza della demoboard. Ogni stringa va poi "chiusa" premendo il tasto di

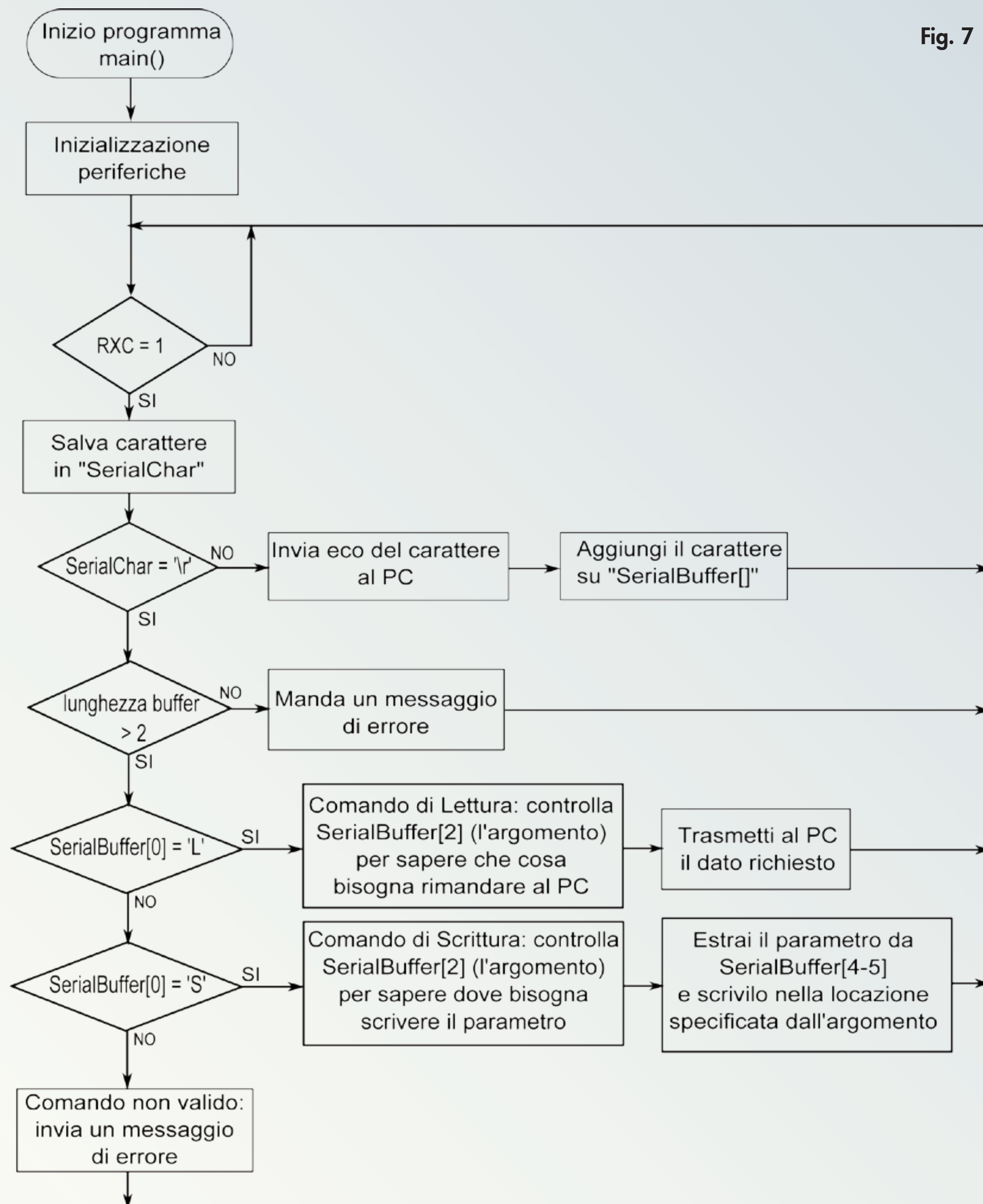


Fig. 7

<Invio>. Da questi esempi sarete naturalmente in grado di aggiungere a vostro piacimento nuovi argomenti di scrittura e lettura per pilotare e/o leggere qualunque altra risorsa della demoboard o del microcontrollore.
Per andare più nel dettaglio, riportiamo in **Fig. 7** il diagramma di flusso del programma principale (il *main()*), che inizia effettuando le consuete inizializzazioni delle periferiche utilizzate (compresa la USART) e poi entra nel *main-loop*. In esso viene effettuata una gestione cosiddetta "*event driven*" dagli informatici: il programma rimane in attesa del verificarsi di un evento per poi rispondere effettuando le operazioni relative. Nel nostro caso, l'evento che aspettiamo è l'arrivo di un carattere trasmesso dal PC che, dopo essere stato convertito tramite il modulo FT782 da USB a seriale asincrono, viene letto dalla USART del micro e depositato nel registro UDR; tale evento viene segnalato dal flag RXC di UCSRA, che noi testiamo continuamente nel main-loop (quindi in *polling*, anche se il flag può essere abilitato a generare anche una richiesta di *interrupt*, ed in tal caso lo gestiremmo con un'apposita *isr*).
Quando rileviamo il flag RXC a '1' possiamo dunque leggere da UDR il nuovo carattere-

re arrivato dal PC e salvarlo nella variabile "SerialChar" per le successive elaborazioni. Assumiamo che ogni carattere in arrivo corrisponda all'equivalente ASCII digitato su PC tramite un programma apposito (di cui parliamo dopo): dato che per leggere la temperatura abbiamo stabilito di trasmettere la stringa "L T <Invio>", da PC arriveranno pertanto in sequenza, a mano a mano che li digitiamo, i caratteri 'L', '*spazio*', 'T' ed infine '*Invio*' (codificato in ASCII con il numero 10 che corrisponde al codice '\r' nel linguaggio C). Nel nostro programma ricomponiamo dunque la stringa salvando i caratteri, in posizione crescente, nel vettore "SerialBuffer[]"; corrispondentemente ritrasmettiamo il carattere al PC in modo che venga visualizzato nella finestra del terminale (di cui parleremo dopo), dando così modo all'utente di verificare la stringa effettivamente immessa.
Nel caso in cui il carattere arrivato sia '\r' significa che è stato premuto il tasto <Invio>: in quel caso si effettua l'analisi della (eventuale) stringa precedentemente ricomposta in "SerialBuffer[]" verificando che:

- "SerialBuffer[]" contenga almeno tre caratteri; il numero di caratteri depositati viene conteggiato dalla variabile "SerialBufferIndex", azzerata ogni volta che si preme <Invio> (per prepararsi al ricevimento di una nuova stringa);
- Il primo carattere della stringa ("SerialBuffer[0]") corrisponda ad un comando di lettura ('L') o scrittura ('S').
  Se queste condizioni non sono soddisfatte il micro invia al PC una stringa di caratteri corrispondente ad un apposito messaggio di errore, in caso contrario viene eseguita l'elaborazione richiesta dal comando rilevato (scrittura o lettura).

