Sistemi di ingresso/uscita

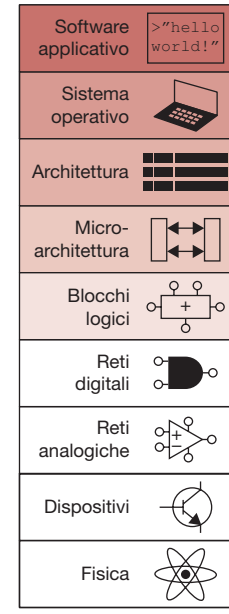
Capitolo **e9**

9.1 Introduzione
9.2 I/O mappato in memoria
9.3 I/O nei sistemi embedded
9.4 Altre periferiche di microcontrollori
9.5 Interfacce a bus
9.6 Sistemi di I/O del PC
9.7 Riassunto

9.1 ■ INTRODUZIONE

I sistemi di Ingresso/Uscita (abbreviati con I/O, dall'inglese *Input/Output*) sono usati per collegare il calcolatore a dispositivi esterni chiamati periferiche. Nei Personal Computer, dispositivi di I/O tipici sono tastiere, video, stampanti e reti *wireless* (senza fili). Nei sistemi *embedded* (cioè nei calcolatori usati per il controllo di macchine di vario tipo) i dispositivi di I/O possono essere la resistenza di un tostapane, il sintetizzatore vocale di una bambola, l'iniettore di carburante in un motore, i motori di posizionamento del pannello solare di un satellite, e così via. Il processore accede ai dispositivi di I/O usando i bus indirizzi e dati, nello stesso modo in cui accede alla memoria.

Questo capitolo presenta esempi concreti di dispositivi di I/O. Il paragrafo 9.2 mostra i principi base per l'interfacciamento di un dispositivo di I/O al processore e per l'accesso a tale dispositivo da parte di un programma. Il paragrafo 9.3 esamina l'I/O nel contesto dei sistemi embedded, mostrando come usare il calcolatore a singola scheda Raspberry Pi (basato su processore ARM) per accedere alle periferiche presenti sulla scheda, compresi gli I/O di uso generale, seriali e analogici, oltre ai timer (temporizzatori). Il paragrafo 9.4 fornisce esempi di interfacciamento con altri dispositivi di largo uso come display a cristalli liquidi, monitor VGA, connessioni radio Bluetooth, motori. Il paragrafo 9.5 descrive le interfacce a bus e illustra il diffuso bus AHB-Lite. Il paragrafo 9.6 presenta una panoramica dei principali sistemi di I/O usati nei PC.



9.2 ■ I/O MAPPATO IN MEMORIA

Nel paragrafo 6.5.1 si è visto che una parte dello spazio di indirizzamento è riservato ai dispositivi di I/O invece che alla memoria. Per esempio, se gli indirizzi fisici nell'intervallo da 0x20000000 a 0x20FFFFFF sono riservati all'I/O, a ogni dispositivo di I/O vengono assegnati uno o più indirizzi di memoria in tale intervallo. Un'istruzione di scrittura a questi indirizzi invia dati al dispositivo, mentre un'istruzione di lettura riceve dati dal dispositivo. Questo

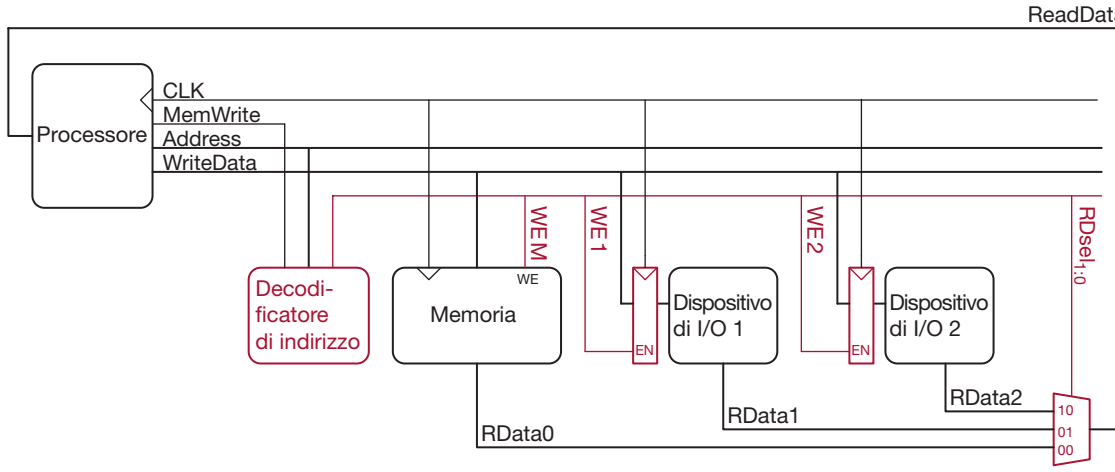


Figura e9.1 Supporto circuitale per I/O mappato in memoria.

metodo di comunicazione con i dispositivi di I/O è denominato **I/O mappato in memoria** (*memory-mapped I/O*).

In un sistema con I/O mappato in memoria, un'istruzione di lettura o scrittura può dunque accedere a memoria oppure a dispositivi di I/O. La **Figura e9.1** mostra la struttura circuitale necessaria per gestire due dispositivi di I/O mappati in memoria. Un decodificatore di indirizzo determina quale dispositivo deve comunicare con il processore: il decoder usa i segnali *Address* e *MemWrite* per generare i segnali di controllo per il resto del circuito. Il multiplexer *ReadData* effettua la selezione tra la memoria e i vari dispositivi di I/O. Registri con abilitazione alla scrittura memorizzano i valori inviati ai dispositivi di I/O.

ESEMPIO e9.1

Comunicazione con i dispositivi di I/O. Si supponga che al dispositivo di I/O numero 1 della Figura e9.1 sia assegnato l'indirizzo di memoria 0x20001000. Scrivere il codice assembly di ARM necessario per inviare il valore 7 a tale dispositivo di I/O e per leggere dal medesimo dispositivo un dato.

Soluzione Il seguente codice assembly scrive il valore 7 nel dispositivo di I/O numero 1.

```
        MOV     R1, #7
        LDR     R2, = indio
        STR     R1, [R2]
indio   DCD     0x2000100
```

Il decoder di indirizzo attiva *WE1* perché l'indirizzo è 0x20001000 e *MemWrite* è VERO. Il valore presente sul bus *WriteData*, cioè 7, viene scritto nel registro collegato ai piedini di ingresso del dispositivo 1.

Per leggere dal dispositivo di I/O numero 1, il processore esegue il seguente codice assembly.

```
        LDR     R1, [R2]
```

Il decoder di indirizzo forza $RDsel_{1:0}$ al valore 01, perché rileva l'indirizzo 0x20001000 e *MemWrite* è FALSO. Il valore emesso dal dispositivo di I/O numero 1 viene passato dal multiplexer sul bus *ReadData* e caricato nel registro R1 del processore.

Gli indirizzi associati ai dispositivi di I/O vengono spesso denominati per brevità **registri di I/O**, perché corrispondono a registri fisici presenti nel dispositivo di I/O, come quelli mostrati nella Figura e9.1.

Ci si riferisce al software che comunica con un dispositivo di I/O con il termine inglese ***driver* del dispositivo**: quando si collega una stampante o un'altra periferica al PC si deve quasi sempre scaricare e installare il driver corrispondente. Scrivere un driver di dispositivo richiede una conoscenza dettagliata della struttura circuitale di tale dispositivo, inclusi gli indirizzi e il comportamento dell'I/O mappato in memoria. Gli altri programmi fanno chiamate alle funzioni rese disponibili dal driver per accedere al dispositivo senza doverne conoscere la struttura circuitale in dettaglio.

Alcune architetture, e in particolare la famiglia x86, usano istruzioni speciali per comunicare con i dispositivi di I/O, invece di adottare l'I/O mappato in memoria. Tali istruzioni hanno il seguente aspetto, dove `disp1` e `disp2` sono gli identificativi univoci del dispositivo di I/O:

```
LDRIO   R1, disp1
STRIO   R2, disp2
```

Questo tipo di comunicazione con i dispositivi di I/O viene denominato *I/O programmato*.

9.3 ■ I/O NEI SISTEMI EMBEDDED

I sistemi embedded usano un processore per controllare le interazioni con l'ambiente fisico circostante. Sono solitamente realizzati partendo da uno o più microcontrollori (MCU, *Micro Controller Unit*), ovvero microprocessori combinati con un insieme di periferiche pronte all'uso, come piedini di I/O digitali e analogici, porte seriali, timer ecc. I microcontrollori sono generalmente molto economici e progettati per minimizzare costo e dimensioni del sistema integrando la maggior parte delle componenti necessarie in un singolo chip. Sono spesso più piccoli e leggeri di una monetina, consumano milliwatt di potenza e costano da qualche decina di centesimi a qualche dollaro. Sono classificati in base alla dimensione dei dati su cui operano: i microcontrollori a 8 bit sono i più piccoli ed economici, quelli a 32 bit offrono più memoria e prestazioni superiori.

Per ragionare in concreto, questo paragrafo illustra l'I/O dei sistemi embedded usando un caso reale. Si focalizza infatti sulla diffusa ed economica scheda Raspberry Pi, che contiene il sistema a singolo chip BCM 2835 della Broadcom con un processore a 32 bit ARM 1176JZ-F a 700 MHz basato sul set di istruzioni ARMv6. Gli aspetti di principio trattati nelle varie parti di questo paragrafo sono seguiti da esempi specifici che possono essere eseguiti sul Pi. Tutti gli esempi sono stati collaudati dagli autori nel 2014 su un Pi dotato del sistema operativo Linux NOOBS Raspbian.

La **Figura e9.2** mostra una fotografia della scheda Raspberry Pi modello B+, che è un calcolatore Linux completo grande come una carta di credito e del costo di circa 35 dollari. Il Pi consuma fino a 1 A da un alimentatore USB a 5 V. Ha 512 MB di RAM e un connettore SD per una scheda di memoria contenente il sistema operativo e i file utente. I connettori forniscono porte audio e video, porte USB per mouse e tastiera, una porta Ethernet (rete locale) e 40 contatti di I/O per usi generali (GPIO, *General Purpose I/O*) che costituiscono l'argomento principale di questo capitolo.

Anche se il sistema a singolo chip BCM2835 ha potenzialità molto superiori a quelle di un tipico microcontrollore economico, l'I/O per usi generali è molto simile. Si inizia quindi a descrivere il BCM2835 del Raspberry Pi e il driver di dispositivi per l'I/O mappato in memoria. Il resto del capitolo illustra come i sistemi embedded eseguono operazioni di I/O digitali, analogiche e seriali. Anche i timer sono spesso usati per generare o misurare precisi intervalli di tempo.

Nel 2014 sono stati venduti microprocessori per un valore di circa 19 miliardi di dollari, e le previsioni per il 2020 parlano di 27 miliardi. Il costo medio di un microcontrollore è inferiore a 1 dollaro, e un microcontrollore a 8 bit può essere integrato in un sistema a chip singolo per meno di un centesimo di dollaro. I microcontrollori sono ormai onnipresenti e praticamente invisibili: si stima che ce ne fossero 150 in ogni abitazione e 50 in ogni automobile già nel 2010. Il componente 8051 è un classico microcontrollore, sviluppato da Intel nel 1980 e oggi venduto da diversi produttori. I componenti Microchip delle serie PIC16 e PIC18 sono i leader di mercato nella fascia 8 bit. La serie di microcontrollori AVR della Atmel è diventata molto popolare tra gli hobbisti come processore della piattaforma Arduino. Tra i microcontrollori a 32 bit, Renesas è il leader di mercato. Freescale, Samsung, Texas Instruments e Infineon sono gli altri principali produttori di microcontrollori. I processori ARM si trovano in quasi tutti gli smartphone e i tablet moderni, e sono usualmente parte di sistemi a singolo chip contenenti il processore multi-core, un'unità di elaborazione grafica e una notevole quantità di I/O.

9.3.1 Il sistema a singolo chip BCM2835

Il BCM2835 è un potente sebbene economico chip progettato dalla Broadcom per dispositivi mobili e altre applicazioni multimediali. Tale sistema a singolo chip contiene un processore ARM definito il **processore applicativo**, un processore VideoCore per la gestione di grafica, video e telecamere, e molte periferiche di I/O. Il contenitore è plastico a matrice di palline (BGA, *Ball Grid Array*) con piccole palline a saldare sulla faccia inferiore: è ideale per saldatura robotizzata, con allineamento preciso del contenitore ai punti di saldatura e riscaldamento. Broadcom non fornisce una scheda dati (*data sheet*) completa, ma una versione

Il Raspberry Pi è stato sviluppato nel 2011-12 in Inghilterra dalla fondazione senza scopo di lucro Raspberry Pi per promuovere l'insegnamento dell'informatica. Costruito sulla base della logica digitale presente in un telefono cellulare economico, questo calcolatore è diventato incredibilmente popolare, tanto da vendere più di 3 milioni di esemplari nel solo 2014. Il nome ("lampone") è un omaggio ai primi home computer, come Apple ("mela"), Apricot ("albicocca") e Tangerine ("mandarino"). Pi deriva da Python, un linguaggio di programmazione spesso utilizzato nell'insegnamento. La documentazione e le informazioni per l'acquisto si possono trovare sul sito `raspberrypi.org`.

Eben Upton (1978-) è l'architetto del Raspberry Pi e il fondatore della Fondazione Raspberry Pi. Si è laureato e ha ottenuto il titolo di dottore di ricerca all'Università di Cambridge prima di entrare alla Broadcom Corporation come architetto di circuiti integrati.



(Fotografia © Eben Upton. Con permesso di riproduzione.)

Il Raspberry Pi continua a evolvere, e sono molto probabilmente disponibili nuovi modelli con processori più avanzati e un diverso insieme di I/O rispetto a quelli descritti nel capitolo. Ma i principi di funzionamento qui descritti valgono ancora, e valgono anche per altri tipi di microcontrollori, dal momento che tutti usano gli stessi tipi di periferiche. Basta consultare la documentazione tecnica per conoscere come sono mappate le periferiche, quali sono i piedini sul chip e i contatti sulla scheda, quali sono gli indirizzi dei registri di I/O associati a ciascuna periferica. Si devono poi programmare opportunamente i registri di configurazione per inizializzare ogni periferica e usare i registri dati per accedere alla periferica.

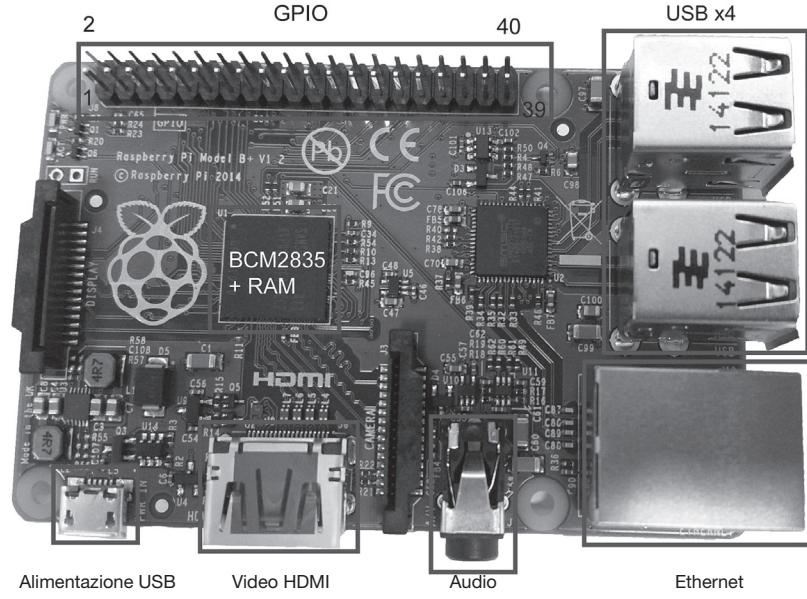


Figura e9.2 Raspberry Pi modello B+.

abbreviata è disponibile sul sito di Raspberry Pi, con le indicazioni su come accedere alle periferiche dal processore ARM. Tale scheda dati descrive molti aspetti e vari registri di I/O non considerati per semplicità in questo capitolo.

`www.raspberrypi.org/documentation/hardware/`

La **Figura e9.3** mostra lo schema semplificato della scheda Raspberry Pi. La scheda riceve l'alimentazione a 5 V da un alimentatore USB e i regolatori producono i livelli 3.3, 2.5 e 1.8 V per I/O, parte analogica e funzioni varie. Il BCM2835 ha anche un regolatore switching interno che produce un basso valore di tensione variabile per consentire risparmi di energia al sistema a singolo chip. Si collega a un controllore USB/Ethernet e dispone anche di uscita video diretta. Dispone di 54 segnali di I/O configurabili, ma per ragioni di spazio solo una parte è accessibile all'utente tramite i contatti sul connettore a pettine (*header*). Il connettore fornisce anche le tensioni 3.3 V e 5 V e la massa per alimentare facilmente piccoli dispositivi collegati al Pi, ma la massima corrente disponibile è di 50 mA dai 3.3 V e circa 300 mA dai 5 V. I modelli B e B+ sono simili, ma il B+ aumenta da 26 a 40 i contatti del connettore a pettine e da 2 a 4 le porte USB. Sono disponibili in commercio vari cavi per collegare il connettore a pettine a una basetta sperimentale (*breadboard*) per la prototipazione di circuiti.

Il Raspberry Pi usa una scheda SD come memoria disco Flash. La scheda SD è solitamente precaricata con Raspbian Linux, una versione compatta di Linux che sta in una scheda SD da 8 GB. Si può lavorare con Raspberry Pi sia collegandogli un monitor HDMI e una tastiera e mouse USB, usandolo quindi come calcolatore autonomo, sia collegandosi con un calcolatore esterno tramite cavo Ethernet.

9.3.2 Driver di dispositivo

I programmatori possono gestire i dispositivi di I/O leggendo e scrivendo direttamente nei registri di I/O mappati in memoria. È tuttavia una migliore pratica di programmazione quella di usare chiamate a funzioni che gestiscono gli accessi ai dispositivi di I/O. Tali funzioni sono denominate **driver di dispositivo**. Tra i vantaggi nell'uso dei driver di dispositivo si possono elencare i seguenti:

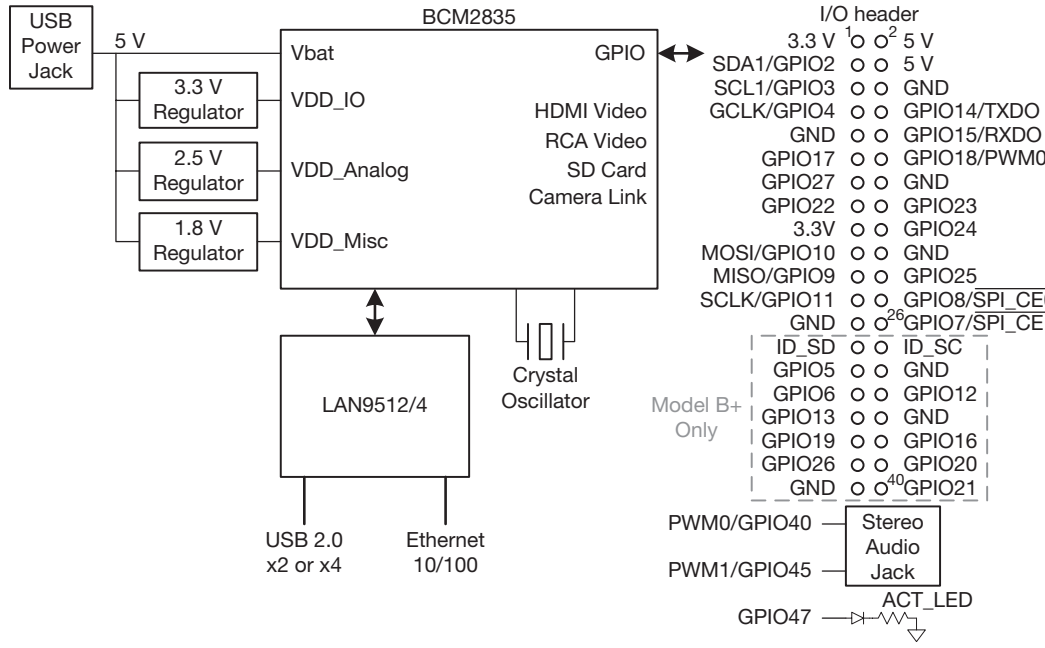


Figura e9.3 Schema dell'I/O del Raspberry Pi.

- Il codice è di più facile lettura se effettua chiamate a funzioni con nomi auto esplicativi piuttosto di scrivere campi di bit a strani indirizzi di memoria.
- Il driver di dispositivo è scritto da persone esperte dei dettagli di uso dei dispositivi di I/O, e i programmatori occasionali possono chiamare le funzioni del driver senza conoscerne i dettagli.
- Il codice è più facilmente portabile su un altro processore con diversa mappatura della memoria o dei dispositivi di I/O, perché l'unica cosa da modificare è il driver.
- Se il driver fa parte del sistema operativo, questo può controllare l'accesso ai dispositivi di I/O condivisi fra i vari programmi in esecuzione, garantendone la sicurezza (evitando per esempio che un virus possa leggere la tastiera mentre l'utente sta inserendo la propria password nel Web browser).

In questo paragrafo si sviluppa un semplice driver di dispositivo, chiamato EasyPIO, di accesso ai dispositivi di I/O del BCM2835, per far capire cosa succede all'interno di un driver. Si può anche evitare la fatica e usare direttamente WiringPi, una libreria di I/O *open source* per il Pi con funzioni simili anche se non proprio identiche a quelle di EasyPIO.

L'I/O mappato in memoria del BCM2835 si trova agli indirizzi fisici compresi tra 0x20000000 e 0x20FFFFFF. Gli indirizzi fisici base usati dalle varie periferiche sono riassunti nella **Tabella e9.1**. Le periferiche hanno vari registri di I/O situati a partire dal proprio indirizzo base. Per esempio, una lettura all'indirizzo 0x20200034 restituisce i valori dei piedini GPIO (*General Purpose I/O*) 31:0. Le periferiche evidenziate in neretto vengono discusse nei vari paragrafi del capitolo.

Raspberry Pi esegue normalmente un sistema operativo Linux con memoria virtuale, che complica ulteriormente l'I/O mappato in memoria: le letture e le scritture dei programmi fanno riferimento a indirizzi virtuali e non fisici, quindi un programma non può accedere direttamente ai dispositivi di I/O. Deve invece chiedere al sistema operativo di mappare gli indirizzi fisici di interesse nel proprio spazio di indirizzamento virtuale. La funzione `pioInit` di

Al tempo della stampa della versione originale di questo testo, la Fondazione Raspberry Pi ha rilasciato il Raspberry Pi 2 modello B, con un sistema a singolo chip BCM2836 contenente un processore quad Cortex-A7 e 1 GB di RAM. Il Pi 2 lavora circa 6 volte più velocemente del B+ ma ha lo stesso I/O del B+ descritto in questo capitolo. L'indirizzo base delle periferiche è stato spostato da 0x20000000 a 0x3F000000.

Attenzione: la disposizione dei contatti nel connettore di I/O è cambiata nelle varie revisioni di Raspberry Pi.

Attenzione: collegare 5 V a uno dei contatti di I/O a 3.3 V danneggia l'I/O e potenzialmente l'intero Raspberry Pi. Se si misura la tensione con una sonda, fare attenzione a non collegare accidentalmente i contatti a 5 V con i contatti vicini!

Il contatto 1 del connettore di I/O a pettine è marcato nella Figura e9.3. Quando si fanno i collegamenti, fare attenzione a non ruotare di 180° (cioè capovolgere) il connettore: è un errore tipico che può portare a danneggiare accidentalmente l'intero Pi.

Tabella e9.1 Indirizzi degli I/O mappati in memoria.

Indirizzo fisico base	Periferica
0x20003000	**Timer di sistema**
0x2000B200	Interrupt
0x2000B400	Timer di ARM
0x20200000	**GPIO**
0x20201000	**UART0**
0x20203000	PCM audio
0x20204000	**SPI0**
0x20205000	Master I²C n°1
0x2020C000	**PWM**
0x20214000	Slave I²C
0x20215000	miniUART1, SPI1, SPI2
0x20300000	Controllore di schede SD
0x20804000	Master I²C n°2
0x20805000	Master I²C n°3

Per motivi di sicurezza, Linux consente l'accesso ai dispositivi mappati in memoria al solo superuser. Per eseguire un programma in modalità superuser, si deve inserire `sudo` prima del comando Linux. Nel prossimo paragrafo si fa un esempio di questa operazione.

EasyPIO riportata nell'Esempio e9.2 svolge questo compito. Il codice implica qualche complessa manipolazione di puntatori in C. Il principio generale è quello di aprire `/dev/mem`, cioè il metodo Linux per accedere alla memoria fisica. Poi si usa la funzione `mmap` per usare `gpio` come puntatore all'indirizzo fisico 0x20200000, ovvero l'inizio dei registri GPIO. Il puntatore è dichiarato di tipo `volatile`, per segnalare al compilatore che il valore dell'I/O mappato in memoria può cambiare, quindi che il programma deve sempre leggere direttamente il registro invece di basarsi su valori precedenti. GPLEV0 accede al registro che si trova 13 parole dopo GPIO, cioè 0x20200034, dove si trovano i bit 31:0 di GPIO. Per brevità, l'esempio omette i controlli di errore che sono invece presenti nell'effettiva libreria EasyPIO. I prossimi sottoparagrafi illustrano nei dettagli i registri e le funzioni di accesso ai dispositivi di I/O.

ESEMPIO e9.2

Inizializzazione dell'I/O mappato in memoria

```
#include <sys/mman.h>
#define BCM2835_PERI_BASE 0x20000000
#define GPIO_BASE         (BCM2835_PERI_BASE + 0x200000)
volatile unsigned int    *gpio; //Pointer to base of gpio
#define GPLEV0            (* (volatile unsigned int *) (gpio + 13))
#define BLOCK_SIZE        (4*1024)
void pioInit(){
  int mem_fd;
  void *reg_map;

  // /dev/mem is a psuedo-driver for accessing memory in Linux
  mem_fd = open("/dev/mem", O_RDWR|O_SYNC);
  reg_map = mmap(
   NULL,                  // Address at which to start local mapping (null = don't-care)
   BLOCK_SIZE,            // 4KB mapped memory block
   PROT_READ|PROT_WRITE,  // Enable both reading and writing to the mapped memory
   MAP_SHARED,            // Nonexclusive access to this memory
   mem_fd,                // Map to /dev/mem
   GPIO_BASE);            // Offset to GPIO peripheral

  gpio = (volatile unsigned *)reg_map;
  close(mem_fd);
}
```

9.3.3 I/O digitali di uso generale

Gli I/O digitali di uso generale (GPIO, *General Purpose I/O*) sono usati per leggere e scrivere segnali digitali. Per esempio, la **Figura e9.4** mostra tre diodi emettitori di luce (LED) e tre interruttori (*switch*) collegati a 6 GPIO. I LED sono collegati in modo da accendersi se ricevono 1 e da spegnersi se ricevono 0. Le resistenze limitatrici di corrente in serie ai LED servono a regolarne la luminosità e a evitare di sovraccaricare l'emissione di corrente da parte del GPIO. Gli interruttori sono collegati in modo da generare 1 se chiusi e 0 se aperti. Nello schema sono indicati i nomi dei GPIO e le posizioni corrispondenti sul connettore a pettine.

Come minimo, ogni piedino GPIO richiede registri per leggere i valori in ingresso dal GPIO, scrivere i valori in uscita al GPIO e definire la direzione del GPIO stesso. In molti sistemi embedded, i GPIO possono essere condivisi con una o più periferiche di tipo speciale, quindi servono registri di configurazione aggiuntivi per determinare se il GPIO è di uso generale o speciale. Inoltre, si possono dover generare interrupt quando si verifica un evento come un fronte di salita o di discesa su un GPIO, quindi possono servire altri registri di configurazione per specificare le condizioni di interrupt.

Come detto, il BCM2835 ha 54 GPIO, controllati dai registri GPFSEL, GPLEV, GPSET e GPCLR. La **Figura e9.5** mostra una mappa di memoria per questi registri GPIO. I registri GPFSEL5…0 determinano se ciascun GPIO è un ingresso di tipo generale, un'uscita di tipo generale o un I/O speciale. Ciascuno di questi registri di selezione della funzione usa 3 bit per specificare la tipologia di ogni GPIO, quindi ogni registro a 32 bit controlla 10 GPIO, come riportato nella **Tabella e9.2**, e servono 6 registri GPFSEL per controllare tutti i 54 GPIO. Per esempio, il GPIO13 è configurato dai bit GPFSEL1[11:9]. Le possibili configurazioni sono riportate nella **Tabella e9.3**: molti piedini GPIO hanno più funzioni speciali discusse nei prossimi sottoparagrafi; la più usata è ALT0. Una lettura di GPLEV1…0 restituisce i valori dei piedini: per esempio, GPIO14 viene letto come GPLEV0[14] e GPIO34 come GPLEV1[2]. I piedini non possono essere scritti direttamente: possono essere forzati alti o bassi attivando i corrispondenti bit nei registri GPSET1…0 o GPCLR1…0. Per esempio, GPIO14 è forzato a 1 scrivendo GPSET0[14] = 1 e forzato a 0 scrivendo GPCLR0[14] = 1.

La scheda dati del BMC 2835 non specifica i livelli logici né la massima corrente di uscita dei GPIO: ma gli utenti hanno determinato sperimentalmente che non si devono assorbire più di 16 mA di corrente da ogni singolo GPIO né più di 50 mA in totale da tutti i GPIO. Quindi un GPIO va bene per pilotare un LED ma non un motore. I GPIO sono compatibili con altri chip a 3.3 V ma non sopportano la tensione di 5 V.

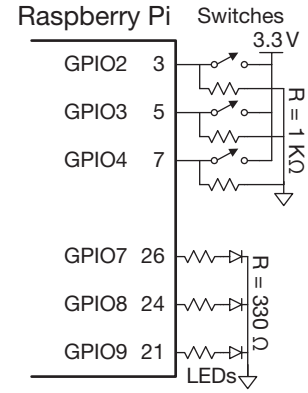


Figura e9.4
LED e *switch* collegati ai piedini GPIO.

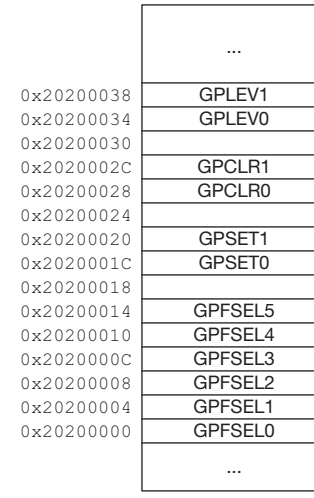


Figura e9.5
Mappa di memoria dei GPIO.

Tabella e9.2 Mappatura dei GPIO nei campi di bit dei registri GPFSEL.

	GPFSEL0	GPFSEL1	GPFSEL2	GPFSEL3	GPFSEL4	GPFSEL5
[2:0]	GPIO0	GPIO10	GPIO20	GPIO30	GPIO40	GPIO50
[5:3]	GPIO1	GPIO11	GPIO21	GPIO31	GPIO41	GPIO51
[8:6]	GPIO2	GPIO12	GPIO22	GPIO32	GPIO42	GPIO52
[11:9]	GPIO3	GPIO13	GPIO23	GPIO33	GPIO43	GPIO53
[14:12]	GPIO4	GPIO14	GPIO24	GPIO34	GPIO44	
[17:15]	GPIO5	GPIO15	GPIO25	GPIO35	GPIO45	
[20:18]	GPIO6	GPIO16	GPIO26	GPIO36	GPIO46	
[23:21]	GPIO7	GPIO17	GPIO27	GPIO37	GPIO47	
[26:24]	GPIO8	GPIO18	GPIO28	GPIO38	GPIO48	
[29:27]	GPIO9	GPIO19	GPIO29	GPIO39	GPIO49	

Tabella e9.3 Configurazione di GPFSEL.

GPFSEL	Funzione del piedino
000	Ingresso
001	Uscita
010	ALT5
011	ALT4
100	ALT0
101	ALT1
110	ALT2
111	ALT3

Il BCM2835 ha una modalità di accesso ai GPIO particolarmente complicata. Alcuni microcontrollori usano un solo registro per configurare se ciascun piedino è un ingresso o un'uscita, e un altro registro per leggere e scrivere i piedini.

ESEMPIO e9.3

GPIO per interruttori e LED. Estendere EasyPIO con le funzioni `pinMode`, `digitalRead` e `digitalWrite` per configurare la direzione di ciascun piedino e per leggerne o scriverne il valore. Scrivere un programma in C che usi queste funzioni per leggere lo stato dei tre interruttori della Figura e9.4 e accendere o spegnere i LED corrispondenti.

Soluzione Il codice aggiuntivo per EasyPIO è riportato sotto. Dal momento che si usano vari registri per controllare l'I/O, le funzioni devono per prima cosa calcolare a quale registro accedere e quale spiazzamento di bit usare nel registro. `pinMode` forza a 0 o a 1 i tre bit a seconda della funzione richiesta. `digitalWrite` effettua la scrittura di 1 o 0 usando i registri GPDSET oppure GPCLR. `digitalRead` restituisce il valore del piedino desiderato e maschera, azzerandoli, tutti gli altri.

```
#define GPFSEL ((volatile unsigned int *) (gpio + 0))
#define GPSET  ((volatile unsigned int *) (gpio + 7))
#define GPCLR  ((volatile unsigned int *) (gpio + 10))
#define GPLEV  ((volatile unsigned int *) (gpio + 13))
#define INPUT  0
#define OUTPUT 1
...
void pinMode(int pin, int function) {
  int reg      = pin/10;
  int offset   = (pin%10)*3;
  GPFSEL[reg] &= ~((0b111 & ~function) << offset);
  GPFSEL[reg] |=  ((0b111 &  function) << offset);
}

void digitalWrite(int pin, int val) {
  int reg    = pin / 32;
  int offset = pin % 32;

  if  (val) GPSET[reg] = 1 << offset;
  else      GPCLR[reg] = 1 << offset;
}
int digitalRead(int pin) {
  int reg    = pin / 32;
  int offset = pin % 32;

  return (GPLEV[reg] >> offset) & 0x00000001;
}
```

Il programma per leggere gli interruttori e attivare i LED è riportato sotto. Per prima cosa inizializza l'accesso ai GPIO, poi definisce i piedini 2-4 come ingressi per gli interruttori e i piedini 7-9 come uscite per i LED. Quindi continua a leggere i valori degli interruttori e a scriverli nei LED corrispondenti.

```
#include "EasyPIO.h"

void main(void) {
  pioInit();
  // Set GPIO 4:2 as inputs
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(4, INPUT);

  // Set GPIO 9:7 as an output
  pinMode(7, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
  while (1) { // Read each switch and write corresponding LED
    digitalWrite(7, digitalRead(2));
    digitalWrite(8, digitalRead(3));
    digitalWrite(9, digitalRead(4));
  }
}
```

Se il programma è memorizzato nel file `dip2led.c` e il file `EasyPIO.h` si trova nella stessa cartella, lo si può compilare ed eseguire usando i seguenti comandi sulla linea di comando del Raspberry Pi. `gcc` è il compilatore C. Si noti la necessità dell'opzione `sudo` perché il programma possa accedere alla memoria di I/O protetta. Per interrompere il programma in esecuzione, premere Ctrl-C.

```
gcc dip2led.c -o dip2led
sudo ./dip2led
```

9.3.4 I/O seriale

Se il microcontrollore deve inviare più bit di quanti sono i piedini GPIO disponibili, deve suddividere il messaggio in parti più piccole da trasmettere. In ogni passo, può trasmettere un solo bit oppure più bit: nel primo caso si parla di I/O seriale, nel secondo caso di I/O parallelo. L'I/O seriale è molto diffuso perché usa meno fili di connessione ed è sufficientemente veloce per molte applicazioni; è così diffuso che sono stati definiti vari standard per le trasmissioni seriali, e molti microcontrollori dispongono di hardware dedicato per facilitare le comunicazioni basate su questi standard. In questo paragrafo si descrivono le due interfacce seriali standard: *Serial Peripheral Interface* (SPI, periferica di interfaccia seriale) e *Universal Asynchronous Receiver/Transmitter* (UART, ricevitore/trasmettitore asincrono universale).

Altri standard seriali diffusi sono I^2C (*Inter-Integrated Circuit*), USB (*Universal Serial Bus*) ed Ethernet. I^2C è un'interfaccia a due fili con un piedino di clock e uno bidirezionale per i dati; viene usato in modo simile a SPI. USB ed Ethernet sono standard più complessi ad alte prestazioni, descritti rispettivamente nei paragrafi 9.6.1 e 9.6.4. Tutti e cinque gli standard sono supportati da Raspberry Pi.

9.3.4.1 *Serial Peripheral Interface* (SPI)

SPI è un semplice protocollo seriale sincrono, facile da usare e relativamente veloce. L'interfaccia fisica è costituita da tre piedini: *Serial Clock* (SCK, il clock seriale), *Master Out Slave In* (MOSI, trasmissione da master a slave, denominato anche SDO da *Serial Data Out*) e *Master In Slave Out* (MISO, trasmissione da slave a master, denominato anche SDI da *Serial Data In*). SPI collega dunque un dispositivo **master** a un dispositivo **slave**, come mostrato nella **Figura e9.6(a)**. Il master genera il clock, e inizia la trasmissione inviando una serie di impulsi di clock su SCK. Se vuole spedire dati allo slave, li invia su

SPI spedisce sempre dati in entrambe le direzioni a ogni trasferimento. Se serve una comunicazione unidirezionale, il sistema deve semplicemente ignorare i bit passati nell'altra direzione. Per esempio, se il master deve solo inviare un dato allo slave, il byte ricevuto dallo slave viene ignorato. Se il master deve solo ricevere un dato dallo slave, deve comunque attivare la comunicazione SPI inviando un byte qualsiasi che lo slave ignora, poi può leggere il dato ricevuto. Il clock di SPI commuta solo quando il master sta trasmettendo dati.

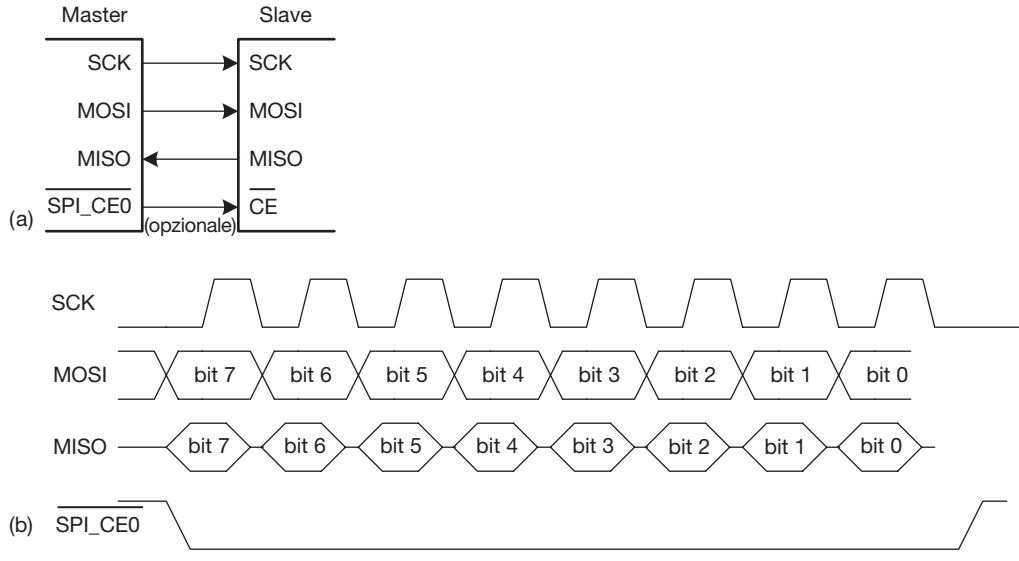


Figura e9.6 Collegamenti e forme d'onda di SPI.

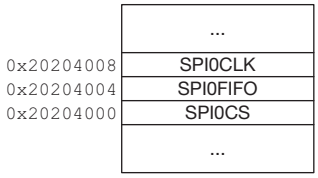


Figura e9.7 Registri della Porta SPI Master 0.

Se la frequenza è troppo alta (≥1 MHz su una basetta sperimentale, o decine di MHz su una scheda di circuito stampato senza terminatori) SPI può diventare inaffidabile a causa di riflessioni, fenomeni di crosstalk o altri problemi di integrità dei segnali.

MOSI partendo dal bit più significativo. Lo slave può rispondere contemporaneamente inviando i suoi dati su MISO. La **Figura e9.6(b)** mostra le forme d'onda SPI per la trasmissione di dati da 8 bit. I dati cambiano sul fronte di discesa di SCK e devono essere stabili per il campionamento sul fronte di salita. L'interfaccia SPI può anche inviare un segnale di *Chip Enable* (abilitazione chip) attivo basso per avvisare il ricevitore che i dati stanno arrivando.

Il BCM2835 ha tre porte SPI master e una slave. In questo paragrafo si descrive la Porta SPI Master 0, accessibile ai contatti GPIO 11:9 del Raspberry Pi: per usare questi contatti come SPI invece di GPIO, il loro GPFSEL deve essere impostato ad ALT0, quindi il Pi deve configurare la porta. Quando il Pi scrive in SPI, il dato viene trasmesso serialmente allo slave. I dati ricevuti contemporaneamente dallo slave vengono memorizzati e Pi li può leggere quando il trasferimento è finito.

La Porta SPI Master 0 è associata a tre registri, indicati nella mappa di memoria riportata nella **Figura e9.7**. SPI0CS è il registro di controllo, usato per attivare SPI e definire alcuni attributi come la polarità del clock; la **Tabella e9.4** elenca nomi e funzioni dei bit di SPI0CS rilevanti per quanto qui discusso: tutti i bit valgono 0 al reset. Molte funzioni, come le selezioni di chip e gli interrupt, non sono usate in questo paragrafo, ma si possono trovare sulla documentazione tecnica. Il registro SPI0FIFO viene scritto per trasmettere un byte e letto per ottenere il byte ricevuto. Il registro SPI0CLK configura la frequenza del clock di SPI dividendo il clock periferico a 250 MHz per la potenza di due specificata nel registro, come riportato nella **Tabella e9.5**.

Tabella e9.4 Campi del registro SPI0CS.

Bit	Nome	Funzione	Significato se 0	Significato se 1
16	DONE	Trasferimento fatto	Trasmissione in corso	Trasmissione completata
7	TA	Trasferimento attivo	SPI disabilitata	SPI abilitata
3	CPOL	Polarità del clock	Clock non attivo se basso	Clock non attivo se alto
2	CPHA	Fase del clock	Prima transizione di SCK a metà del bit di dato	Prima transizione di SCK all'inizio del bit di dato

Tabella e9.5 Frequenze di SPI0CLK.