DENTRO AL CODICE!

Siccome le funzionalità che abbiamo previsto in questa applicazione sono la gestione delle linee di potenza e la lettura del sensore di temperatura, abbiamo riportato nel programma i relativi brani di codice ovvero le funzioni di controllo del latch 74HC573, del display LCD (benché non indispensabile in questa applicazione, lo inizializziamo e vi scriviamo una stringa che identifichi il programma in funzione) e del sensore DS18B20 (nella isr "TIMER1_COMPA_vect"). Dato che queste funzioni erano già state adeguatamente sviscerate nelle precedenti puntate, analizzeremo adesso il codice relativo alla gestione della USART del microcontrollore. La prima funzione che incontriamo, "Usart_Init()", è relativa all'inizializzazione della periferica: in essa si scrivono i registri di configurazione per impostare la modalità di funzionamento asincrona con 8 bit di dato ed uno di stop (registro UCSRC), ed il bit-rate (registri UBRRH-L). Inoltre bisogna "accendere" le circuiterie di ricezione (per poter leggere i dati dal piedino Rx) e di trasmissione (per poter trasmettere i dati sul piedino Tx) settando i bit RXEN e TXEN del registro UCSRB. Dal momento in cui si attiva RXEN, la USART si mette in attesa di ricevere un 'pacchetto' sul piedino PD0/RXD che abbia il formato descritto nella **Fig. 6**; quando questo accade, come già visto, il flag RXC viene alzato ed il carattere ricevuto è disponibile nel registro UDR. La successiva lettura di UDR è importante in quanto serve ad azzerare il flag RXC, e dev'essere eseguita prima che dalla seriale arrivi un successivo carattere. In caso contrario vi è *concettualmente* il rischio che UDR venga sovrascritto, perdendo il dato precedente. Nel caso dell'USART dell'ATMEGA16 il "buffer di ricezione" (può essere visto come una pila in cui si accatastano i dati in arrivo) è composto da due byte, pertanto si potrebbe ragionevolmente anche attendere la ricezione di due dati prima di effettuare la lettura di UDR; questo però potrebbe non essere vero con un altro microcontrollore. Ad ogni modo la condizione di sovrascrittura viene segnalata dal flag DOR (Data OverRun) del registro UCSRA: se si sospetta che il proprio programma possa non riuscire tempestivamente a leggere UDR in seguito all'arrivo di un carattere varrebbe la pena di testarlo per prevedere una condizione di errore (scartando quindi il dato ricevuto).
La trasmissione di un dato su Tx (con TXEN posto a '1') avviene semplicemente scrivendolo su UDR; l'unica accortezza è quella di verificare che esso sia "vuoto", ovvero che il dato precedentemente scritto sia già stato inviato al registro a scorrimento che lo serializza e lo manda sul piedino Tx, in caso contrario bisogna attendere che ciò avvenga per non sovrascriverlo. Allo scopo vi è il flag UDRE, sempre del registro UCSRA, che quando è