SPI0CLK	Frequenza di SPI (kHz)
2	125 000
4	62 500
8	31 250
16	15 625
32	7812
64	3906
128	1953
256	976
512	488
1024	244
2048	122

ESEMPIO e9.4

Trasmissione e ricezione di byte con SPI. Progettare un sistema per comunicare tra un Raspberry Pi master e un dispositivo FPGA slave tramite SPI. Tracciare uno schema dell'interfaccia; scrivere il codice C per il Pi necessario per trasmettere il carattere "A" e ricevere un altro carattere in risposta; scrivere il codice HDL per uno slave SPI sul dispositivo FPGA. Come si può semplificare lo slave se deve solo ricevere dati?

Soluzione La **Figura e9.8** mostra il collegamento tra i due dispositivi utilizzando la Porta SPI Master 0. I numeri dei contatti sono ricavati dalla documentazione tecnica (per es., la Figura e9.3). Nella figura sono riportati sia i numeri dei contatti sia i loro nomi per indicare la connessione a livello fisico e logico. Se SPI è abilitata, i contatti non possono essere usati come GPIO.

Il codice seguente, preso da EasyPIO, è usato per predisporre SPI a trasmettere e ricevere un carattere. Il codice necessario per definire la mappa di memoria e gli indirizzi dei registri è simile a quello già visto per i GPIO e non viene qui riportato nuovamente.

```
void spiInit(int freq, int settings) {
  pinMode(8, ALT0);      // CEOb
  pinMode(9, ALT0);      // MISO
```

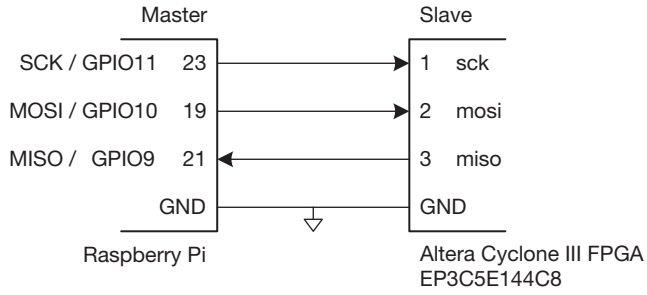


Figura e9.8
Connessioni SPI tra il Pi e un dispositivo FPGA.

```
  pinMode(10, ALT0);        // MOSI
  pinMode(11, ALT0);        // SCLK

  SPI0CLK = 250000000/freq; // Set SPI clock divider to desired freq

  SPI0CS = settings;
  SPI0CSbits.TA = 1;        // Turn SPI on
}
char spiSendReceive(char send){
  SPI0FIFO = send;          // Send data to slave
  while (!SPI0CSbits.DONE); // Wait until SPI complete
  return SPI0FIFO;          // Return received data
}
```

Il codice C seguente inizializza SPI, quindi spedisce e riceve un carattere. Il clock di SPI viene impostato a 244 kHz.

```
#include "EasyPIO.h"

void main(void) {
  char received;

  pioInit();
  spiInit(244000, 0);             // Initialize the SPI:
                                  // 244 kHz clk, default settings
  received = spiSendReceive('A'); // Send letter A and receive byte
}
```

Il codice HDL per il dispositivo FPGA è riportato sotto. La **Figura e9.9** mostra il diagramma a blocchi e le temporizzazioni. Il dispositivo FPGA usa un registro a scorrimento per memorizzare i bit ricevuti dal master e i bit che restano da spedire al master.

Al primo fronte di salita di `sck` dopo il reset, e in seguito ogni 8 cicli di clock, un nuovo byte viene caricato da `d` nel registro a scorrimento. A ogni ciclo successivo, un bit viene fatto scorrere in ingresso da `mosi` e un bit viene fatto scorrere in uscita verso `miso`. `miso` viene ritardato fino al fronte di discesa di `sck`, in modo che possa essere campionato dal master al prossimo fronte di salita. Dopo 8 cicli, il byte ricevuto è disponibile in `q`.

```
module spi_slave(input logic sck,       // From master
                 input logic mosi,      // From master
                 output logic miso,     // To master
                 input logic reset,     // System reset
                 input logic [7:0] d,   // Data to send
                 output logic [7:0] q); // Data received

 logic [2:0] cnt;
 logic       qdelayed;

 // 3-bit counter tracks when full byte is transmitted
 always_ff @(negedge sck, posedge reset)
  if (reset) cnt = 0;
  else       cnt =cnt + 3'b1;
 // Loadable shift register
```

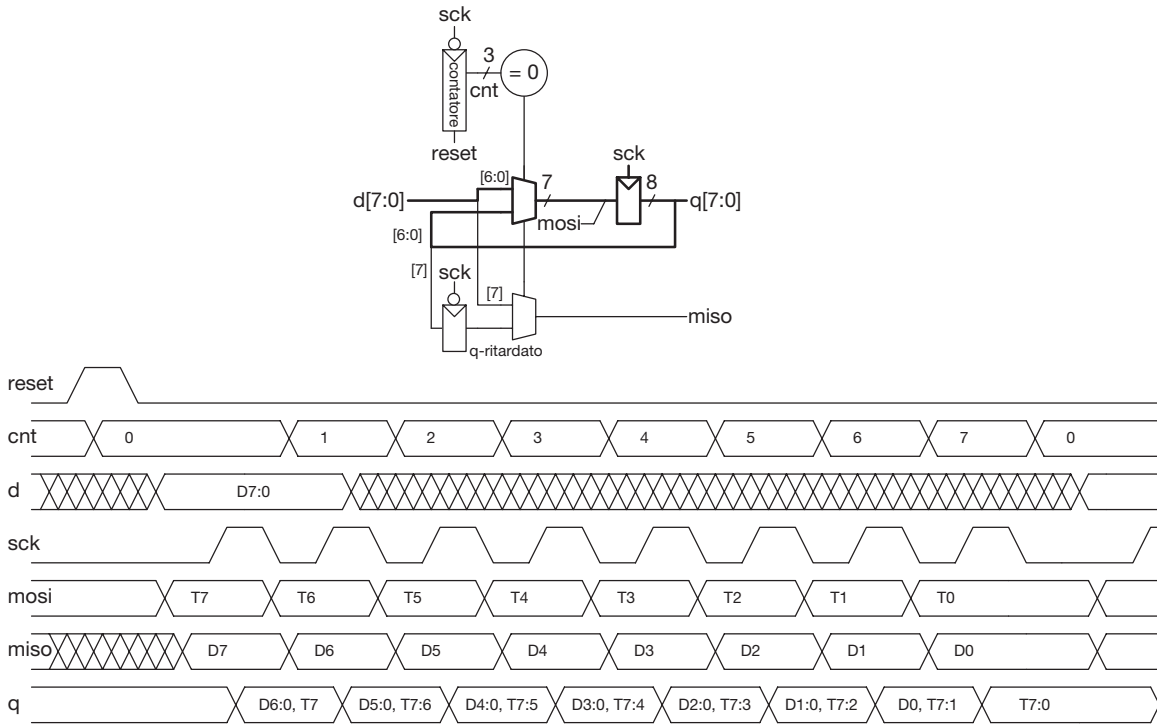


Figura e9.9 Struttura circuitale e temporizzazioni dello slave SPI.

```
 // Loads d at the start, shifts mosi into bottom on each step
 always_ff @(posedge sck)
  q <= (cnt == 0) ? {d[6:0], mosi} : {q[6:0], mosi};

 // Align miso to falling edge of sck
 // Load d at the start
 always_ff @(negedge sck)
  qdelayed = q[7];
 assign miso = (cnt == 0) ? d[7] : qdelayed;
endmodule
```

Se lo slave deve solo ricevere dati, si riduce a un semplice registro a scorrimento, descritto dal seguente codice HDL.

```
module spi_slave_receive_only(input logic  sck,      // From master
                              input logic  mosi,     // From master
                              output logic [7:0] q); // Data received
 always_ff @(posedge sck)
  q <= {q[6:0], sdi}; // shift register

endmodule
```

Le porte SPI sono ampiamente configurabili, quindi possono comunicare con una notevole varietà di dispositivi seriali: questo però può condurre a configurare in modo scorretto la porta e ad avere errori di comunicazione. A volte è necessario cambiare i bit di configurazione per comunicare con dispositivi che prevedono temporizzazioni differenti: quando CPOL = 1, SCK è negato; quando CPHA = 1, il clock commuta mezzo ciclo in anticipo rispetto ai dati. Queste modalità sono mostrate nella **Figura e9.10**: bisogna fare attenzione al fatto che diversi prodotti SPI possono usare nomi e polarità differenti per queste opzioni, quindi è importante controllare attentamente le forme d'onda sul proprio dispositivo. Può essere utile analizzare SCK, MOSI e MISO con un oscilloscopio quando si verificano problemi di comunicazione seriale.

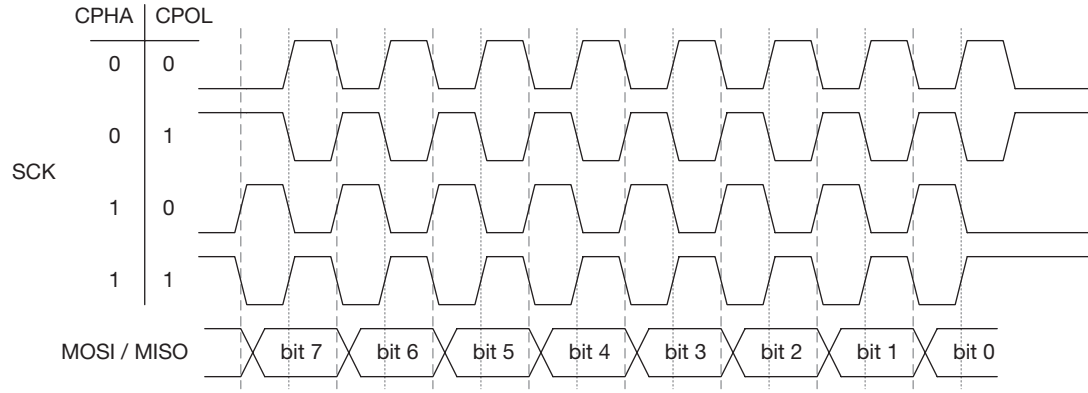


Figura e9.10 **Configurazioni delle temporizzazioni di clock e dati per SPI.**

9.3.4.2 *Universal Asynchronous Receiver/Transmitter* (UART)

Un dispositivo UART è una periferica di I/O seriale che effettua la comunicazione tra due sistemi senza inviare il segnale di clock. I due sistemi devono accordarsi preventivamente sul tasso di trasmissione dei bit, ed entrambi devono generare localmente il proprio clock. Quindi la trasmissione è asincrona perché i due clock non sono sincronizzati. Anche se tali clock possono avere piccole differenze di frequenza e fasi prive di correlazione, il dispositivo UART gestisce in modo affidabile la comunicazione. Dispositivi UART sono usati in protocolli come RS-232 e RS-485: per esempio, le porte seriali dei vecchi computer usavano lo standard RS-232C introdotto nel 1969 dalle *Electronics Industries Associations*. Tale standard inizialmente prevedeva di collegare un sistema DTE (*Digital Terminal Equipment*), come un calcolatore dipartimentale (*mainframe*), a un dispositivo DCE (*Digital Communication Equipment*) come un modem. Anche se i dispositivi UART sono lenti rispetto a SPI e soggetti a errori di configurazione, i suddetti standard sono stati utilizzati per così tanto tempo che anche oggi rimangono dispositivi importanti.

La **Figura e9.11(a)** mostra un collegamento seriale asincrono. Il DTE invia dati al DCE tramite la linea TX e li riceve tramite la linea RX. La **Figura e9.11(b)** mostra una di queste due linee mentre invia un carattere al tasso di trasmissione di 9600 baud. Le linee, quando non sono utilizzate, sono inattive (*idle*) al valore 1. Ogni carattere viene inviato partendo da un bit di start di valore 0, seguito da 7 o 8 bit di dato, un bit opzionale di parità, e uno o più bit di stop di valore 1. Il dispositivo UART rileva il fronte di discesa della linea da inattiva a start per agganciarsi in modo corretto alla trasmissione. Anche se sette bit sono sufficienti per trasmettere un carattere ASCII, si usano di solito otto bit perché in grado di trasmettere un byte qualsiasi.

Il bit opzionale di parità permette al sistema di accorgersi se un bit è stato alterato durante la trasmissione. Può essere configurato come **parità pari** (*even*) o come **parità dispari** (*odd*); nel primo caso il bit di parità viene definito in modo tale che il numero totale di uni presenti nei bit di dato e nel

Il baud rate (tasso di baud) dà il tasso di segnalazione misurato in numero di simboli al secondo, mentre il bit rate (tasso di bit) dà il tasso di trasmissione dati misurato in numero di bit al secondo. Il tipo di segnalazione discusso in questo testo è a due livelli, dove ogni simbolo rappresenta un bit. Tuttavia, una segnalazione multi livello può inviare più bit per simbolo: per esempio, una segnalazione a quattro livelli può inviare due bit per simbolo: in questo caso, il bit rate risulta pari al doppio del baud rate. In un sistema semplice come SPI, dove ogni simbolo è un bit e ogni simbolo rappresenta un dato, baud rate e bit rate sono uguali. Invece, i dispositivi UART e altre convenzioni di segnalazione richiedono bit aggiuntivi rispetto ai dati. Per esempio, un sistema di segnalazione a due livelli che aggiunge un bit di start e uno di stop a dati di 8 bit e lavora a un baud rate di 9600 ha un bit rate pari a (9600 simboli/secondo) × (8 bit/10 simboli) = 7680 bit/secondo = 960 caratteri/secondo.

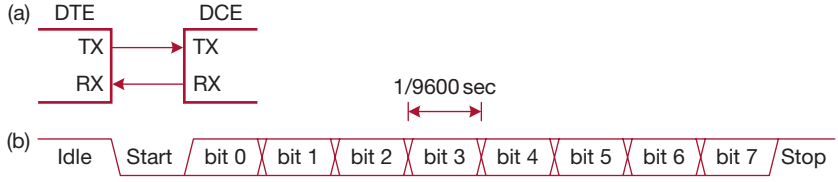


Figura e9.11
Collegamento seriale asincrono.

Dagli anni '50 agli anni '70, i primi hacker (che si autodefinivano i fricchettoni telefonici) avevano imparato a controllare i commutatori delle compagnie telefoniche inviando sulla linea opportuni toni. Il tono a 2600 Hz prodotto da un fischietto giocattolo trovato in una scatola di cereali della Cap'n Crunch poteva essere usato per fare gratis telefonate interurbane e internazionali.



Figura e9.12 Fischietto da nostromo della Cap'n Crunch.
(Fotografia di Evrim Sen. Con permesso di riproduzione.)

Il termine *handshaking* si riferisce alla negoziazione tra due sistemi: tipicamente, un sistema segnala che è pronto a trasmettere o ricevere dati, e l'altro sistema comunica di aver ricevuto la richiesta.

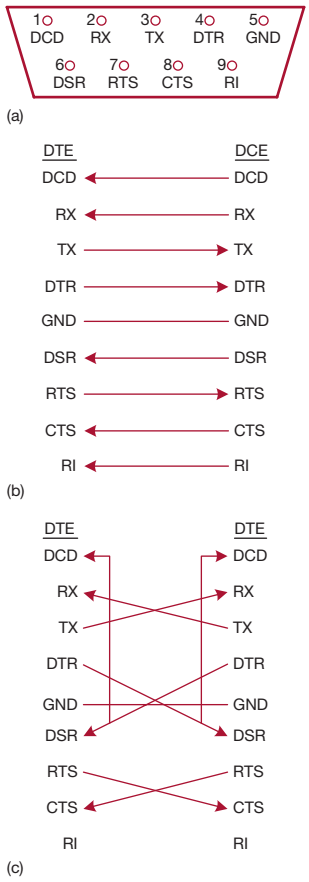


Figura e9.13
Connettore DE-9 maschio: (a) descrizione dei contatti, (b) collegamento standard, (c) collegamento null modem.

bit di parità sia pari: in altre parole, il bit di parità è lo XOR dei bit di dato. Il ricevitore può verificare se il numero di uni ricevuti è pari, e segnalare errore in caso contrario. La parità dispari è naturalmente l'opposto.

Una scelta comune è quella di trasmettere 8 bit di dato, nessuna parità e un bit di stop, in modo da usare in totale 10 simboli per trasportare un'informazione di 8 bit. I tassi di segnalazione si indicano quindi usando i baud come unità di misura invece dei bit/secondo. Per esempio, 9600 baud indica 9600 simboli/secondo ovvero 960 caratteri/secondo. Naturalmente entrambi i sistemi devono essere configurati per il baud rate adeguato e per i numeri di bit di dato, di parità e di stop, pena la perdita di dati. Questo può essere un problema, soprattutto per utenti con poche conoscenze tecniche, ed è il motivo per cui USB (*Universal Serial Bus*) ha sostituito i dispositivi UART nei personal computer.

I baud rate più tipici sono 300, 1200, 2400, 9600, 14 400, 19 200, 38 400, 57 600 e 115 200. I tassi più bassi erano usati negli anni '70 e '80 per i modem che inviavano dati sulle linee telefoniche mediante sequenze di toni audio. Nei sistemi più moderni, 9600 e 115 200 sono i due baud rate più frequenti: 9600 è usato quando la velocità non è un problema, 115 200 è usato perché è il baud rate più elevato, anche se comunque lento rispetto ad altri standard moderni di I/O seriale.

Lo standard RS-232 definisce vari segnali aggiuntivi. RTS (*Request To Send*) e CTS (*Clear To Send*) possono essere usati per effettuare ***handshaking*** (letteralmente stretta di mano) a livello hardware, e possono operare in due modi. Nel modo a controllo di flusso (*flow control*) il DTE porta RTS a 0 quando è pronto a ricevere dati dal DCE; analogamente, il DCE porta CTS a 0 quando è pronto a ricevere dati dal DTE. Nella documentazione tecnica i due segnali sono spesso indicati con una barra sopra la sigla per indicare che sono attivi bassi. Nel modo più vecchio (*simplex*) il DTE porta RTS a 0 quando è pronto a trasmettere, e il DCE risponde portando CTS a 0 quando è pronto a ricevere la trasmissione.

Alcuni sistemi, soprattutto quelli collegati tramite linea telefonica, usano anche DTR (*Data Terminal Ready*), DCD (*Data Carrier Detect*), DSR (*Data Set Ready*) e RI (*Ring Indicator*) per comunicare quando il sistema è connesso alla linea.

Lo standard originale raccomandava l'uso di un grosso connettore DB-25 a 25 contatti, ma i personal computer usano un connettore DE-9 maschio a 9 contatti con la struttura riportata nella **Figura e9.13(a)**. Normalmente i fili nel cavo sono collegati direttamente come mostrato nella **Figura e9.13(b)**. Quando però si devono collegare tra di loro due DTE, può servire il cavo denominato *null modem* mostrato nella **Figura e9.13(c)**, dove TX e RX sono incrociati e i segnali di *handshaking* connessi in modo opportuno. A tutto ciò si aggiunge il problema dei connettori maschi e femmine, con il risultato di aver bisogno di diverse versioni di cavi e di un buon intuito per riuscire a collegare due sistemi RS-232 (altro motivo per il passaggio a USB). Fortunatamente i sistemi embedded usano di solito connessioni semplificate a 3 o 5 fili, con massa (GND) TX, RX e a volte RTS e CTS.

RS-232 rappresenta il valore 0 con una tensione tra 3 V e 15 V, e il valore 1 con una tensione tra –3 V e –15 V: si parla in questo caso di segnalazione **bipolare**. Un dispositivo *transceiver* (ricetrasmettitore) converte i livelli logici provenienti dall'UART nei livelli positivi e negativi previsti da RS-232 e fornisce inoltre protezione da scariche elettrostatiche per evitare danni alla porta seriale quando l'utente inserisce la spina nel connettore. MAX3232E è un *transceiver* molto diffuso, compatibile sia con la logica a 3.3 V sia con quella a 5 V. Contiene una pompa di carica che insieme a condensatori esterni è in grado di generare uscite a ±5 V utilizzando un solo alimentatore a bassa

tensione. Alcune periferiche seriali progettate per sistemi embedded evitano il *transceiver* e usano semplicemente 0 V per il valore 0 e 3.3 V per il valore 1, quindi bisogna fare attenzione a cosa è riportato nella documentazione tecnica.

Il BCM2835 contiene due UART, denominati UART0 e UART1. Entrambi possono essere configurati per comunicare tramite i contatti 14 e 15, ma UART0 è più completo e viene descritto di seguito. Per usare questi contatti con UART0 invece di GPIO, i loro registri GPFSEL devono essere forzati a ALT0, quindi come nel caso di SPI il Pi deve prima configurare la porta. Al contrario di SPI, scritture e letture possono avvenire indipendentemente perché entrambi i sistemi collegati possono ricevere senza trasmettere e viceversa. I registri di UART0 sono mostrati nella **Figura e9.14**.

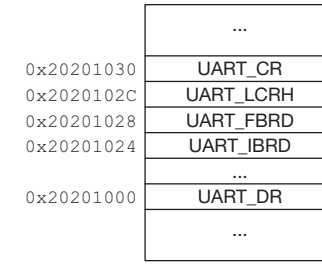


Figura e9.14
Registri di UART0.

Per configurare l'UART si deve per prima cosa impostare il baud rate. L'UART ha un clock interno a 3 MHz che deve essere diviso in modo da produrre un clock pari a 16 volte il baud rate desiderato, quindi il divisore BRD (*Baud Rate Divisor*) da utilizzare è:

$$\text{BRD} = 3\,000\,000/(16 \times \text{baud rate})$$

BRD è codificato con una parte intera a 16 bit in UART_IBRD e una parte frazionaria a 6 bit in UART_FBRD, ovvero: BRD = IBRD + FBRD/64. La **Tabella e9.6** riporta le impostazioni da utilizzare per i baud rate più usati[1].

Si può ora impostare il numero di bit di dato, di stop e di parità utilizzando il registro di controllo della linea UART_LCRH. Per default, l'UART ha un bit di stop e nessuna parità, ma stranamente lavora con dati a 5 bit, quindi il campo WLEN (i bit 6:5) di UART_LCRH va forzato a 3 per gestire dati a 8 bit. Infine si deve abilitare l'UART portando a 1 il bit di posto 0 (UARTEN) nel registro di controllo UART_CR.

I dati sono trasmessi e ricevuti usando il registro dati UART_DR e il registro di *framing* UART_FR. Per trasmettere un dato si deve attendere che il bit 7 (TXFE) di UART_FR sia 1 per indicare che il trasmettitore non è occupato, quindi scrivere il byte in UART_DR. Per ricevere un dato si deve attendere che il bit 4 (RXFE) di UART_FR sia 0 per indicare che il ricevitore ha un dato pronto, quindi leggere il byte da UART_DR.

Tabella e9.6 **Impostazioni di BRD.**

Baud rate desiderato	UART_IBRD	UART_FBRD	Baud rate effettivo	Errore (%)
300	625	0	300	0
1200	156	16	1200	0
2400	78	8	2400	0
9600	19	34	9600	0
19200	9	49	19200	0
38400	4	56	38461	0.16
57600	3	16	57692	0.16
115200	1	40	115384	0.16

[1] Il baud rate non sempre è un sottomultiplo esatto di 3 MHz, quindi alcuni divisori danno luogo a errori di frequenza, che però se modesti vengono tollerati dall'UART per la sua natura intrinsecamente asincrona.

ESEMPIO e9.5

Comunicazione seriale con un PC. Realizzare un circuito e un programma in C per consentire al Raspberry Pi di comunicare con un PC tramite la porta seriale a 115 200 baud, con 8 bit di dato, un bit di stop e nessun bit di parità. Il PC deve eseguire un programma di emulazione di terminale come PuTTY[2] per leggere e scrivere tramite la porta seriale. Il programma in C deve chiedere all'utente di inserire una stringa e mostrare all'utente la stringa inserita.

Soluzione La **Figura e9.15(a)** mostra lo schema di base del collegamento seriale evidenziando gli aspetti legati alla conversione di livello elettrico e al cablaggio. Poiché oggi pochi PC hanno ancora connettori per le porte seriali, si usa l'adattatore seriale da USB a RS-232 Plugable DB9 della `plugable.com` mostrato nella **Figura e9.16** per consentire la connessione seriale al PC. L'adattatore si collega a un connettore DE-9 femmina saldato ai fili di un *transceiver* che converte le tensioni dai valori bipolari di RS-232 al livello di 3.3 V del Pi. Sia Pi sia il PC sono configurati come *Data Terminal Equipment*, quindi i fili TX e RX devono essere incrociati. I segnali di *handshaking* RTS/CTS del Pi non sono usati, e i contatti RTS e CTS sul connettore DE-9 sono collegati tra loro in modo che il PC "stringa la mano a se stesso" (nessun *handshaking*). La **Figura e915(c)** mostra una soluzione più semplice che usa un cavo seriale da USB a TTL Adafruit 954: il cavo è compatibile con i livelli di tensione di 3.3 V e ha i contatti del connettore femmina collegabili direttamente ai contatti del connettore maschio del Raspberry Pi.

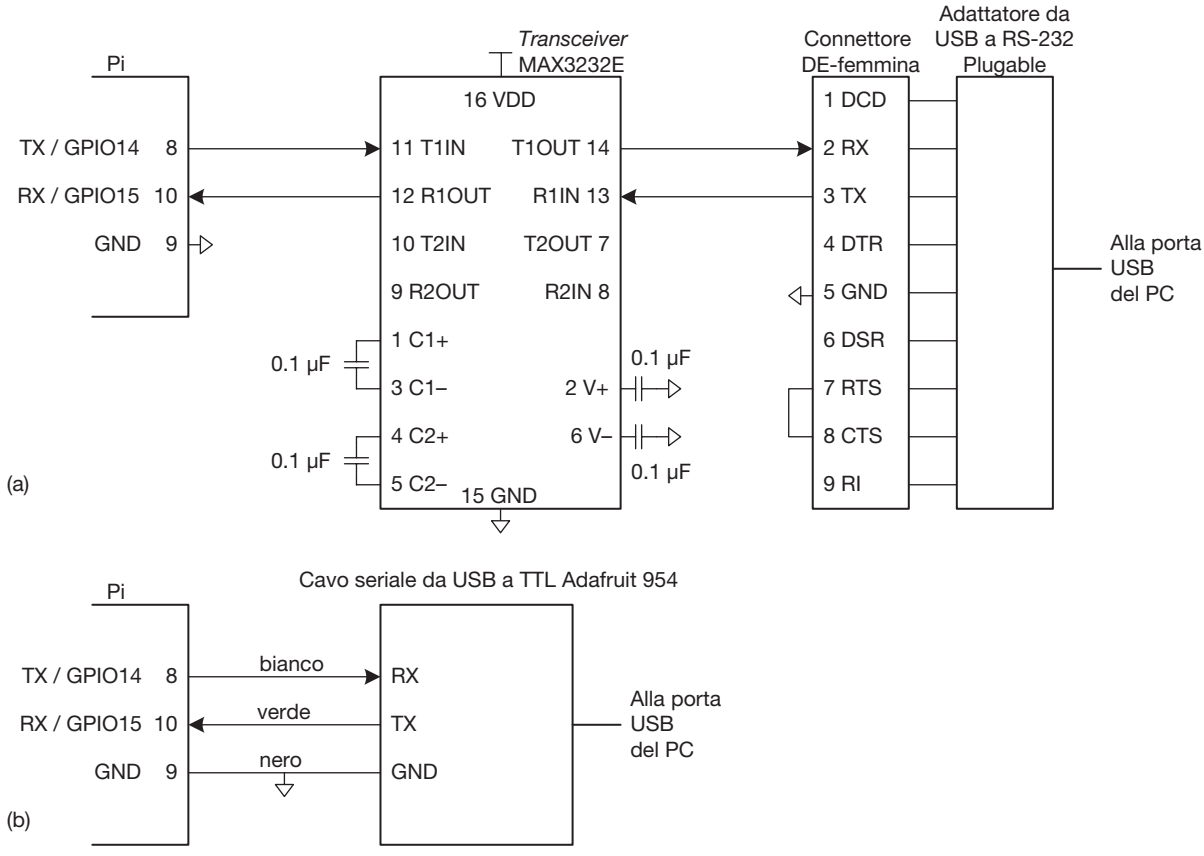


Figura e9.15 Collegamento seriale tra Rasperry Pi e un PC (a) cavo Plugable e (b) cavo Adafruit.

[2] PuTTY è scaricabile gratuitamente dal sito `www.putty.org`.

Per configurare PuTTY a operare con il collegamento seriale, si deve impostare *Connection type* a seriale e *Speed* a 115 200. Bisogna anche impostare *Serial line* alla porta COM usata dal sistema operativo per l'adattatore da USB a seriale. In Windows si deve andare in Gestione dispositivi; potrebbe essere il dispositivo COM3. Nel menu *Connessione seriale* si deve impostare il controllo di flusso a NESSUNO o RTS/CTS. Nel menù *Terminal* impostare l'Echo locale a Sempre attivo per vedere i caratteri su terminale quando si inseriscono da tastiera.

Il codice del driver della porta seriale presente in EasyPIO.h è riportato qui sotto. Il tasto Invio nel programma di emulazione di terminale corrisponde a un ritorno a capo (*Carriage Return*) rappresentato in *C* come `'\r'` avente codice ASCII 0x0D. Per passare alla riga di testo successiva quando si visualizza la risposta, serve inviare entrambi i caratteri `'\n'` e `'\r'` (nuova riga e ritorno a capo)[3]. La funzione `uartInit` configura l'UART come descritto prima, mentre le funzioni `getCharSerial` e `putCharSerial` attendono che l'UART sia pronto e poi leggono o scrivono un byte dal registro dati.

```
void uartInit(int baud) {
   uint fb = 12000000/baud;      // 3 MHz UART clock

   pinMode(14, ALT0);            // TX
   pinMode(15, ALT0);            // RX
   UART_IBRD = fb >> 6;          // 6 Fract, 16 Int bits of BRD
   UART_FBRD = fb & 63;
   UART_LCRHbits.WLEN = 3;       // 8 Data, 1 Stop, no Parity, no FIFO, no Flow
   UART_CRbits.UARTEN = 1;       // Enable uart
}

char getCharSerial(void) {
   while (UART_FRbits.RXFE);     // Wait until data is available
   return UART_DRbits.DATA;      // Return char from serial port
}

void putCharSerial(char c) {
   while (!UART_FRbits.TXFE);    // Wait until ready to transmit
   UART_DRbits.DATA = c;         // Send char to serial port
}
```

La funzione `main` mostra come scrivere su terminale e leggere da terminale una stringa di caratteri usando le funzioni `putStrSerial` e `getStrSerial`.

```
#include "EasyPIO.h"

#define MAX_STR_LEN 80

void getStrSerial(char *str) {
    int i = 0;
    do {                                              // Read an entire string until
       str[i] = getCharSerial();                      // Carriage return
    } while ((str[i++] !='\r') && (i < MAX_STR_LEN)); // Look for carriage return
    str[i-1] = 0;    // Null-terminate the string
}

void putStrSerial(char *str) {
    int i = 0;
    while (str[i] != 0) {                             // Iterate over string
       putCharSerial(str[i++]);                       // Send each character
    }
}
             int main(void) {
                 char str[MAX_STR_LEN];

  pioInit();
  uartInit(115200);                                   // Initialize UART with baud rate
```

Si noti che il sistema operativo visualizza anche un prompt di login sulla porta seriale. Si possono quindi manifestare strane interazioni tra il sistema operativo e il programma sviluppato nell'esercizio quando entrambi fanno uso della porta seriale.



Figura e9.16 Adattatore seriale da USB a RS-232 DB9 Plugable. (© 2012 Plugable Technologies. Con permesso di riproduzione.)

[3] PuTTY funziona correttamente anche se si omette il `\r`.

```
  while (1) {
    putStrSerial("Please type something: \r\n");
    getStrSerial(str);
    putStrSerial("You typed: ");
    putStrSerial(str);
    putStrSerial("\r\n");
    }
}
```

Comunicare con la linea seriale tra un programma in C e il PC è piuttosto complesso perché le librerie dei driver delle porte seriali non sono standardizzate tra i vari sistemi operativi. Altri ambienti di programmazione come Python, Matlab e LabVIEW rendono la comunicazione seriale priva di problemi.

9.3.5 Timer

I sistemi embedded hanno normalmente la necessità di misurare dei tempi. Per esempio, un forno a microonde ha bisogno di un timer per sapere l'ora del giorno e di un altro timer per misurare i tempi di cottura. Può anche usare un timer per generare impulsi al motore di rotazione del piatto, e un ulteriore timer per controllare la potenza di riscaldamento attivando il generatore di microonde ogni secondo solo per una frazione di tempo.

L'unità di elaborazione grafica e il sistema operativo possono usare i canali 0, 2 e 3, quindi il codice utente deve limitarsi a usare SYSTEM_TIMER_C1.

Il BCM2835 ha un timer di sistema con un contatore a 64 bit a scorrimento libero che viene incrementato di uno ogni microsecondo (quindi a 1 MHz) e quattro canali a 32 bit di confronto del timer. La **Figura e9.17** mostra la mappa di memoria per il timer di sistema. SYS_TIMER_CLO e CHI contengono rispettivamente i 32 bit meno significativi e i 32 bit più significativi del valore a 64 bit del contatore. SYS_TIMER_C0…C3 sono i quattro canali di confronto a 32 bit. Quando uno dei canali di confronto contiene lo stesso valore di SYS_TIMER_CLO, viene portato a 1 il corrispondente bit (M0…M3) nei quattro bit meno significativi di SYS_TIMER_CS. Un bit di confronto viene azzerato scrivendo un 1 in tale bit nel registro SYS_TIMER_CS: può sembrare contro intuitivo, ma serve a evitare di azzerare inavvertitamente altri bit di confronto. Si può dunque misurare un intervallo di tempo dato da un certo numero di microsecondi sommando tale numero a CLO, scrivendo il risultato per esempio in C1, azzerando il bit SYS_TIMER_CS.M1 e aspettando finché il bit SYS_TIMER_CS.M1 viene portato a 1.

Sfortunatamente, Linux è un sistema operativo concorrente che può commutare tra diversi processi senza darne avviso alcuno. Se un programma viene messo in attesa perché aspetta lo scadere del timer e viene attivata l'esecuzione di un altro processo, il programma può essere riattivato anche molto tempo dopo che il timer è scaduto, con conseguente errata valutazione dell'intervallo di tempo trascorso. Per evitare questo problema, il programma in questione deve disabilitare gli interrupt quando deve effettuare misure di tempo critiche, in modo che Linux non possa passare a eseguire un altro processo, ma naturalmente è fondamentale riabilitare gli interrupt appena terminate le misure critiche. EasyPIO definisce le due funzioni `noInterrupts` e `interrupts` rispettivamente per disabilitare e riabilitare gli interrupt. Quando gli interrupt sono disabilitati, il Pi non commuta tra processi diversi e non può quindi neppure rispondere alla pressione dei tasti Ctrl-C per uccidere un programma. Quindi se il programma che ha disabilitato gli interrupt si blocca si deve fisicamente spegnere e riaccendere il Pi per ripartire.

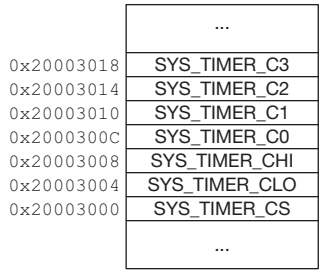


Figura e9.17
Registri del timer di sistema.

ESEMPIO e9.6

LED lampeggiante. Scrivere un programma che fa lampeggiare il LED di stato del Raspberry Pi 5 volte al secondo per 4 secondi.

Soluzione La funzione `delayMicros` di EasyPIO genera un ritardo pari al numero di microsecondi specificato utilizzando il canale di confronto timer numero 1.

```
void delayMicros(int micros) {
 SYS_TIMER_C1 = SYS_TIMER_CLO + micros; // Set the compare register
 SYS_TIMER_CSbits.M1 = 1;               // Reset match flag to 0
 while (SYS_TIMER_CSbits.M1 == 0);      // Wait until match flag is set
}

void delayMillis(int millis) {
 delayMicros(millis*1000);              // 1000 µs per ms
}
```

GPIO47 gestisce il LED di attività del Pi B+. Questo programma definisce il piedino un'uscita e disabilita gli interrupt. Poi accende e spegne il LED con una serie di chiamate a `digitalWrite` ogni 200 ms (5 Hz). Al termine, il programma riabilita gli interrupt.

```
#include "EasyPIO.h"

void main(void) {
 int i;

 pioInit();

 pinMode(47, OUTPUT);     // Status led as output
 noInterrupts();          // Disable interrupts

 for (i=0; i<20; i++) {
  delayMillis(150);
  digitalWrite(47, 0);    // Turn led off
  delayMillis(50);
  digitalWrite(47, 1);    // Turn led on
 }
 interrupts();            // Re-enable interrupts
}
```

9.3.6 I/O analogici

Il mondo esterno è analogico, quindi molti sistemi embedded richiedono ingressi e uscite analogici per interfacciarsi con l'esterno. Usano convertitori analogico-digitali (ADC, *Analog to Digital Converter*) per discretizzare grandezze analogiche in valori digitali, e convertitori digitale-analogici (DAC, *Digital to Analog Converter*) per fare l'operazione contraria. La **Figura e9.18** mostra i simboli circuitali di questi componenti. I convertitori sono caratterizzati dalla risoluzione, dall'intervallo dinamico, dalla frequenza di campionamento e dall'accuratezza. Per esempio, un ADC può avere una risoluzione di $N = 12$ bit nell'intervallo da V_{ref}^- a V_{ref}^+ pari a 0-5 V, con una frequenza di campionamento $f_s = 44$ kHz e un'accuratezza di ±3 bit meno significativi. Le frequenze di campionamento sono spesso espresse in numero di campioni al secondo (sps, *samples per second*) dove 1 sps = 1 Hz. La relazione tra la tensione analogica di ingresso $V_{in}(t)$ e il valore digitale del campione $X[n = t/f_s]$ è data da:

$$X[n] = 2^N \frac{V_{in}(t) - V_{ref-}}{V_{ref+} - V_{ref-}}$$

Per esempio, una tensione di ingresso di 2.5 V (metà scala) corrisponde a un valore digitale $100000000000_2 = 800_{16}$ con un'incertezza massima di 3 bit meno significativi.

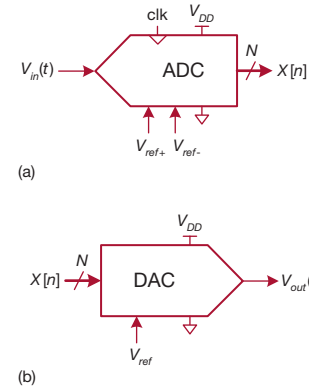


Figura e9.18
Simboli circuitali di ADC e DAC.

Analogamente, un DAC può avere una risoluzione di 16 bit su un intervallo di uscita dato da V_{ref} = 2.56 V. Produce quindi una tensione di uscita data da:

$$V_{out}(t) = \frac{X[n]}{2^N} V_{ref}$$

Molti microcontrollori hanno integrati a bordo dei convertitori ADC di prestazioni moderate; per prestazioni più elevate (per es. con risoluzione di 16 bit a frequenze di campionamento superiori a 1 MHz) è spesso necessario ricorrere ad ADC separati, collegati al microcontrollore. Pochi microcontrollori hanno integrati a bordo dei convertitori DAC, quindi devono essere usati circuiti integrati aggiuntivi. Alcuni microcontrollori generano uscite analogiche con la tecnica denominata PWM (*Pulse Width Modulation*, modulazione di ampiezza impulsi).

9.3.6.1 Conversione D/A

Il BCM 2835 ha un DAC speciale per l'uscita video composita, ma nessun DAC a uso generale. In questo paragrafo si descrive come effettuare conversione D/A (*Digital to Analog*) con DAC esterni e come interfacciare il Raspberry Pi ad altri componenti mediante le porte parallela e seriale. Nel prossimo paragrafo si vedrà come ottenere lo stesso risultato con la modulazione di ampiezza impulsi.

Alcuni DAC ricevono il dato a *N* bit da convertire da un'interfaccia parallela con *N* fili, mentre altri lo ricevono da un'interfaccia seriale come SPI. Alcuni DAC richiedono tensioni di alimentazione sia positive sia negative, mentre altri lavorano su una singola tensione di alimentazione. Alcuni supportano intervalli di tensione modificabili in modo flessibile, altri richiedono una tensione specifica. I livelli logici in ingresso devono naturalmente essere compatibili con la sorgente digitale dei valori da convertire. Alcuni DAC generano una tensione di uscita proporzionale all'ingresso digitale, altri generano una corrente di uscita: in questi casi può servire un amplificatore operazionale per trasformare la corrente in tensione nell'intervallo desiderato.

In questo paragrafo si fa riferimento al DAC parallelo a 8 bit AD558 della Analog Devices, e al DAC seriale a 12 bit LTC1257 della Linear Technology. Entrambi generano in uscita un livello di tensione, lavorano su singola alimentazione a 5-15 V, usano una V_{IH} = 2.4 V quindi sono compatibili con I/O a 3.3 V, si presentano come integrati DIP (*Dual Inline Package*, a due file di piedini) per facilitarne l'uso in schede prototipali e sono di semplice utilizzo. L'integrato AD558 produce una tensione di uscita nell'intervallo 0-2.56 V, consuma 75 mW, è un integrato a 16 piedini e ha un tempo di assestamento di 1 μs che consente una frequenza massima di 1 Mcampioni/sec. La documentazione tecnica si trova sul sito `analog.com`. L'integrato LTC1257 produce una tensione di uscita nell'intervallo 0-2.048 V, consuma meno di 2 mW, è un integrato a 8 piedini e ha un tempo di assestamento di 6 μs. La sua interfaccia SPI opera alla frequenza massina di 1.4 MHz. La documentazione tecnica si trova sul sito `linear.com`.

ESEMPIO e9.7

Uscita analogica con DAC esterni. Tracciare lo schema circuitale e scrivere il programma per un semplice generatore di segnali capace di produrre forme d'onda sinusoidali e triangolari utilizzando il Raspberry Pi, un AD558 e un LTC1257.