Attenzione ai livelli elettrici delle interfacce!

I livelli elettrici dei segnali sui piedini facenti capo all'UART (PD0 e PD1) sono gli stessi dichiarati per gli altri I/O del microcontrollore: se questi viene alimentato a 5V, su PD0 e PD1 verranno riconosciuti come livelli logici '0' e '1' le tensioni di 0V e 5V (o meglio i valori $V_{(I/O)L}$ e $V_{(I/O)H}$ dettagliati nel datasheet tra le caratteristiche elettriche del dispositivo).
Il protocollo RS232 utilizzato dalle interfacce seriali dei PC (quando presenti) prevede, invece, dei livelli completamente diversi: tipicamente -12V per il livello '1' e +12V per il livello '0'. Tali tensioni sono state scelte fondamentalmente per migliorare l'immunità ai disturbi ed aumentare la lunghezza massima raggiungibile (i dispositivi collegati potrebbero trovarsi, in base alle specifiche, fino ad alcuni metri di distanza tra loro), ma sono completamente incompatibili con qualunque microcontrollore. Per tale motivo è necessario frapporre un *convertitore* per adattare il livello dei segnali: da RS232 (±12V) ai livelli del microcontrollore per la linea dati in ingresso (Rx), e viceversa per i dati in uscita (linea Tx). Naturalmente esistono diverse soluzioni circuitali per realizzare un convertitore di livello, ma la strada normalmente scelta (a ragione) dai progettisti prevede l'adozione di circuiti integrati dedicati allo scopo, dei quali i più famosi sono quelli appartenenti alla famiglia "MAX232" della Maxim (www.maxim-ic.com).
I livelli elettrici dei segnali Tx e Rx dei convertitori USB - seriale (come il "nostro" FT782) sono invece generalmente già compatibili con il mondo digitale (sull'FT782 si può addirittura scegliere tramite il ponticello se lavorare con livelli di 3,3 o 5V, compatibilmente all'alimentazione del microcontrollore a cui viene collegato). D'altra parte questi dispositivi sono pensati per essere collocati vicino al microcontrollore o comunque montati sulla stessa scheda (il collegamento a "filo volante" con il PC è realizzato tramite USB), dunque i segnali Tx e Rx non sono in teoria soggetti a disturbi che giustifichino l'adozione di livelli di tensione maggiori.

a '1' segnala che il registro UDR è vuoto e dunque pronto a ricevere un nuovo dato da trasmettere. Siccome nel programma vi sono diversi punti in cui dobbiamo trasmettere dati al PC abbiamo scritto due funzioni, i cui prototipi sono i seguenti:

```
void SerialPrintChar(uint8_t datoSeriale);
void SerialPrintString(char Str[]);
```