Soluzione Lo schema circuitale è mostrato nella **Figura e9.19**. Il componente AD558 si collega al Pi tramite i GPIO14, 15, 17, 18, 22, 23, 24 e 25. I suoi piedini *Vout Sense* e *Vout Select* sono collegati a *Vout* per usare l'intervallo completo di tensioni di uscita 0-2.56 V. Il componente LTC1257 si collega al Pi tramite SPI0. Entrambi i DAC

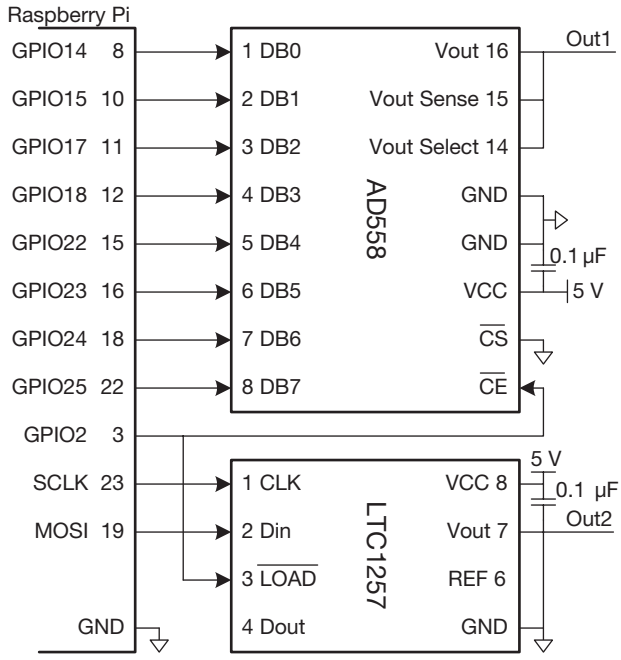


Figura e9.19
Interfacciamento del Raspberry Pi a DAC parallelo e seriale.

usano l'alimentazione a 5 V e hanno un condensatore di disaccoppiamento da 1 µF per ridurre il rumore di alimentazione. I segnali di abilitazione (CE, *Chip Enable*) e di caricamento (LOAD) attivi bassi dei due DAC indicano quando convertire il valore digitale, che viene caricato nei DAC tenendo tali segnali al livello logico 1.

Il programma è riportato sotto. Le funzioni `pinsMode` e `digitalWrites` sono simili a `pinMode` e `digitalWrite`, ma operano su un vettore di contatti. Il programma definisce uscite gli 8 contatti della porta parallela e configura anche GPIO2 come uscita per poter pilotare i segnali di abilitazione e caricamento. Inizializza inoltre SPI0 a 1.4 MHz. La funzione `initWaveTables` predispone un vettore di valori corrispondenti a campioni di una forma d'onda sinusoidale e di una triangolare: la funzione sinusoidale è predisposta su una scala a 12 bit, quella triangolare su una scala a 8 bit, con 64 campioni nel periodo di ciascuna delle due forme d'onda. Modificando quest'ultimo valore si può aumentare la precisione a scapito della frequenza o viceversa. La funzione `genWaves` opera ciclicamente sui campioni. Disabilita gli interrupt per evitare commutazioni di processo e alterazioni delle forme d'onda. Per ogni campione, disabilita i segnali CE e LOAD dei due DAC, invia il nuovo valore tramite le porte parallela e seriale, riabilita i DAC, quindi attende finché il timer indica che è ora di passare al campione successivo. La funzione `spiSendReceive16` trasmette due byte, ma il componente LTC1257 utilizza solo gli ultimi 12 bit inviati. La frequenza massima delle forme d'onda di poco superiore a 1000 Hz (64 Kcampioni/sec.) è dovuta al tempo necessario per spedire ogni campione nella funzione `genWaves`, tempo in gran parte dovuto alla trasmissione SPI.

```
#include "EasyPIO.h"
#include math.h> // required to use the sine function

#define NUMPTS 64
int sine[NUMPTS], triangle[NUMPTS];
int parallelPins[8] = {14,15,17,18,22,23,24,25};
void initWaveTables(void) {
  int i;
  for (i=0; i<NUMPTS; i++) {
    sine[i] = 2047*(sin(2*3.14159*i/NUMPTS) + 1); // 12-bit scale
    if (i<NUMPTS/2) triangle[i] = i*511/NUMPTS;  // 8-bit scale
    else triangle[i] = 510-i*511/NUMPTS;
  }
}
```

```
void genWaves(int freq) {
  int i, j;
  int microPeriod = 1000000/(NUMPTS*freq);

  noInterrupts(); // disable interrupts to get accurate timing
  for (i=0; i<2000; i++){
    for (j=0; j<NUMPTS; j++) {
      SYS_TIMER_C1 = SYS_TIMER_CLO + microPeriod; // Set time between samples
      SYS_TIMER_CSbits.M1 = 1;                    // Clear timer match
      digitalWrite(2,1);                          // No load while changing inputs
      spiSendReceive16(sine[j]);
      digitalWrites(parallelPins, 8, triangle[j]);
      digitalWrite(2,0);                          // Load new points into DACs
      while (!SYS_TIMER_CSbits.M1);               // Wait until timer matches
    }
  }
  interrupts();
}

void main(void) {
  pioInit();

  pinsMode(parallelPins, 8, OUTPUT); // Set pins connected to the AD558 as outputs
  pinMode(2, OUTPUT);                // Make pin 2 an output to control LOAD and CE
  spiInit(1400000, 0);               // 1.4MHz SPI clock, default settings
  initWaveTables();
  genWaves(1000);
}
```

9.3.6.2 Modulazione di ampiezza impulsi

Un altro modo che può essere usato da un sistema digitale per generare un'uscita analogica è la **modulazione di ampiezza impulsi** (PWM, *Pulse-Width Modulation*), che consiste in un segnale periodico tenuto alto per parte del periodo e basso per la parte rimanente. Il cosiddetto ***duty cycle*** del segnale è la frazione del periodo per la quale il segnale è tenuto alto, come mostrato nella **Figura e9.20**. Il valor medio dell'uscita è quindi proporzionale al *duty cycle*. Per esempio, se il segnale commuta tra 0 e 3.3 V e ha un duty cycle del 25%, il valor medio risulta $0.25 \times 3.3 = 0.825$ V. Un filtro passa-basso del segnale PWM elimina le oscillazioni e produce un segnale continuo del valore desiderato. La tecnica PWM è quindi un metodo molto valido per produrre un'uscita analogica quando la frequenza degli impulsi è molto maggiore delle frequenze di interesse per il segnale analogico prodotto.

Il BCM2835 ha un controllore PWM in grado di generare simultaneamente due uscite. PWM0 è disponibile al contatto GPIO18 come funzione ALT5, mentre entrambe le uscite PWM sono disponibili sulla presa per uno spinotto audio stereo. La **Figura e9.21** mostra la mappa di memoria per l'unità PWM e per il gestore di clock da cui essa dipende.

Il registro PWM_CTL è usato per attivare la modulazione di ampiezza impulsi. Il bit 0 (PWEN1) deve essere posto a 1 per attivare l'uscita. Il bit 7 (MSEN1, *Mark-Space ENable*) deve anch'esso essere posto a 1 per produrre

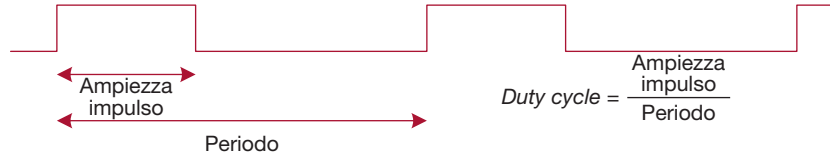


Figura e9.20
Segnale a modulazione di ampiezza impulsi (PWM).

la modulazione riportata nella Figura e9.20, nella quale l'uscita è ALTA per una parte del periodo e BASSA per la parte rimanente. I segnali PWM vengono derivati da un clock PWM generato dal gestore di clock del BCM2835. I registri PWM_RNG1 e PWM_DAT1 sono usati per controllare il periodo e il duty cycle, specificando rispettivamente il numero di colpi del clock PWM per l'intero periodo e di quelli per la parte ALTA. Per esempio, se il gestore di clock produce un clock a 25 MHz e i registri valgono PWM_RNG1 = 1000 e PWM_DAT1 = 300, l'uscita PWM lavora a (25 MHz/1000) = 25 kHz di frequenza, con un duty cycle di 300/1000 = 30%.

Il gestore di clock viene configurato tramite il registro CM_PWMCTL e la frequenza del clock PWM viene impostata tramite il registro CM_PWMDIV. La **Tabella e9.7** riassume i campi di bit del registro CM_PWMCTL. La massima frequenza del clock PWM è di 25 MHz, e può essere ottenuta dal clock PLLD a 500 MHz del Pi nel modo seguente:

- CM_PWMCTL: Scrivere 0x5A in PASSWD e 1 in KILL per fermare il generatore di clock.
- CM_PWMCLT: Attendere che BUSY diventi 0 a indicare che il clock si è fermato.
- CM_PWMCTL: Scrivere 0x5A in PASSWD, 1 in MASH e 6 in SRC per selezionare PLLD senza *noise shaping* audio.
- CM_PWMDIV: Scrivere 0x5A in PASSWD e 20 nei bit 23:12 per dividere per 20 il clock PLLD portandolo da 500 MHz a 25 MHz.
- CM_PWMCTL: Scrivere 0x5A in PASSWD e 1 in ENAB per far ripartire il generatore di clock.
- CM_PWMCLT: Attendere che BUSY diventi 1 a indicare che il clock è partito.

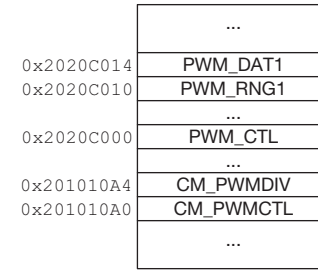


Figura e9.21
Registri del PWM e del gestore di clock.

I registri CM_PWM non sono documentati nella documentazione tecnica del BCM2835. Le informazioni si trovano con una ricerca su Internet del documento "BCM2835 Audio & PWM Clocks" di G.J. van Loo.

Tabella e9.7 Campi di bit del registro CM_PWMCTL.

Bit	Nome	Descrizione
31:24	PASSWD	Da impostare a 5A per scrivere
10:9	MASH	*Noise shaping* audio
7	BUSY	Generatore di clock operativo
5	KILL	Scrivere 1 per fermare il generatore di clock
4	ENAB	Scrivere 1 per far partire il generatore di clock
3:0	SRC	Sorgente del clock

ESEMPIO e9.8

Uscita analogica con PWM. Scrivere la funzione `analogWrite(val)` per generare una tensione analogica in uscita usando PWM e un filtro RC esterno. La funzione deve accettare come parametro di ingresso un valore compreso tra 0 (corrispondente a 0 V in uscita) e 255 (corrispondente al valore massimo 3.3 V in uscita).

Soluzione Si può usare PWM0 per generare un segnale a 78.125 kHz su GPIO18. Il filtro passa-basso della **Figura e9.22** ha una frequenza di taglio (*corner frequency*) pari a

$$f_c = \frac{1}{2\pi RC} = 1.6 \text{ kHz}$$

per eliminare le oscillazioni ad alta frequenza e passare in uscita il valor medio.

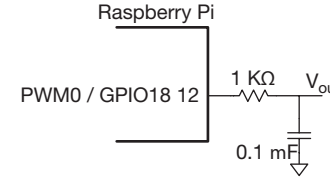


Figura e9.22
Uscita analogica usando PWM e un filtro passa-basso.

Le funzioni PWM presenti in EasyPIO sono riportate sotto. `pwmInit` inizializza il modulo PWM su GPIO18 come descritto in precedenza. `setPWM` imposta la frequenza e il *duty cycle* dell'uscita PWM; il duty cycle è espresso da un valore compreso tra 0 (forma d'onda sempre bassa) e 1 (sempre alta). La funzione `analogWrite` imposta il duty cycle nella scala di valori da 0 a 255.

```
// Default PLLD value is 500 [MHz]
#define PLL_FREQUENCY 500000000
// Max pwm clk is 25 [MHz]
#define CM_FREQUENCY 25000000
#define PLL_CLOCK_DIVISOR (PLL_FREQUENCY / CM_FREQUENCY)

void pwmInit() {
 pinMode(18, ALT5);

 // Configure the clock manager to generate a 25 MHz PWM clock.
 // Documentation on the clock manager is missing in the datasheet
 // but found in "BCM2835 Audio and PWM Clocks" by G.J. van Loo 6 Feb 2013.
 // Maximum operating frequency of PWM clock is 25 MHz.
 // Writes to the clock manager registers require simultaneous writing
 // a "password" of 5A to the top bits to reduce the risk of accidental
    writes.

 CM_PWMCTL = 0; // Turn off PWM before changing
 CM_PWMCTL = PWM_CLK_PASSWORD|0x20; // Turn off clock generator
 while (CM_PWMCTLbits.BUSY); // Wait for generator to stop
 CM_PWMCTL = PWM_CLK_PASSWORD|0x206; // Src = unfiltered 500 MHz CLKD
 CM_PWMDIV = PWM_CLK_PASSWORD|(PLL_CLOCK_DIVISOR << 12); // 25 MHz
 CM_PWMCTL = CM_PWMCTL|PWM_CLK_PASSWORD|0x10; // Enable PWM clock
 while (!CM_PWMCTLbits.BUSY); // Wait for generator to start
 PWM_CTLbits.MSEN1 = 1; // Channel 1 in mark/space mode
 PWM_CTLbits.PWEN1 = 1; // Enable PWM
}

void setPWM(float freq, float duty) {
 PWM_RNG1 = (int)(CM_FREQUENCY / freq);
 PWM_DAT1 = (int)(duty * (CM_FREQUENCY / freq));
}

void analogWrite(int val) {
 setPWM(78125, val/255.0);
}
```

La funzione `main` prova la conversione PWM impostando l'uscita a metà scala (1.65 V).

```
#include "EasyPIO.h"

void main(void) {
 pioInit();
 pwmInit();
 analogWrite(128);
}
```

9.3.6.3 Conversione A/D

Il BCM 2835 non ha nessun ADC integrato. In questo paragrafo si descrive some effettuare conversione A/D (*Analog to Digital*) con un convertitore esterno simile al DAC usato in precedenza.

ESEMPIO e9.9

Ingresso analogico con ADC esterno. Interfacciare il convertitore A/D a 10 bit MCP3002 al Raspberry Pi usando SPI, e visualizzare il valore acquisito. Impostare un fondo scala di 3.3 V. Per tutti i dettagli operativi, cercare su Web le informazioni tecniche.

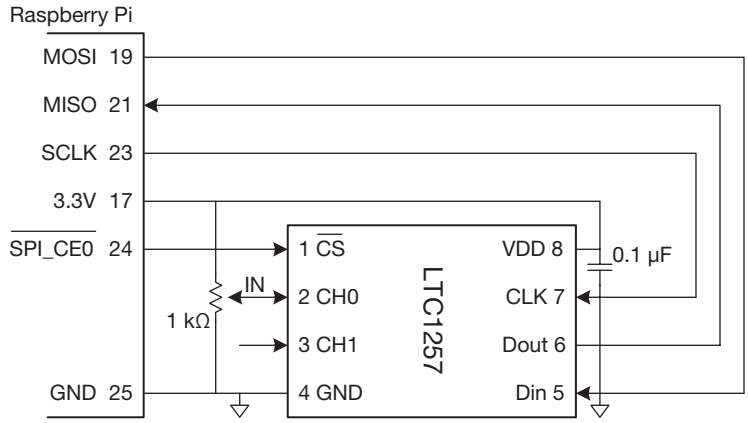


Figura e9.23
Ingresso analogico mediante ADC esterno.

Soluzione La **Figura e9.23** mostra uno schema circuitale del collegamento. Il componente MPC3002 usa VDD come tensione di fondo scala di riferimento. Può essere alimentato con tensione compresa nell'intervallo 3.3-5.5 V e si è scelta in questa caso la tensione 3.3 V. L'ADC ha due canali di ingresso, e si è collegato un potenziometro al canale 0 per poter variare la tensione da convertire tra 0 V e 3.3 V.

Il codice per il Pi inizializza SPI, poi ripetutamente legge il valore campionato e lo visualizza. Come da documentazione tecnica, il Raspberry Pi deve inviare il valore a 16 bit 0x6000 sul collegamento SPI per leggere il canale CH0 dell'ADC, ricevendo i 10 bit del risultato nei 10 bit meno significativi del dato a 16 bit restituito da SPI. Il convertitore richiede anche un segnale di selezione (*chip select*) fornito tramite il segnale di abilitazione di SPI.

```
#include "EasyPIO.h"

void main(void) {
 int sample;

 pioInit();
 spiInit(200000, 0); // 200 kHz SPI clock, default settings

 while (1){
  sample = spiSendReceive16(0x6000);
  printf("Read %d\n", sample);
 }
}
```

9.3.7 Interrupt

Sinora ci si è affidati alla tecnica del ***polling***, nella quale il programma in esecuzione continua a controllare se si è verificato un evento come l'arrivo di un dato da UART, o il raggiungimento del valore impostato per un timer. Questo approccio può sprecare molta potenza di calcolo del processore, e impedisce di scrivere programmi che fanno cose utili in attesa del verificarsi di eventi esterni.

Praticamente tutti i microcontrollori supportano la modalità di lavoro a **interrupt**: quando un evento si verifica, il microcontrollore interrompe l'esecuzione dei normali programmi e passa a eseguire un gestore di interrupt che gestisce l'evento e poi riprende il lavoro interrotto.

Il Raspberry Pi normalmente lavora con Linux, che intercetta gli interrupt prima che arrivino al programma in esecuzione. Non è quindi semplice scrivere programmi che lavorano a interrupt, e in questo libro non si forniscono esempi per il Pi.

9.4 ■ ALTRE PERIFERICHE DI MICROCONTROLLORI

I microcontrollori si interfacciano spesso con altre periferiche. In questo paragrafo si descrivono alcuni esempi molto comuni: un display di caratteri a cristalli liquidi, un monitor VGA, un collegamento Bluetooth e il controllo di un motore. Le interfacce di comunicazione standard USB ed Ethernet sono descritte nei paragrafi 9.6.1 e 9.6.4.

9.4.1 Display di caratteri a cristalli liquidi

I display di caratteri a cristalli liquidi (LCD, *Liquid Crystal Display*) sono dispositivi in grado di visualizzare una o più linee di testo; sono usati generalmente nei pannelli frontali di apparecchiature come registratori di cassa, stampanti laser, apparecchi fax, che devono visualizzare una quantità limitata di informazioni; sono facilmente interfacciabili a un microcontrollore tramite le interfacce parallele, RS-232 o SPI. La Crystalfontz America produce un'ampia gamma di questi display, con dimensioni da 8 colonne × 1 riga a 40 colonne × 4 righe, con varie opzioni di colore, di retro illuminazione, di tensione di alimentazione di 3.3 V o 5 V e di visibilità con luce diurna. I prezzi si aggirano sui 20 dollari e oltre per acquisti di pochi pezzi, ma per acquisti di grosse quantità possono scendere a meno di 5 dollari.

In questo paragrafo si fornisce un esempio di interfacciamento dal Raspberry Pi con uno di questi display mediante la porta parallela a 8 bit. L'interfaccia è compatibile con il controllore di LCD standard industriale HD44780 sviluppato originariamente dalla Hitachi. La **Figura e9.24** mostra il display LCD parallelo Crystalfontz CFAH2002A-TMI-JT 20 × 2.

La **Figura e9.25** mostra il display LCD collegato al Pi mediante l'interfaccia parallela a 8 bit. La logica lavora a 5 V ma è compatibile con gli ingressi a 3.3 V provenienti dal Pi. Il contrasto del display viene regolato da una seconda tensione prodotta mediante un potenziometro: la leggibilità migliore si ha solitamente per valori compresi tra 4.2 V e 4.8 V. Il display LCD riceve tre segnali di controllo: RS (1 per caratteri, 0 per istruzioni), R/W (1 per leggere dal display, 0 per scrivere) ed E (da portare a 1 per almeno 250 ns per abilitare il display quando è pronto il prossimo byte da leggere). Quando il display legge un'istruzione, il bit 7 restituisce la flag di disponibilità, che vale 1 se il display è occupato e 0 se è pronto a ricevere un'altra istruzione.

Per inizializzare il display, il Pi deve scrivere la sequenza di istruzioni riportata nella **Tabella e9.8**. Le istruzioni vengono scritte tenendo R = 0 e R/W = 0, inserendo il loro valore sulle otto linee di dato e dando un impulso a E. Dopo ogni istruzione, il Pi deve attendere un certo tempo (oppure finché viene portata a 0 la flag di disponibilità).

Ora, per scrivere testo, il Pi può inviare al display una sequenza di caratteri ASCII. Dopo ogni carattere, deve attendere che la flag di disponibilità diventi 0. Può anche inviare l'istruzione 0x01 per cancellare il display, o l'istruzione 0x02 per tornare con il cursore nella posizione iniziale in alto a sinistra.