La prima funzione serve a trasmettere il singolo byte "datoSeriale", la seconda manda invece una serie (la "stringa") di caratteri contenuti nel vettore "Str[]", sino a che non incontra il carattere di terminazione '\0': infatti per ricavare il numero di caratteri da trasmettere abbiamo utilizzato la funzione "`strlen()`", definita dall'header "string.h" della libreria standard del C per la quale (e come in tutte le funzioni di manipolazione stringhe del linguaggio C) il terminatore è rappresentato appunto da '\0'. Per creare allora stringhe di dati *compatibili* con questa funzione si possono utilizzare, come visibile nel programma, funzioni come `sprintf()`, oppure chiamare direttamente la funzione con l'argomento esplicitato tra doppie virgolette; ad esempio per mandare al PC la stringa "Ciao!" basta scrivere il seguente codice:

```
SerialPrintString("Ciao!");
```

in quanto la stringa passata in questo modo viene automaticamente completata dal compilatore con il carattere '\0'.
Nota: per compatibilità con il linguaggio C, anche i compilatori per microcontrollori hanno la possibilità di inserire funzioni di librerie "standard" definite dagli header *stdio.h*, *stdlib.h* e così via. Tuttavia, per contenere lo spazio di memoria richiesto dal codice prodotto, esse sono spesso "ridotte" rispetto alle caratteristiche stabilite dalle specifiche ANSI C. Questo significa ad esempio che le funzioni di formattazione stringa (come `printf()`, `scanf()`, ecc.) potrebbero non trattare il tipo esadecimale (%x) o il carattere (%c), le funzioni di libreria matematica potrebbero usare solo numeri interi, e così via. Si tenga infine presente che, se la libreria non contiene una certa funzionalità come ad esempio la formattazione in esadecimale, il compilatore potrebbe *non dare errore* se inseriamo comunque nella funzione `sprintf()` la direttiva %x (in quanto sintatticamente è prevista dal linguaggio C), ma ci si accorgerà del problema solo eseguendo il programma in quanto le variabili formattate in esadecimale non saranno incluse nella stringa prodotta da `sprintf()`. Pertanto, per sapere esattamente cosa può e non può fare ciascuna di queste funzioni è necessario consultare la documentazione del compilatore utilizzato e *non affidarsi* ad un generico manuale di C utilizzato magari per scrivere i primi programmi su PC.