Figura e9.24 Display di caratteri LCD Crystalfontz CFAH2002A-TMI 20 × 2.
(© 2012 Crystalfontz America. Con permesso di riproduzione)

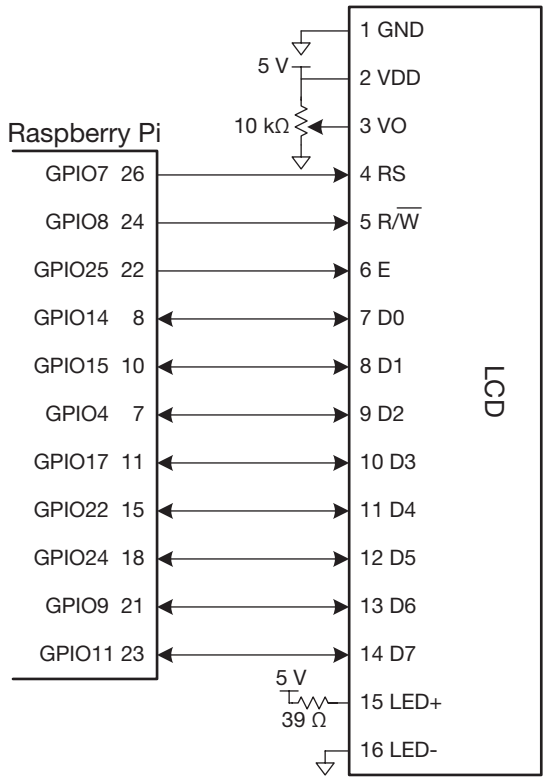


Figura e9.25 **Interfaccia parallela a display LCD.**

Tabella e9.8 **Sequenza di inizializzazione del display LCD.**

Scrivere	Scopo	Attendere (µs)
(applicare V_{DD})	Accendere il dispositivo	15000
0x30	Impostare la modalità a 8 bit	4100
0x30	Impostare di nuovo la modalità a 8 bit	100
0x30	Impostare ancora una volta la modalità a 8 bit	Fino a quando la flag di disponibilità diventa 0
0x3C	Impostare 2 righe e formato caratteri 5×8	Fino a quando la flag di disponibilità diventa 0
0x08	Spegnere il display	Fino a quando la flag di disponibilità diventa 0
0x01	Cancellare il display	1530
0x06	Impostare il modo inserimento a incremento del cursore dopo ogni carattere	Fino a quando la flag di disponibilità diventa 0
0x0C	Accendere il display senza cursore	

ESEMPIO e9.10

Controllo del display LCD. Scrivere un programma per visualizzare la scritta "I love LCDs" al display.

Soluzione Il seguente programma scrive sul display "I love LCDs" inviandogli la sequenza di caratteri dopo averlo inizializzato.

```
#include "EasyPIO.h"

int LCD_IO_Pins[] = {14, 15, 4, 17, 22, 24, 9, 11};

typedef enum {INSTR, DATA} mode;
#define RS 7
```

```
#define RW 8
#define E 25

char lcdRead(mode md) {
 char c;
 pinsMode(LCD_IO_Pins, 8, INPUT);
 digitalWrite(RS,(md == DATA));       // Set instr/data mode
 digitalWrite(RW, 1);                 // Read mode
 digitalWrite(E, 1);                  // Pulse enable
 delayMicros(10);                     // Wait for LCD response
 c = digitalReads(LCD_IO_Pins, 8);    // Read a byte from parallel port
 digitalWrite(E, 0);                  // Turn off enable
 delayMicros(10);
 return c;
}

void lcdBusyWait(void) {
 char state;
 do {
  state = lcdRead(INSTR);
 } while (state & 0x80);
}

void lcdWrite(char val, mode md) {
 pinsMode(LCD_IO_Pins, 8, OUTPUT);
 digitalWrite(RS, (md == DATA));      // Set instr/data mode. OUTPUT=1, INPUT=0
 digitalWrite(RW, 0);                 // Set RW pin to write (aka: 0)
 digitalWrites(LCD_IO_Pins, 8, val);  // Write the char to the parallel port
 digitalWrite(E, 1); delayMicros(10); // Pulse E
 digitalWrite(E, 0); delayMicros(10);
}

void lcdClear(void) {
 lcdWrite(0x01, INSTR); delayMicros(1530);
}

void lcdPrintString(char* str) {
 while (*str != 0) {
  lcdWrite(*str, DATA); lcdBusyWait();
  str++;
 }
}

void lcdInit(void) {
 pinMode(RS, OUTPUT); pinMode(RW, OUTPUT); pinMode(E,OUTPUT);
 // send initialization routine:
 delayMicros(15000);
 lcdWrite(0x30, INSTR); delayMicros(4100);
 lcdWrite(0x30, INSTR); delayMicros(100);
 lcdWrite(0x30, INSTR); lcdBusyWait();
 lcdWrite(0x3C, INSTR); lcdBusyWait();
 lcdWrite(0x08, INSTR); lcdBusyWait();
 lcdClear();
 lcdWrite(0x06, INSTR); lcdBusyWait();
 lcdWrite(0x0C, INSTR); lcdBusyWait();
}

void main(void) {
 pioInit();
 lcdInit();
 lcdPrintString("I love LCDs!");
}
```

9.4.2 Monitor VGA

Un'opzione più flessibile riguardo alla visualizzazione di informazioni è quella di utilizzare un monitor come quello dei PC. Il Raspberry Pi ha integrati i supporti per le uscite video HDMI e composita. Vedremo ora in dettaglio come pilotare un monitor VGA direttamente tramite un componente FPGA.

Lo standard dei monitor VGA (*Video Graphic Array*) è stato introdotto nel 1987 dalla IBM per i computer della serie PS/2, con una risoluzione di 640×480 pixel su un monitor realizzato con un tubo a raggi catodici (CRT, *Cathode Ray Tube*) collegato tramite un connettore a 15 contatti per l'invio delle informazioni cromatiche con tensioni analogiche. I moderni monitor LCD hanno risoluzioni superiori ma mantengono retrocompatibilità con lo standard VGA.

In un tubo a raggi catodici un pennello elettronico scandisce lo schermo da sinistra verso destra eccitando del materiale fluorescente (fosforo) per visualizzare un'immagine. I CRT a colori usano tre tipi di fosfori per il rosso, il verde e il blu (da cui la sigla RGB, *Red Green Blue*) e hanno tre pennelli elettronici. L'intensità del pennello elettronico determina il livello di ciascun colore nel pixel. Al termine di una linea orizzontale di scansione, il pennello elettronico deve spegnersi per un intervallo di oscuramento orizzontale, per tornare all'inizio della linea successiva. Quando tutte le linee di scansione sono terminate, il pennello deve di nuovo spegnersi per un intervallo di oscuramento verticale per tornare nell'angolo in alto a sinistra del video. Il procedimento si ripete 60-75 volte al secondo per rinfrescare la fluorescenza e dare l'illusione visiva di un'immagine stabile. Un display a cristalli liquidi non ha bisogno della scansione del pennello elettronico, ma mantiene la stessa temporizzazione dell'interfaccia VGA per compatibilità.

In un monitor VGA 640×480 rinfrescato a 59.94 kHz, il clock di pixel lavora a 25.175 MHz, quindi ogni pixel dura 39.72 ns. L'intero schermo è diviso in 525 linee di scansione orizzontale ciascuna di 800 pixel, ma solo 480 linee di scansione da 640 pixel ciascuna portano informazione, il resto delle linee e dei pixel è sempre nero. Una linea di scansione inizia con il *back porch* (letteralmente "veranda posteriore"), ovvero la sezione vuota sul lato sinistro dello schermo. Poi contiene i 640 pixel, seguiti dal *front porch* ("veranda anteriore") sul lato destro dello schermo e termina con un impulso di sincronizzazione orizzontale (*hsync*) per riportare rapidamente il pennello elettronico sul lato sinistro.

La **Figura e9.26(a)** mostra la temporizzazione di ciascuna di queste sezioni della linea di scansione, a partire dai 640 pixel attivi. L'intera linea ha

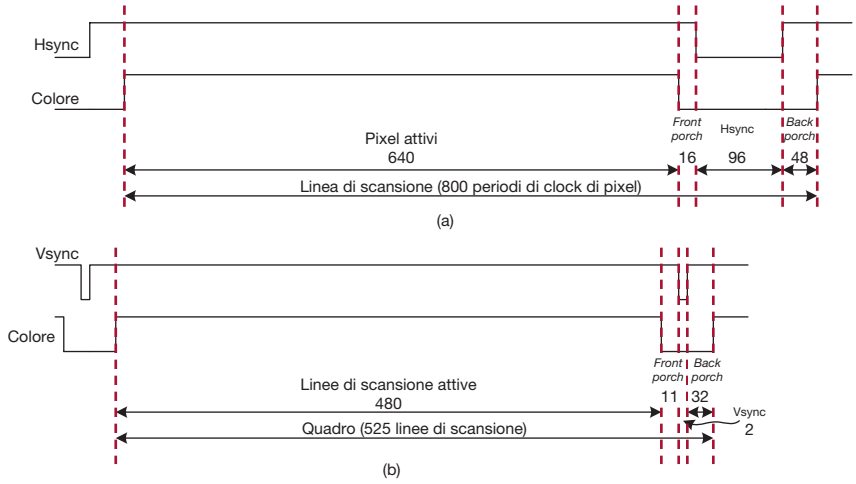
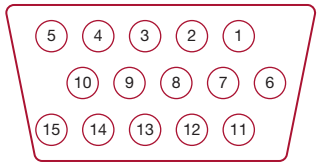


Figura e9.26 Temporizzazioni VGA: (a) orizzontale, (b) verticale.



1: Red
2: Green
3: Blue
4: Reserved
5: GND
6: GND
7: GND
8: GND
9: 5 V (optional)
10: GND
11: Reserved
12: I^2C data
13: Hsync
14: Vsync
15: I^2C clock

Figura e9.27
Contatti del connettore VGA.

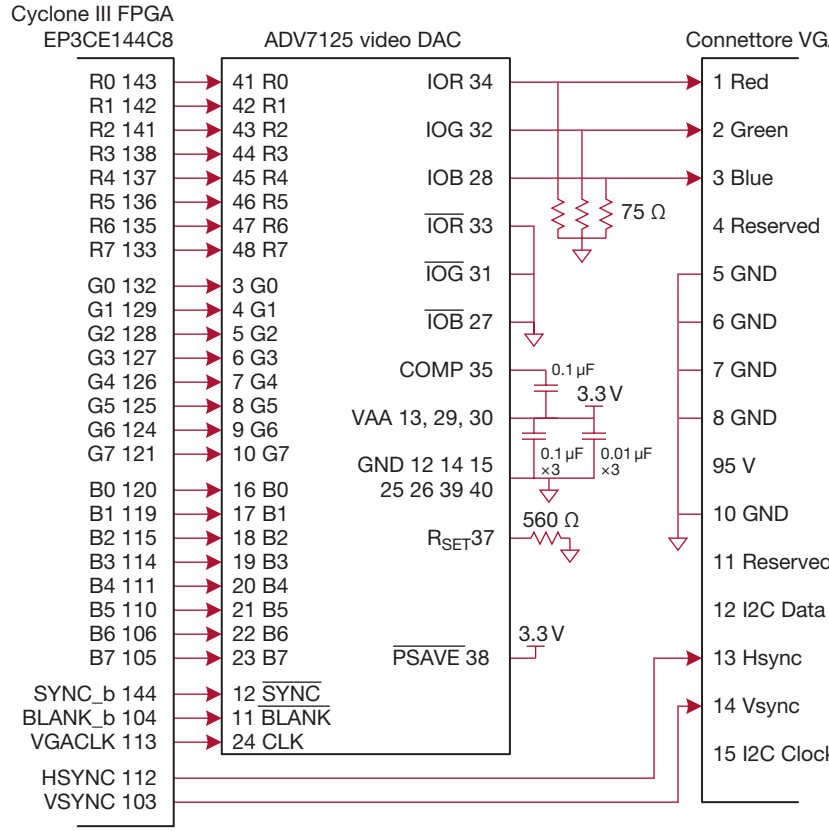
una durata di 31.778 µs. In direzione verticale, lo schermo parte con un back porch in alto, seguito dalle 480 linee attive, quindi da un front porch in basso e da un impulso di sincronizzazione verticale (*vsync*) per ritornare in alto. Un nuovo quadro (*frame*) viene disegnato 60 volte al secondo.

La **Figura e9.26(b)** mostra la temporizzazione verticale; si noti che le unità di tempo sono ora linee di scansione invece di periodi del clock di pixel. Risoluzioni superiori usano clock di pixel più veloci, fino a 388 MHz per rinfrescare un video 2048×1536 a 85 Hz. Per esempio, un video 1024×768 a 60 Hz richiede un clock di pixel a 65 MHz.

La temporizzazione orizzontale comprende il front porch di 16 periodi di clock, l'impulso hsync di 96 periodi di clock e il back porch di 48 periodi di clock. La temporizzazione verticale comprende il front porch di 11 linee di scansione, l'impulso vsync di 2 linee e il back porch di 32 linee.

La **Figura e9.27** mostra la disposizione dei contatti per un connettore femmina proveniente dalla sorgente video. L'informazione di pixel è codificata da tre tensioni analogiche per le componenti rossa, verde e blu. Ogni tensione varia nell'intervallo 0-0.7 V, dove valori più positivi corrispondono a maggiore intensità luminosa. Le tensioni devono essere 0 durante i front e back porch. Il segnale video deve essere generato in tempo reale ad alta velocità, cosa molto difficile per un microcontrollore ma facile per un componente FPGA. Un semplice monitor in bianco e nero può essere realizzato pilotando tutti e tre i contatti dei colori solo con 0 V oppure 0.7 V mediante un divisore di tensione collegato a un'uscita digitale. Un monitor a colori richiede invece un DAC video con tre convertitori D/A separati per pilotare in modo indipendente i tre contatti dei colori. La **Figura e9.28** mostra un componente FPGA che pilota un monitor VGA mediante un triplo DAC video a 8 bit ADV7125. Il DAC riceve dal componente FPGA 8 bit per ciascuno dei tre colori R, G e

Figura e9.28
FPGA di pilotaggio di un monitor VGA tramite un DAC video.



B e un segnale di sincronismo SYNC_b portato a 0 quando vengono attivati HSYNC oppure VSYNC. Il DAC video produce tre correnti di uscita per pilotare le linee analogiche dei tre colori, che normalmente sono linee di trasmissione a 75 Ω terminate in parallelo da entrambi i lati. La resistenza R_{SET} imposta la scala delle correnti di uscita per ottenere l'intera gamma di colori. La frequenza di clock dipende dalla risoluzione e dalla frequenza di rinfresco: può arrivare a 330 MHz se si adotta il modello di DAC di categoria veloce ADV7125JSTZ330.

ESEMPIO e9.11

Monitor VGA. Scrivere il codice HDL per visualizzare del testo e un riquadro verde su un monitor VGA usando la circuiteria della Figura e9.28.

Soluzione Il codice ipotizza una frequenza del clock di sistema di 40 MHz e usa un PLL (*Phase Locked Loop*) sul componente FPGA per generare i 25.175 MHz del clock VGA. La configurazione del PLL varia nei diversi componenti FPGA: nel caso di Cyclone III, le frequenze sono impostate tramite il *wizard* Altera megafunction. Altrimenti si può fornire il clock VGA direttamente con un generatore di segnale.

Il controllore VGA conta righe e colonne dello schermo, generando i segnali hsync e vsync al momento opportuno e il segnale blank_b portato a 0 per produrre schermo nero fuori della zona attiva di 640×480 pixel.

Il generatore video produce i valori di rosso, verde e blu in base alla posizione (x,y) del pixel corrente, dove (0,0) indica l'angolo in alto a sinistra. Visualizza una serie di caratteri e un rettangolo verde. Il generatore di caratteri disegna caratteri di 8×8 pixel, quindi ha una dimensione video di 80×60 caratteri. La rappresentazione del carattere viene prelevata da una ROM, dove è stata codificata in binario come 6 colonne per 8 righe (con due colonne spente per separare i caratteri di una stessa parola). L'ordine dei bit viene rovesciato nel codice SystemVerilog perché la colonna più a sinistra nella ROM è il bit più significativo, che va tracciato su video nella posizione x meno significativa.

La **Figura e9.29** mostra una fotografia del monitor VGA durante l'esecuzione del programma. Le righe delle lettere sono alternativamente rosse e blu, e il rettangolo verde si sovrappone a una parte dell'immagine.

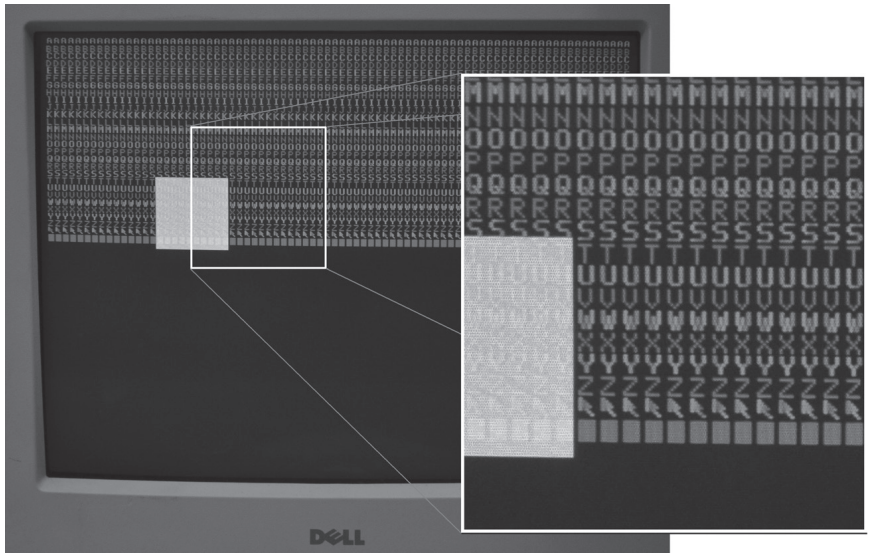


Figura e9.29
Uscita VGA.

vga.sv

```
module vga(input logic  clk,
  output logic  vgaclk,          // 25.175 MHz VGA clock
  output logic  hsync, vsync,
  output logic  sync_b, blank_b, // To monitor & DAC
  output logic [7:0] r, g, b);   // To video DAC

 logic [9:0] x, y;

 // Use a PLL to create the 25.175 MHz VGA pixel clock
 // 25.175 MHz clk period = 39.772 ns
 // Screen is 800 clocks wide by 525 tall, but only 640 x 480 used
 // HSync = 1/(39.772 ns *800) = 31.470 kHz
 // Vsync = 31.474 kHz / 525 = 59.94 Hz (~60 Hz refresh rate)
 pll vgapll(.inclk0(clk), .c0(vgaclk));

 // Generate monitor timing signals
 vgaController vgaCont(vgaclk, hsync, vsync, sync_b, blank_b, x, y);

 // User-defined module to determine pixel color
 videoGen videoGen(x, y, r, g, b);
endmodule

module vgaController #(parameter HACTIVE = 10'd640,
                                 HFP     = 10'd16,
                                 HSYN    = 10'd96,
                                 HBP     = 10'd48,
                                 HMAX    = HACTIVE+HFP+HSYN+HBP,
                                 VBP     = 10'd32,
                                 VACTIVE = 10'd480,
                                 VFP     = 10'd11,
                                 VSYN    = 10'd2,
                                 VMAX    = VACTIVE+VFP+VSYN+VBP)
                     (input logic       vgaclk,
                     output logic       hsync, vsync, sync_b, blank_b,
                     output logic [9:0] x, y);
 // counters for horizontal and vertical positions
 always @(posedge vgaclk) begin
  x++;
  if (x==HMAX) begin
   x = 0;
   y++;
   if (y==VMAX) y = 0;
  end
 end

 // Compute sync signals (active low)
 assign hsync = ~(x >= HACTIVE + HFP & x < HACTIVE + HFP + HSYN);
 assign vsync = ~(y >= VACTIVE + VFP & y < VACTIVE + VFP + VSYN);
 assign sync_b = hsync & vsync;
 // Force outputs to black when outside the legal display area
 assign blank_b = (x <HACTIVE) & (y < VACTIVE);
endmodule

module videoGen(inputlogic [9:0] x, y, output logic [7:0] r, g, b);
 logic  pixel, inrect;

 // Given y position, choose a character to display
 // then look up the pixel value from the character ROM
 // and display it in red or blue.Also draw a green rectangle.
 chargenrom chargenromb(y[8:3]+8'd65, x[2:0], y[2:0], pixel);
 rectgen rectgen(x, y, 10'd120, 10'd150, 10'd200, 10'd230, inrect);
 assign {r, b} = (y[3]==0) ? {{8{pixel}},8'h00} : {8'h00,{8{pixel}}};
```

```
 assign g = inrect ? 8'hFF : 8'h00;
endmodule

module chargenrom(input logic [7:0] ch,
                  input logic [2:0] xoff, yoff,
                  output logic  pixel);

 logic [5:0] charrom[2047:0]; // character generator ROM
 logic [7:0] line;            // a line read from the ROM
 // Initialize ROM with characters from text file
 initial
  readmemb("charrom.txt", charrom);

 // Index into ROM to find line of character
 assign line = charrom[yoff+{ch-65, 3'b000}]; // Subtract 65 because A
                                              // is entry 0

 // Reverse order of bits
 assign pixel = line[3'd7-xoff];
endmodule

module rectgen(input logic [9:0] x, y, left, top, right, bot,
               output logic      inrect);

 assign inrect = (x > = left & x < right & y >= top & y < bot);
endmodule
```

charrom.txt

```
// A ASCII 65
011100
100010
100010
111110
100010
100010
100010
000000
//B ASCII 66
111100
100010
100010
111100
100010
100010
111100
000000
//C ASCII 67
011100
100010
100000
100000
100000
100010
011100
000000
...
```

9.4.3 Comunicazioni wireless Bluetooth

Sono oggi disponibili vari standard per le comunicazioni wireless (senza fili) tra i quali Wi-Fi, Zigbee e Bluetooth. Gli standard sono complessi e necessitano di circuiti integrati sofisticati, ma sono sempre più diffusi moduli che

Il nome Bluetooth deriva dal re di Danimarca Harald Bluetooth, che regnò nel X secolo e unificò le tribù Danesi in guerra tra di loro. Il protocollo Bluetooth non ha avuto lo stesso successo nell'unificare tutti i protocolli di comunicazione wireless oggi diffusi…

risolvono al proprio interno tali complessità e danno all'utente una semplice interfaccia per le comunicazioni wireless. Uno di questi moduli è BlueSMiRF: un'interfaccia Bluetooth facile da utilizzare per comunicazioni senza necessità di cavi seriali.