PREPARAZIONE DEL PC

Il driver USB

Analogamente a quanto avviene per ogni

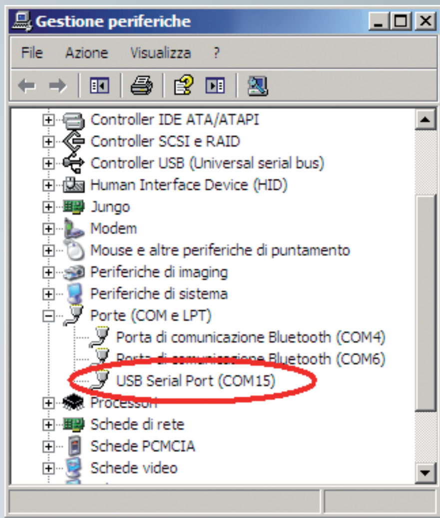


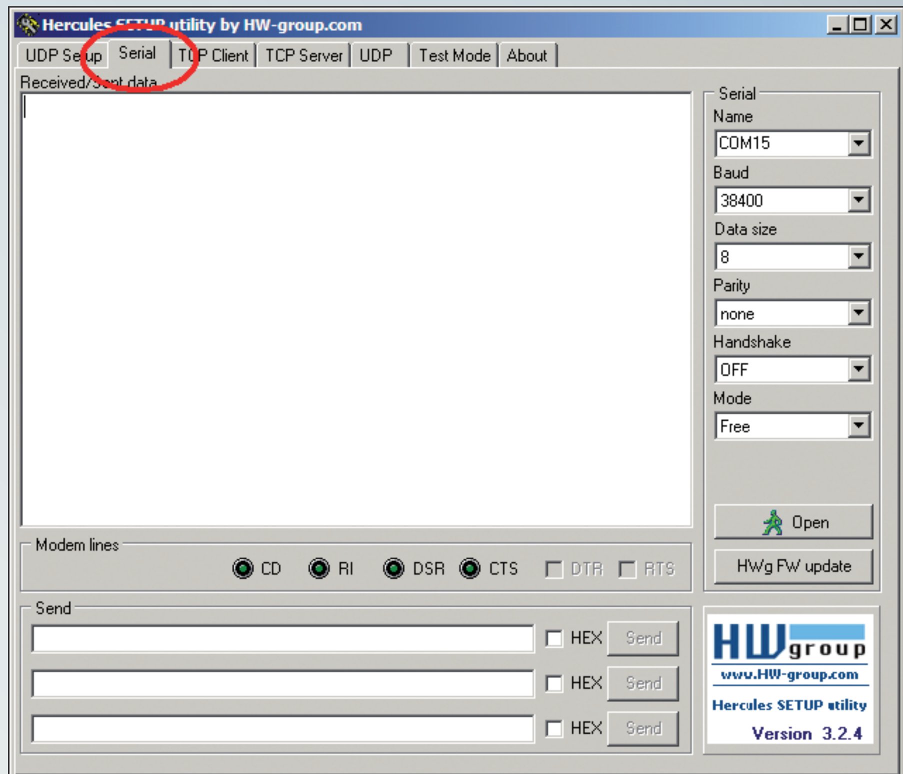
Fig. 8

periferica USB, quando colleghiamo per la prima volta al PC l'interfaccia FT782, il sistema operativo informerà della presenza di un nuovo hardware, per il quale viene richiesto un apposito modulo software (il *driver*) per poterlo gestire. Siccome questa interfaccia fa uso di un convertitore USB/seriale della FTDI, è sul sito di tale Casa che va reperito il driver opportuno: consigliamo di andare direttamente alla pagina [2] e di scaricare, se usate un PC con sistema operativo Windows, il file contraddistinto dal collegamento "Setup Executable", nella colonna "Comments" della tabella. Dopo averlo scaricato, l'esecuzione di questo file provvederà ad installare automaticamente nel sistema operativo i driver necessari al funzionamento della nostra interfaccia.
Nota: prima di effettuare questa operazione provate comunque a collegare l'interfaccia FT782 al PC: se anche alla prima inserzione il sistema operativo non effettua alcuna segnalazione, significa che i driver del dispositivo sono già installati: tenete presente che i convertitori della FTDI sono adottati da molti dispositivi commerciali e quindi, magari inconsapevolmente, li avevate già caricati durante l'installazione del programma di qualche altra periferica.
In ogni caso, una volta che il driver è presente e funzionante, quando collegate il cavo dell'interfaccia FT782 al PC, nell'elenco delle periferiche (in Windows XP sotto "Start → Impostazioni → Pannello di controllo → Sistema" e cliccando sulla voce "Gestione periferiche" della sezione "Hardware"), sotto il gruppo "Porte (COM e LPT)" dovrà apparire una voce del tipo "USB serial port", a cui è stato assegnato un numero di porta seriale tipo COM15 come visibile in **Fig. 8.** Pertanto il sistema operativo vedrà l'interfaccia come una porta di comunicazione seriale vera e propria, e come tale andrà trattata da qualunque programma venga utilizzato per gestirla: dal punto di vista della programmazione il bus USB, ovvero il reale collegamento fisico tra il PC e l'interfaccia FT782, viene completamente "nascosto".

IL PROGRAMMA DI COMUNICAZIONE

Dato che i comandi che impartiamo alla demoboard sono inviati dal nostro PC, anche in quest'ultimo deve "girare" un applicativo