Bluetooth è uno standard wireless inizialmente sviluppato dalla Ericsson nel 1994 per comunicazioni a bassa potenza e limitata velocità su distanze di 5-100 metri, a seconda della potenza del trasmettitore. Viene comunemente usato per collegare un auricolare a un telefono cellulare o una tastiera a un PC. A differenza delle comunicazioni a raggi infrarossi, non richiede che i dispositivi che devono comunicare siano visibili uno dall'altro.

Bluetooth lavora nella banda senza autorizzazione a 2.4 GHz denominata ISM (*Industrial Scientific Medical*). Definisce 79 canali radio distanziati a intervalli di 1 MHz partendo dalla frequenza di 2402 MHz, e salta tra questi canali secondo una sequenza pseudo-casuale per evitare eccessive interferenze con altri dispositivi come i router di rete wireless che operano nella stessa banda. Come specificato nella **Tabella e9.9**, i trasmettitori Bluetooth sono classificati in tre livelli di potenza, che definiscono il consumo di potenza e la distanza massima della comunicazione. Nella modalità a frequenza base, Bluetooth opera a 1 Mbit/sec utilizzando la modulazione numerica a traslazione di frequenza Gaussiana (GFSK, *Gaussian Frequency Shift Keying*). Nella modulazione numerica a traslazione di frequenza normale (FSK) ogni bit viene inviato trasmettendo una frequenza pari a $f_c \pm f_d$, dove f_c è la frequenza centrale del canale e f_d è una traslazione di almeno 115 kHz. Il brusco cambio di frequenza tra un bit e il successivo consuma ulteriore banda passante. Nella GFSK il cambio di frequenza avviene in modo più graduale per fare migliore uso dello spettro di frequenze disponibile. La **Figura e9.30** mostra le frequenze trasmesse per inviare una sequenza di 0 e 1 in un canale a 2402 MHz usando FSK e GFSK.

Il modulo BlueSMiRF, mostrato nella **Figura e9.31(a)**, contiene un trasmettitore Bluetooth di classe 2, un modem e la circuiteria di interfaccia in una piccola scheda di circuito stampato dotata di interfaccia seriale. Può comunicare con un altro dispositivo Bluetooth, come una chiavetta USB Bluetooth collegata a un PC. Può quindi permettere una connessione seriale wireless tra il Pi e un PC simile a quella rappresentata nella Figura e9.15 ma senza la necessità del cavo. Il collegamento wireless è compatibile con lo stesso software usato per il collegamento con cavo.

La **Figura e9.32** mostra lo schema di questo collegamento. Il contatto TX di BlueSMiRF si collega al contatto RX del Pi e viceversa. I contatti RTS e CTS sono collegati tra loro cosicché BlueSMiRF "stringe le mani a se stesso" (nessun *handshaking*).

La comunicazione di default di BlueSMiRF è a 115.2 kilo baud, con 8 bit di

Tabella e9.9 Classi Bluetooth.

Classe	Potenza del trasmettitore (mW)	Distanza (m)
1	100	100
2	2.5	10
3	1	5

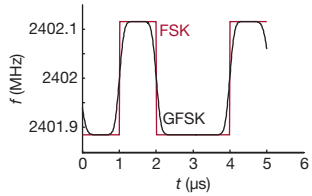


Figura e9.30
Forme d'onda FSK e GFSK.

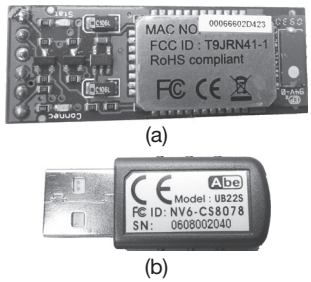


Figura e9.31
Il modulo BlueSMiRF e una chiavetta USB Bluetooth.

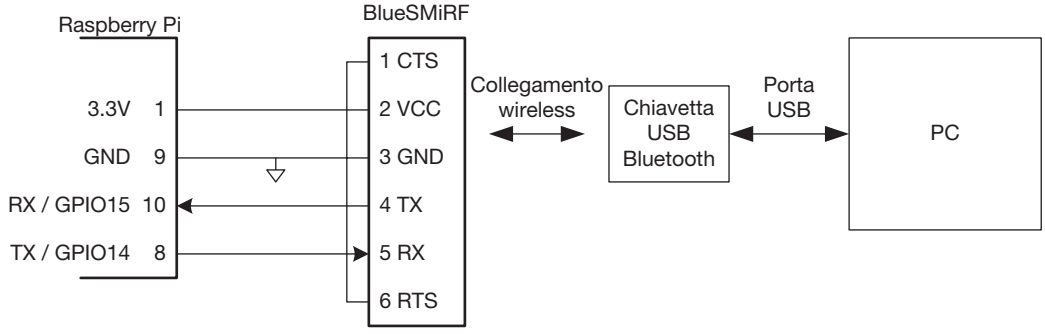


Figura e9.32 Collegamento tra Raspberry Pi e un PC mediante BlueSMiRF.

dato, 1 bit di stop e nessun bit di parità né controllo di flusso. Il modulo lavora con i livelli della logica digitale a 3.3 V, quindi non è necessario un *transceiver* RS-232 per collegarlo a un altro dispositivo a 3.3 V.

Per usare l'interfaccia basta inserire la chiavetta USB Bluetooth nel PC e accendere il Pi e il BlueSMiRF: la luce rossa STAT lampeggia sul BlueSMiRF a indicare che il modulo è in attesa di stabilire una connessione. Si deve quindi aprire l'icona Bluetooth sul PC e usare la procedura di aggiunta di un dispositivo Bluetooth per accoppiare la chiavetta con il BlueSMiRF. La password predefinita è 1234. Ricordarsi di prendere nota di quale porta COM è assegnata alla chiavetta. Ora la comunicazione può avvenire esattamente come con un cavo seriale. Attenzione che la chiavetta opera normalmente a 9600 baud, quindi PuTTY va configurato di conseguenza.

9.4.4 Controllo di motori

Un'altra tipica applicazione dei microcontrollori è il pilotaggio di attuatori come i motori. Questo paragrafo descrive tre tipi di motori: i motori in corrente continua (DC, *Direct Current*), i servomotori e i motori passo-passo. I motori in corrente continua richiedono un'elevata corrente di pilotaggio, quindi un pilotaggio di potenza come un ***H-bridge*** (ponte-H) deve essere inserito tra il microcontrollore e il motore. Questi motori hanno anche bisogno di un encoder ad albero (*shaft encoder*) se serve conoscere la posizione attuale del motore. I servomotori accettano un segnale a modulazione di ampiezza impulsi (PWM) che ne specifica la posizione su un limitato intervallo angolare. Sono facili da pilotare ma non altrettanto potenti e non adatti per rotazione continua. I motori passo-passo accettano una sequenza di impulsi, ciascuno dei quali provoca la rotazione del motore per un certo valore angolare detto appunto passo. Sono più costosi e richiedono anche loro un H-bridge per generare l'elevata corrente di pilotaggio, ma la loro posizione può essere controllata con molta precisione.

I motori possono richiedere correnti decisamente elevate e produrre picchi negli alimentatori che disturbano la logica digitale. Un modo per ridurre questi problemi è quello di usare per il motore un alimentatore o una batteria separati da quelli della logica digitale.

9.4.4.1 Motori in corrente continua

La **Figura e9.33** mostra la struttura di un motore in corrente continua a spazzole. Il motore è un dispositivo a due contatti. Contiene dei magneti permanenti fissi che costituiscono lo **statore** e un elettromagnete rotante, detto **rotore**, collegato all'albero. La parte anteriore del rotore è collegata a un anello metallico spezzato in due parti denominato **commutatore**. Due spazzole metalliche collegate ai contatti di potenza (cioè i contatti di ingresso) strisciano sul commutatore fornendo corrente all'elettromagnete del rotore; questo crea un campo magnetico nel rotore che lo forza a ruotare per allinearsi al campo magnetico generato dallo statore. Appena il rotore ha fatto una certa rotazione e si è quasi allineato con il campo magnetico dello statore, le spazzole toccano i lati opposti del commutatore, invertendo il flusso di corrente e il campo magnetico e portando alla continua rotazione del rotore.

I motori in corrente continua tendono a ruotare a diverse migliaia di rotazioni al secondo (RPM, *Rotations Per Second*) con una coppia molto bassa, quindi si usa spesso un treno di ingranaggi per diminuire la velocità a livelli più ragionevoli e aumentare la coppia. La ditta Pittman produce un'ampia gamma di motori in corrente continua e relativi accessori, mentre i motori giocattolo molto più economici sono diffusi tra gli hobbisti.

Un motore in corrente continua richiede corrente e tensione elevate per

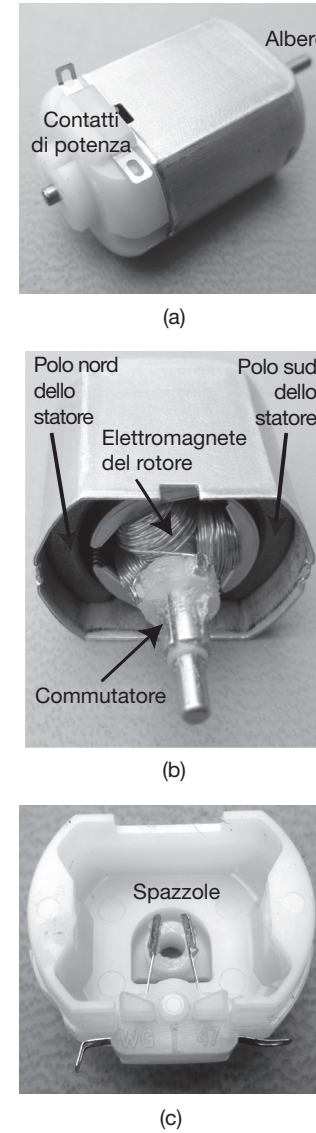


Figura e9.33
Motore in corrente continua.

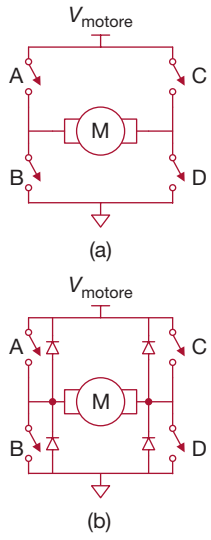


Figura e9.34
H-bridge.

generare una potenza significativa verso il carico, e la corrente deve essere reversibile per consentire la rotazione in entrambi i sensi. La maggior parte dei microcontrollori non è in grado di generare abbastanza corrente da pilotare direttamente un motore in corrente continua: fanno quindi uso di un H-bridge, che concettualmente contiene quattro interruttori controllati elettricamente, come mostrato nella **Figura e9.34(a)**. Se gli interruttori A e D sono chiusi, la corrente fluisce da sinistra verso destra nel motore e lo fa ruotare in un senso. Se sono chiusi B e C la corrente fluisce da destra verso sinistra e il motore ruota in senso opposto. Se A e C oppure B e D sono chiusi, la tensione ai capi del motore è 0 e il motore frena. Se nessun interruttore è chiuso, il motore avanza fino a fermarsi per inerzia. Gli interruttori dell'H-bridge sono dei transistori di potenza. L'H-bridge contiene anche della logica digitale per controllare adeguatamente gli interruttori.

Quando la corrente nel motore cambia bruscamente, l'induttanza dell'elettromagnete del motore genera un picco di tensione molto elevato che può danneggiare i transistori di potenza. Quindi molti H-bridge hanno anche dei diodi di protezione in parallelo agli interruttori, come mostrato nella **Figura e9.34(b)**. Se il picco induttivo porta uno dei contatti del motore sopra V_{motore} o sotto il riferimento di massa, i diodi si accendono e bloccano la tensione a un livello di sicurezza. Gli H-bridge possono dover dissipare grosse quantità di potenza, quindi può essere necessario un dissipatore di calore per evitare surriscaldamenti eccessivi.

ESEMPIO e9.12

Veicolo autonomo. Progettare un sistema nel quale un Raspberry Pi controlla due motori per un'auto a guida automatica. Scrivere una libreria di funzioni per inizializzare il driver dei motori e per far avanzare o retrocedere la macchina, farla curvare a destra o a sinistra e farla fermare. Usare PWM per controllare la velocità dei motori.

Soluzione La **Figura e9.35** mostra una coppia di motori in corrente continua controllati da un Pi tramite il dual *H-bridge* SN754410 della Texas Instruments. L'H-bridge richiede un'alimentazione V_{CC1} a 5 V per la logica e una seconda alimentazione V_{CC2} a 4.5-36 V per i motori; ha un valore V_{IH} = 2 V quindi è compatibile con l'I/O a 3.3 V del Pi. Può erogare fino a 1 A di corrente a ciascuno dei due motori. V_{motore} deve essere fornita da una batteria separata: l'uscita a 5 V del Pi non è in grado di fornire corrente sufficiente per i motori, e il Pi potrebbe venire danneggiato.

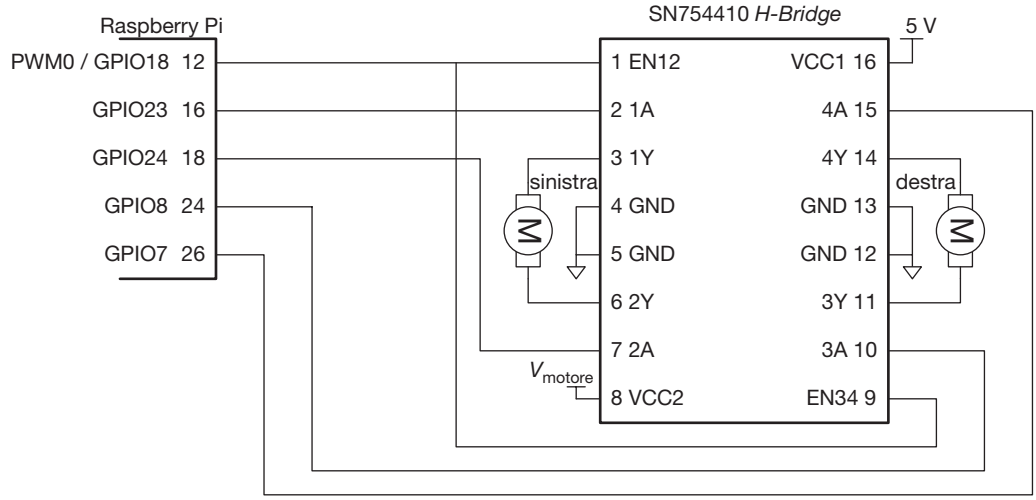


Figura e9.35 Controllo di motori con un dual H-bridge.

Tabella e9.10 **Controllo dell'H-Bridge.**

EN12	1A	2A	Motore
0	X	X	Rallenta per inerzia
1	0	0	Frena
1	0	1	Va indietro
1	1	0	Va avanti
1	1	1	Frena

La **Tabella e9.10** mostra come gli ingressi di ciascun H-bridge controllano il relativo motore. Il microcontrollore attiva i contatti di abilitazione con un segnale PWM per controllare la velocità dei motori, e gestisce gli altri quattro contatti per controllare la direzione di ogni motore.

Il PWM è configurato per lavorare a circa 5 kHz con un duty cycle variabile da 0 a 100%. Frequenze di PWM molto superiori alla banda passante del motore danno come risultato un lento movimento. Si noti che la relazione tra duty cycle e velocità del motore non è lineare, e che sotto un certo valore di duty cycle il motore non si muove.

```
#include "EasyPIO.h"

// Motor Constants
#define MOTOR_1A 23
#define MOTOR_2A 24
#define MOTOR_3A 8
#define MOTOR_4A 7

void setSpeed(float dutycycle) {     // pwmInit() must be called first.
  setPWM(5000, dutycycle);
}

void setMotorLeft(int dir) {         // dir of 1 = forward, 0 = backward
  digitalWrite(MOTOR_1A, dir);
  digitalWrite(MOTOR_2A, !dir);
}

void setMotorRight(int dir) {        // dir of 1 = forward, 0 = backward
  digitalWrite(MOTOR_3A, dir);
  digitalWrite(MOTOR_4A, !dir);
}

void forward(void) {
  setMotorLeft(1); setMotorRight(1); // Both motors drive forward
}

void backward(void) {
  setMotorLeft(0); setMotorRight(0); // Both motors drive backward
}

void left(void) {
  setMotorLeft(0); setMotorRight(1); // Left back, right forward
}

void right(void) {
  setMotorLeft(1); setMotorRight(0); // Right back, left forward
}

void halt(void) {                    // Turn both motors off
  digitalWrite(MOTOR_1A, 0);
  digitalWrite(MOTOR_2A, 0);
  digitalWrite(MOTOR_3A, 0);
  digitalWrite(MOTOR_4A, 0);
```

```
}

void initMotors(void) {
  pinMode(MOTOR_1A, OUTPUT);
  pinMode(MOTOR_2A, OUTPUT);
  pinMode(MOTOR_3A, OUTPUT);
  pinMode(MOTOR_4A, OUTPUT);
  halt();                              // Ensure motors are not spinning
  pwmInit();                           // Turn on PWM
  setSpeed(0.75);                      // Default to partial power
}

main(void) {
  pioInit();
  initMotors();
  forward(); delayMillis(5000);
  backward(); delayMillis(5000);
  left(); delayMillis(5000);
  right(); delayMillis(5000);
  halt();
}
```

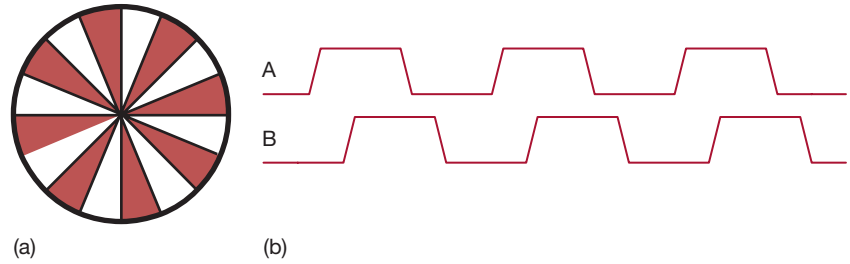


Figura e9.36 **Encoder ad albero: (a) disco, (b) uscite di quadratura.**

Nell'esempio precedente non c'è modo di misurare la posizione di ciascun motore. I due motori difficilmente sono identici: quasi certamente uno dei due gira un po' più veloce dell'altro, portando fuori strada la macchina. Per risolvere questo problema alcuni sistemi aggiungono degli encoder ad albero (*shaft encoder*). La **Figura e9.36(a)** mostra un semplice encoder ad albero costituito da un disco diviso in spicchi fissato sull'albero del motore. Un LED viene messo da un lato del disco e un sensore di luce dall'altro lato. L'encoder produce un impulso ogni volta che uno spicchio trasparente passa davanti al LED. Un microcontrollore può contare questi impulsi per misurare l'angolo di rotazione totale dell'albero. Usando due coppie LED/sensore sfasate di mezzo spicchio, un encoder più evoluto produce le uscite di quadratura mostrate nella **Figura e9.36(b)** che indicano sia la direzione sia l'angolo di rotazione dell'albero. A volte gli encoder aggiungono un buco sul disco per segnalare quando l'albero si trova in posizione di inizio giro.

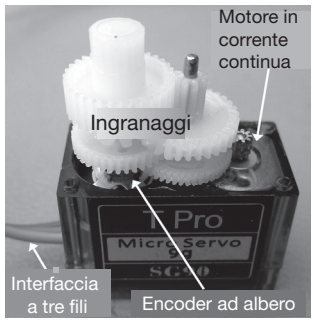


Figura e9.37
Il servomotore SG90.

9.4.4.2 Servomotori

I servomotori sono motori in corrente continua integrati con un treno di ingranaggi, un encoder ad albero e un po' di logica per facilitarne l'utilizzo. Hanno una limitata capacità di rotazione, generalmente 180°. La **Figura e9.37** mostra un servomotore senza il coperchio per mostrare gli ingranaggi. Il servomotore ha tre contatti di interfaccia: alimentazione (di solito 5 V), massa e controllo. Il controllo è generalmente un segnale modulato in ampiezza impulsi a 50 Hz. La logica di controllo porta l'albero alla posizione determinata dal duty cycle del segnale di controllo. L'encoder ad albero è generalmente

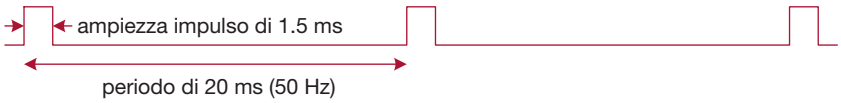


Figura e9.38
Forma d'onda di controllo del servomotore.

un potenziometro a rotazione che produce una tensione che dipende dalla posizione dell'albero.