Fig. 9



apposito che ci consenta di farlo: la scelta di utilizzare per la comunicazione i soli caratteri ASCII ci consente di evitare di dovere creare un programma *ad hoc*, ed utilizzarne invece uno di quelli già esistenti in grado di effettuare comunicazioni su linee seriali: in Rete se ne trovano di diversi e per tutti i sistemi operativi, e talvolta nel loro nome si richiama il concetto di *terminale* (inteso in questi contesti come uno dei dispositivi con cui si comunica) o di *comunicazione*: per Windows esistono ad esempio i programmi "Hyperterminal" e "TeraTerm", mentre nel mondo Linux viene spesso utilizzato il "minicom" o la sua versione grafica "cutecom".
Per sviluppare la nostra applicazione noi abbiamo scelto di utilizzare il programma "Hercules Setup Utility", che si può liberamente scaricare dal sito [3] (sul collegamento *Download*), in quanto molto semplice e leggero poiché occupa 1,3MB di spazio su disco e non dev'essere neppure installato dal sistema operativo: il file che si scarica è già l'eseguibile pronto all'uso. Questo programma pur nella sua semplicità consente di gestire diversi tipi di comunicazione anche su reti ethernet con diversi protocolli Internet, ma noi lo useremo unicamente come *terminale seriale*, selezionando appunto la linguetta "Serial" nella finestra del programma come visibile in **Fig. 9**; anche le impostazioni della porta (sul lato destro della finestra) vanno immesse come in figura: 38400 Baud, 8 bit di dato, nessuna parità e nessun *handshake* (che serve per implementare un particolare tipo di protocollo che utilizza altri segnali della porta seriale, oltre a Rx e Tx).
Per quanto riguarda il nome della porta seriale (prima tendina in alto), abbiamo selezionato COM15 poiché, quando colleghiamo la demoboard al PC, Windows ce la enumerava così (**Fig. 8**); questo non è tuttavia garantito: se la COM15 è già attivata oppure si cambia la porta USB su cui ci colleghiamo può cambiare anche il numero assegnatole. Pertanto, conviene sempre verificarlo andando a consultare la "Gestione periferiche"; se poi nel PC fossero collegati più convertitori USB/seriale, potrebbero esservi altrettante voci "USB serial port", ma per sapere qual è il *nostro* basta scollegare un attimo il suo cavo USB: la voce relativa sparirà dall'elenco, per poi riapparire al successivo ricollegamento.
Infine, cliccando sul tasto "Open" si avvia la comunicazione con la porta seriale (nel nostro caso *virtuale* in quanto è rappresentata dall'interfaccia FT782 collegata tramite USB, ma potrebbe anche essere una porta RS232 realmente presente nel PC): nella finestra centrale ("Received/Sent data", che noi chiameremo più brevemente *console*), deve comparire la scritta "Serial port COM15 opened" ed a questo punto il programma è pronto a leggere e trasmettere i dati.

RICEZIONE

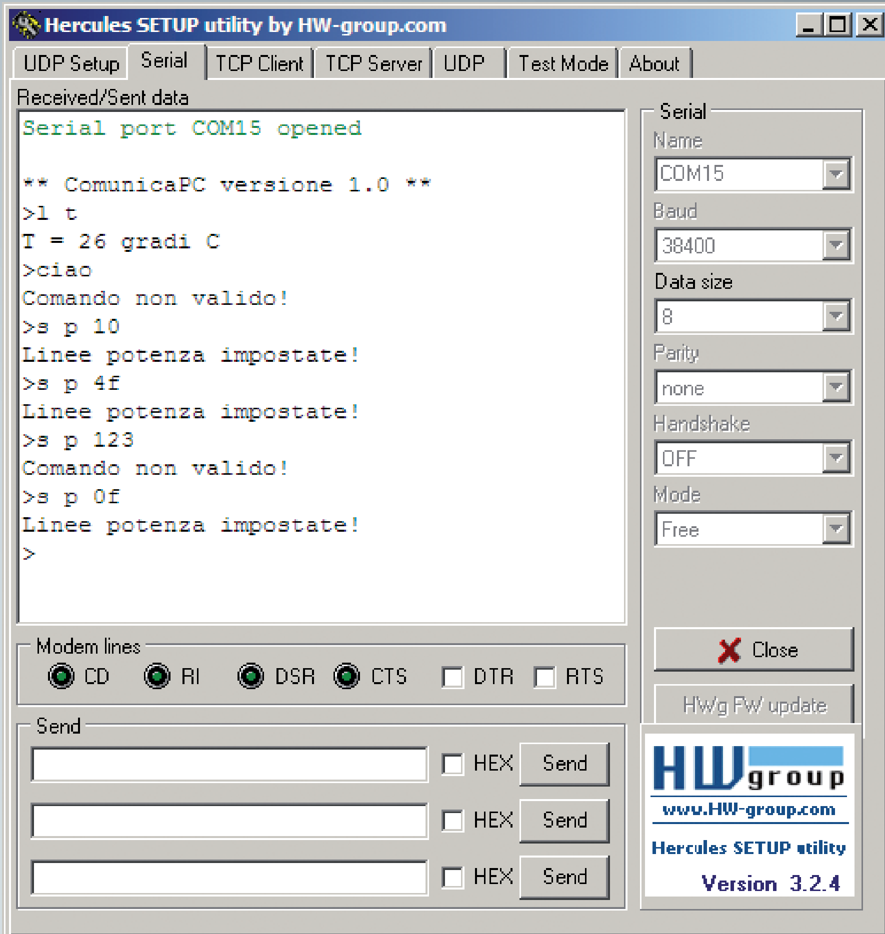
I caratteri ricevuti saranno elaborati dal programma "Hercules" (ma lo stesso discorso è valido anche per gli altri programmi di comunicazione come ad esempio "Hyperterminal") nel seguente modo:

- se sono *caratteri stampabili* saranno visualizzati (in codifica ASCII) sulla console, nella posizione corrente del cursore: ad esempio, se il micro trasmettesse al PC il byte 0x61, verrà visualizzato il carattere 'a' (ed il cursore avanza di una posizione);
- se il byte rientra invece nella categoria dei caratteri *non stampabili*, dipendentemente dal suo esatto valore potrà essere ignorato oppure interpretato come un *comando* (sempre sulla base della codifica ASCII): ad esempio il valore 0x13 (comando di *Carriage return*), riposiziona il cursore della console sul primo carattere della riga attuale; il valore 0x10 (*line feed*) sposta il cursore alla riga successiva (ma non cambia la colonna in cui si trova), e così via.

TRASMISSIONE

Per trasmettere da PC alla demoboard è sufficiente avere il *focus* nella *console* (eventualmente cliccandovi dentro) del programma, e premere un tasto qualunque: il byte corrispondente alla sua codifica ASCII verrà immediatamente inviato, tramite l'interfaccia FT782, alla linea Rx del microcontrollore. Tale carattere non sarà visualizzato nella console che, come abbiamo visto, riporta solamente i caratteri *in arrivo* dalla seriale. Per visualizzare anche i caratteri corrispondenti ai tasti pigiati (come succede normalmente quando scriviamo qualcosa in un editor di testo) è necessario che il microcontrollore, dopo averli ricevuti (sulla linea Rx), li rispedisca subito al PC

Fig. 10



(sulla linea Tx) facendo quindi una sorta di eco. Nel programma del microcontrollore “ComunicaPC”, come si nota nel diagramma di flusso di **Fig. 7**, è infatti quello che facciamo ogni volta che riceviamo un nuovo carattere, a meno che non sia stato pigiato “Invio” (il carattere ‘\r’).
Detto questo, mettiamo in atto la comunicazione: colleghiamo, tramite l’interfaccia FT782, la demoboard al PC con il cavetto USB; in seguito, ad ogni successiva accensione della demoboard stessa dovrà apparire sulla console di Hercules il messaggio “ComunicaPC versione 1.0”, e sulla riga successiva verrà visualizzata un freccetta a rappresentare che il programma del microcontrollore è in attesa di un nuovo comando, che come abbiamo visto può essere solo la richiesta di lettura della temperatura o l’invio di un dato sulle linee di potenza, salvo che non decidiate di aggiungerne altri a piacimento.
In **Fig. 10** è mostrata la console di Hercules dopo aver inviato alla demoboard alcuni comandi; la riga successiva a ciascuno mostra la risposta al comando trasmessa dalla demoboard. Volontariamente abbiamo inserito come esempio anche alcuni comandi sbagliati. Si noti che il parametro di scrittura delle otto linee di potenza viene espresso con un numero esadecimale di due caratteri; nel programma “ComunicaPC” la traduzione da questo dato (che è a tutti gli effetti una stringa di caratteri) nel corrispondente valore numerico decimale viene svolta dalla funzione `hex_2_int()`.
Abbiamo scelto questa notazione in quanto il valore degli otto bit è più intuitivo rispetto a quello di un numero decimale. Naturalmente, ad ogni nuova impostazione delle linee di potenza dovrete vedere i led LD<1:9> accendersi coerentemente con il valore immesso.

IN CONCLUSIONE

In questa puntata abbiamo mostrato che cos’è e come gestire una comunicazione seriale asincrona con il microcontrollore ATMEGA16, e soprattutto abbiamo utilizzato queste nozioni per mettere in comunicazione un qualsiasi PC con la nostra demoboard, che con il semplice programma “ComunicaPC” ci permette di andare a leggerne il valore di temperatura o impostarne le linee di potenza. Questa vuole solo essere una base di partenza, in quanto si può modificare ed espandere il programma a piacimento (nei limiti della memoria del microcontrollore ovviamente!) per trasformare la scheda in una interfaccia multiuso in grado di leggere sensori e pilotare automatismi di ogni genere, in base alle proprie esigenze e desideri, pilotandola dal PC.

Riferimenti

[1] http://it.wikipedia.org/wiki/EIA_RS-232
[2] http://www.ftdichip.com/Drivers/VCP.htm
[3] http://www.hw-group.com/products/hercules/index_en.html