In un tipico servomotore con 180° di rotazione, un'ampiezza di impulsi di 0.5 ms fa ruotare l'albero in posizione 0°, 1.5 ms in posizione 90° e 2.5 ms in posizione 180°. La **Figura e9.38** mostra un segnale di controllo con impulsi di 1.5 ms. Portare il servomotore oltre le sue massime posizioni lo fa urtare con i punti di arresto meccanici e può danneggiarlo. La potenza per il motore proviene dal contatto di alimentazione invece che da quello di controllo, quindi si può collegare il controllo direttamente a un microcontrollore senza bisogno dell'H-bridge. I servomotori sono comunemente usati per il controllo remoto di aeromodelli e piccoli robot perché sono piccoli, leggeri ed economici. Trovare un servomotore con documentazione tecnica precisa può essere difficile: di solito comunque il filo rosso è la potenza e il filo nero o marrone la massa.

ESEMPIO e9.13

Servomotore. Progettare un sistema con un Raspberry Pi che pilota un servomotore alla posizione angolare desiderata.

Soluzione La **Figura e9.39** mostra lo schema delle connessioni al servomotore SG90, con i colori dei fili nel cavo di connessione. Il servomotore lavora a tensione di alimentazione di 4.0-7.2 V. Può richiedere fino a 0.5 A di corrente se deve generare una forza elevata, ma può essere pilotato direttamente dal Pi se il carico è leggero. Un filo porta il segnale PWM, che può essere fornito a 3.3 V oppure a 5 V. Il codice configura la generazione del segnale PWM e calcola il duty cycle appropriato per la rotazione angolare desiderata. Nell'esempio passa ciclicamente nelle tre posizioni 0°, 90° e 180°.

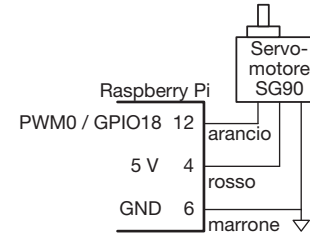


Figura e9.39
Controllo del servomotore.

```
#include "EasyPIO.h"
void setServo(float angle) {
 setPWM(50.0, 0.025 + (0.1 * (angle / 180)));
}

void main(void) {
 pioInit();
 pwmInit();
 while (1) {
  setServo(0.0);   // Left
  delayMillis(1000);
  setServo(90.0);  // Center
  delayMillis(1000);
  setServo(180.0); // Right
  delayMillis(1000);
 }
}
```

È anche possibile convertire un normale servomotore in un servomotore a rotazione continua smontandolo con attenzione, rimuovendo gli arresti meccanici e sostituendo il potenziometro con un divisore di tensione fisso. Su Web si trovano molte descrizioni di come procedere con i diversi tipi di servomotori. In tal caso il segnale PWM controlla la velocità invece della posizione, con 1.5 ms per indicare l'arresto, 2.5 ms per indicare la massima velocità in avanti e 0.5 ms per indicare la massima velocità all'indietro. Un servomotore a

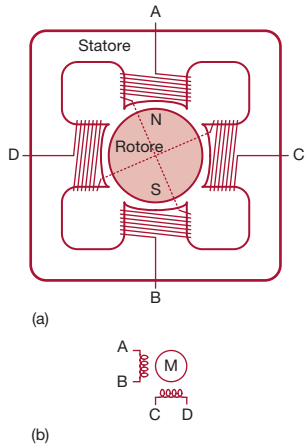


Figura e9.40
Motore bipolare a due fasi: (a) schema semplificato e (b) simbolo.

rotazione continua può essere molto più economico di un motore in corrente continua con H-bridge e treno di ingranaggi.

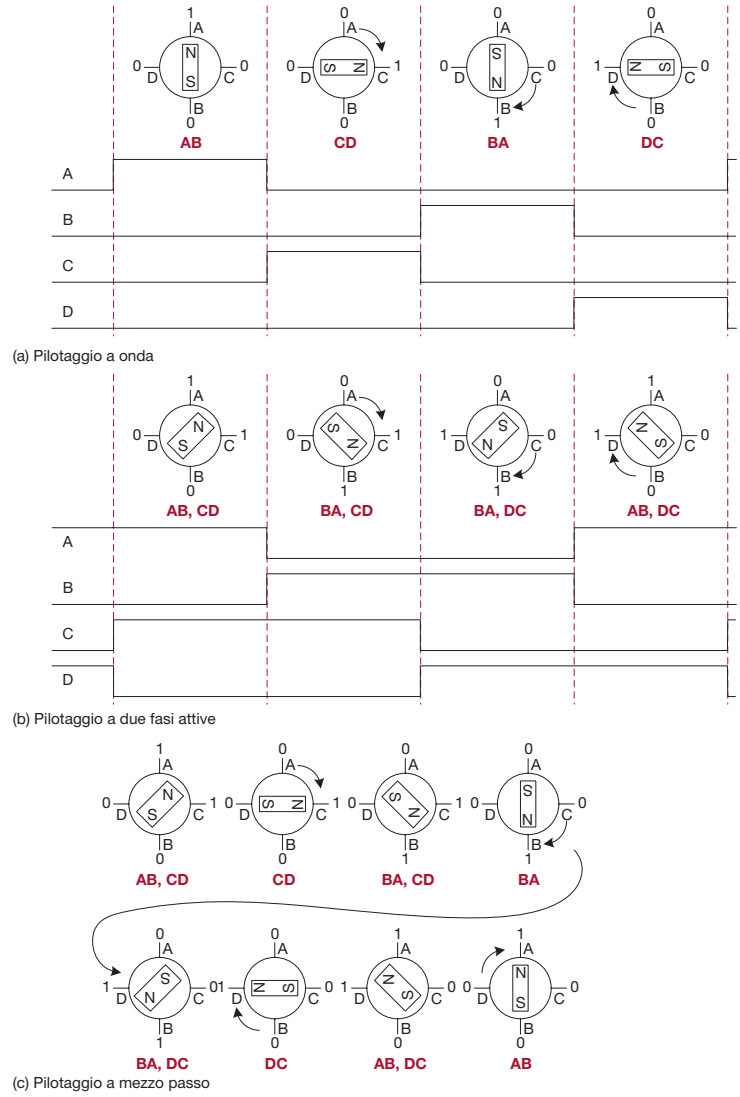
9.4.4.3 Motori passo-passo

I motori passo-passo ruotano di passi discreti a seguito degli impulsi elettrici forniti agli ingressi. Il singolo passo è di solito di pochi gradi, il che consente posizionamenti precisi e rotazione continua. I motori passo-passo piccoli sono generalmente dotati di due insiemi di bobine detti **fasi** collegati in modo **bipolare** o **unipolare**. A parità di dimensione i motori bipolari sono più potenti e meno costosi, ma richiedono un H-bridge, mentre i motori unipolari possono essere pilotati con transistori funzionanti da interruttori. In questo paragrafo ci si concentra sui motori passo-passo bipolari, più efficienti.

La **Figura e9.40(a)** mostra lo schema semplificato di un motore bipolare a due fasi con un passo di 90°. Il rotore è un magnete permanente con poli nord e sud. Lo statore è un elettromagnete con due coppie di bobine che costituiscono le due fasi. I motori bipolari a due fasi hanno dunque quattro contatti. La **Figura e9.40(b)** mostra il simbolo del motore passo-passo nel quale le due bobine sono rappresentate come induttori. I motori commerciali aggiungono treni di ingranaggi per ridurre il passo di uscita e aumentare la coppia.

La **Figura e9.41** mostra tre comuni sequenze di pilotaggio di un motore

Figura e9.41
Pilotaggio del motore bipolare.



bipolare a due fasi. La **Figura e9.41(a)** mostra il **pilotaggio a onda** (*wave drive*) nel quale le bobine sono energizzate nella sequenza AB – CD – BA – DC, dove BA indica che l'avvolgimento AB viene energizzato con corrente che fluisce nella direzione opposta (da cui il nome di motore bipolare). Il rotore ruota di 90° a ogni passo. La **Figura e9.41(b)** mostra il **pilotaggio a due fasi attive** (*two-phase-on drive*) dato dalla sequenza (AB, CD) – (BA, CD) – (BA, DC) – (AB, DC), dove (AB, CD) indica che entrambe le bobine AB e CD vengono energizzate simultaneamente. Il rotore ruota ancora di 90° ad ogni passo, ma si allinea a metà strada tra le posizioni dei due poli. Questa modalità dà la massima coppia, perché entrambe le bobine forniscono energia simultaneamente. La **Figura e9.41c** mostra il **pilotaggio a mezzo passo** (*half-way drive*) dato dalla sequenza (AB, CD) – CD – (BA, CD) – BA – (BA, DC) – DC – (AB, DC) – AB. Il rotore ruota di 45° a ogni passo. La frequenza di avanzamento della sequenza determina la velocità di rotazione. Per invertire il senso di rotazione, le stesse sequenze vengono applicate in ordine inverso.

Nei motori reali, il rotore ha molti poli per rendere l'angolo di rotazione a ogni passo molto più piccolo. Per esempio, la **Figura e9.42** mostra il motore passo-passo bipolare AIRPAX LB82773-M1 che ha un passo di 7.5 gradi. Opera a 5 V e richiede 0.8 A per ogni bobina.

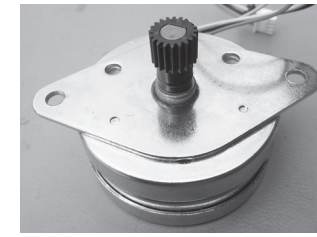


Figura e9.42
Il motore passo-passo bipolare AIRPAX LB82773-M1.

La coppia nel motore è proporzionale alla corrente nella bobina, che a sua volta è determinata dalla tensione applicata e dal valore di induttanza L e di resistenza R della bobina. La modalità operativa più semplice è definita **pilotaggio a tensione diretta** (L/R *drive*), nel quale la tensione V è direttamente applicata alla bobina: la corrente sale fino a $I = V/R$ con una costante di tempo pari a L/R, come mostrato nella **Figura e9.43(a)**. Questo funziona bene alle basse velocità. A velocità più alte, la corrente non ha abbastanza tempo per salire fino al massimo livello, come mostrato nella **Figura e9.43(b)**, e la coppia scende.

Una modalità più efficiente di pilotare un motore passo-passo è usare la modulazione di ampiezza impulsi a tensioni più elevate. Tali tensioni fanno

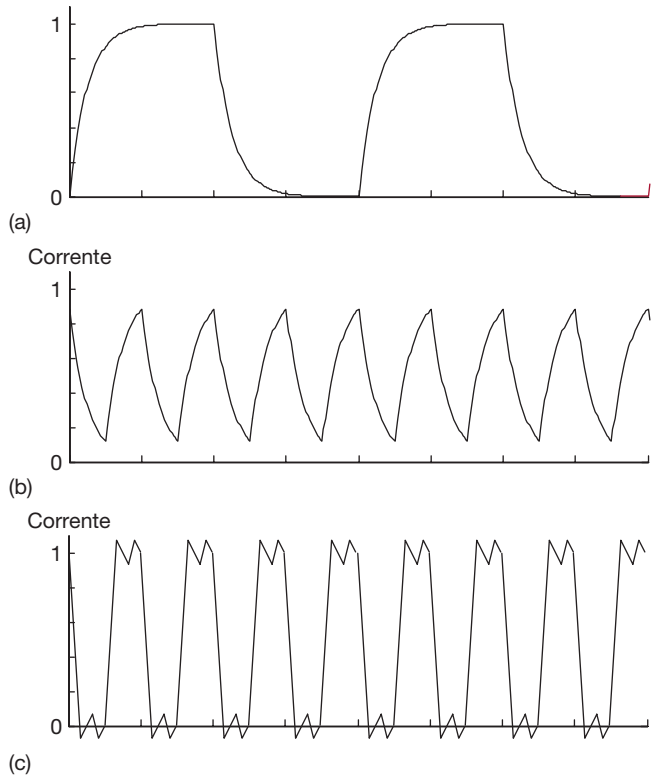


Figura e9.43
Correnti nel motore passo-passo bipolare a pilotaggio diretto: (a) rotazione lenta, (b) rotazione veloce), (c) rotazione veloce con pilotaggio con modulatore.

salire la corrente fino al valore massimo più rapidamente, poi la corrente viene spenta (PWM) per evitare di sovraccaricare il motore. La tensione viene quindi modulata (*chopped*) per mantenere la corrente vicina al livello desiderato. Si parla di **pilotaggio con modulatore a corrente costante** (*chopper constant current drive*), mostrato nella **Figura e9.43c**. Il controllore usa una piccola resistenza in serie al motore per conoscere la corrente applicata misurando la caduta di tensione sulla resistenza, e dà un segnale di abilitazione all'H-bridge per spegnere il pilotaggio quando la corrente raggiunge il livello desiderato. Teoricamente il microcontrollore potrebbe generare le forme d'onda necessarie, ma è più facile usare un controllore di motore passo-passo. Il controllore L297 della ST Microelectronics è una buona scelta, specie se usato insieme al dual H-bridge L298 con contatti di misurazione corrente e corrente di picco di 2A. Sfortunatamente il componente L298 non è disponibile in formato DIP, quindi è difficile da usare su schede prototipali. Le note applicative della ST AN460 e AN470 sono un ottimo riferimento tecnico per i progettisti di motori passo-passo.

ESEMPIO e9.14

Pilotaggio a onda diretta di un motore passo-passo bipolare. Progettare un sistema di pilotaggio di un motore passo-passo bipolare AIRPAX con senso di rotazione e velocità specificati usando il pilotaggio a onda diretta.

Soluzione La **Figura e9.44** mostra il motore passo-passo bipolare pilotato direttamente da un H-bridge con la stessa interfaccia del motore in corrente continua. Si noti che V_{CC2} deve fornire tensione e corrente abbastanza alte da soddisfare la richiesta del motore, altrimenti il motore potrebbe saltare dei passi all'aumentare della velocità.

```
#include "EasyPIO.h"

#define STEPSIZE 7.5
#define SECS_PER_MIN 60
#define MICROS_PER_SEC 1000000
#define DEG_PER_REV 360

int stepperPins[] = {18, 8, 7, 23, 24};
int curStepState; // Keep track of the current position of stepper motor

void stepperInit(void) {
  pinsMode(stepperPins, 5, OUTPUT);
```

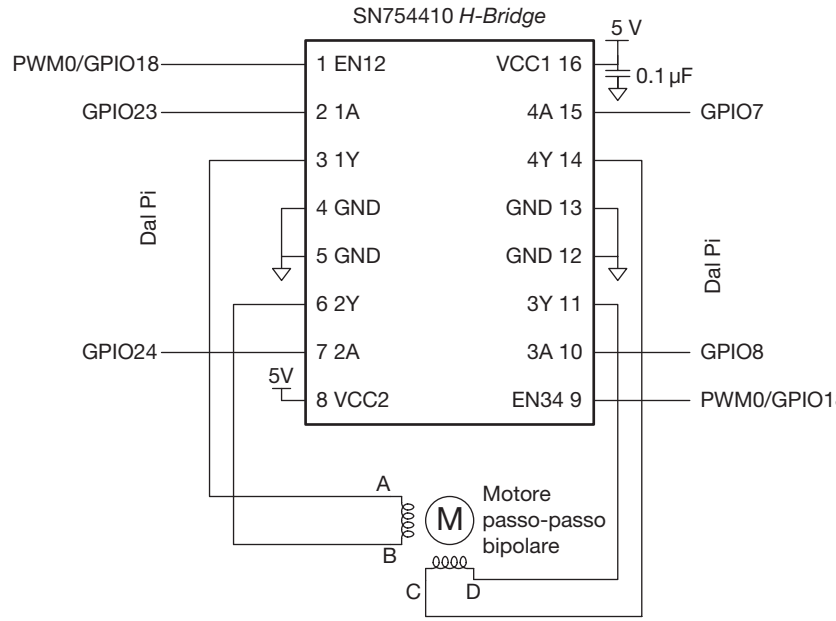


Figura e9.44
Pilotaggio a onda diretta di un motore passo-passo bipolare con H-bridge.

```
  curStepState = 0;
}
void stepperSpin(int dir, int steps, float rpm) {
  int sequence[4] = {0b00011, 0b01001, 0b00101, 0b10001};  //{2A, 1A, 4A,
3A, EN}
  int step = 0;
  unsigned int microsPerStep = (SECS_PER_MIN * MICROS_PER_SEC * STEPSIZE) /
     (rpm * DEG_PER_REV);
  for (step = 0; step < steps; step++) {
   digitalWrites(stepperPins, 5, sequence[curStepState]);
   if (dir==0) curStepState = (curStepState + 1) % 4;
   else        curStepState = (curStepState + 3) % 4;
   delayMicros(microsPerStep);
  }
}
void main(void) {
  pioInit();
  stepperInit();
  stepperSpin(1, 12000, 120); // Spin 60 revolutions at 120 rpm
}
```

9.5 ■ INTERFACCE A BUS

Un'**interfaccia a bus** (o semplicemente un **bus**) collega il processore alla memoria e/o ai dispositivi periferici. In generale, un bus supporta uno o più dispositivi **master** ("padrone") che possono lanciare richieste di lettura o scrittura sul bus, e uno o più dispositivi **slave** ("schiavo") che rispondono alle richieste; normalmente il processore è master e la memoria e le periferiche sono slave.

L'architettura AMBA (*Advanced Microcontroller Bus Architecture*) è uno standard open di interfaccia a bus per collegare i diversi componenti sullo stesso chip. Introdotta da ARM nel 1996, si è sviluppata in varie revisioni successive per incrementare prestazioni e potenzialità, ed è diventata lo standard di fatto per i microcontrollori embedded. Il bus AHB (*Advanced High-performance Bus*) è uno degli standard AMBA, ed esiste anche in una versione semplificata (AHB-Lite) che supporta un solo master. Questo paragrafo descrive AHB-Lite per illustrare le caratteristiche di una tipica interfaccia a bus e per mostrare come progettare memoria e periferiche in grado di collegarsi a un bus standard.

AHB è un esempio di bus di lettura punto-a-punto, a differenza di architetture di bus meno recenti che usano un unico bus condiviso al quale ogni slave accede mediante porte tristate. L'uso di collegamenti punto-a-punto tra ogni slave e il master e di un multiplexer di lettura permette al bus di operare più velocemente ed evita gli sprechi di potenza quando uno slave attiva le proprie porte tristate prima che un altro le abbia disattivate.

9.5.1 AHB-Lite

La **Figura e9.45** mostra un semplice bus AHB-Lite che collega un processore (master del bus) a una RAM, una ROM e due periferiche (slave). Si noti che il bus è molto simile a quello della Figura e9.1, a parte i nomi cambiati. Il master

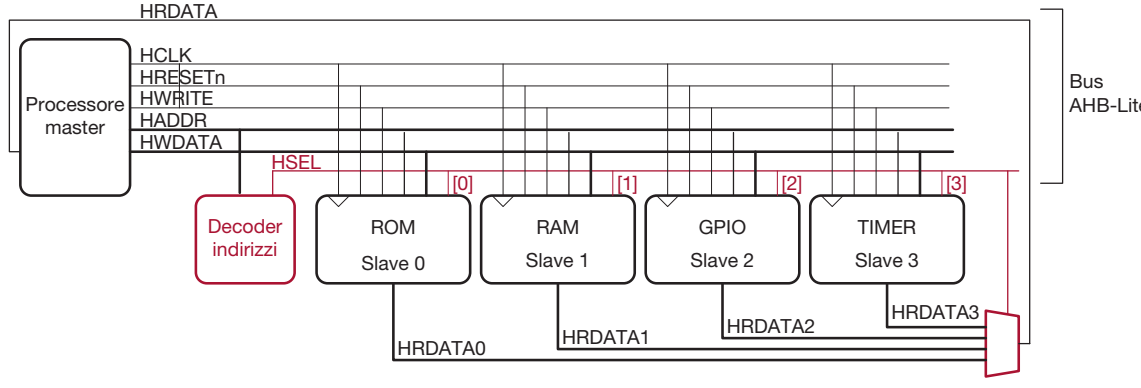


Figura e9.45 Il bus AHB-Lite.

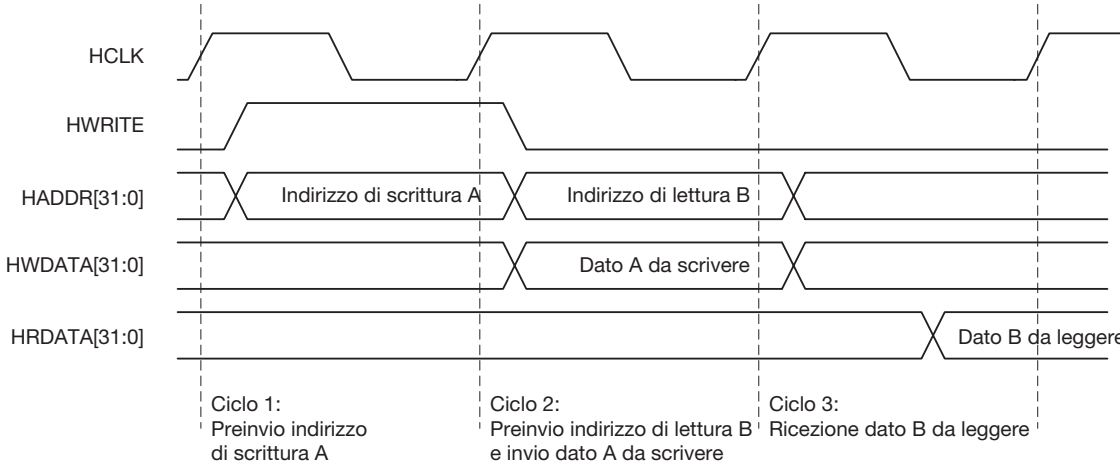


Figura e9.46 Temporizzazioni dei trasferimenti su AHB-Lite.

fornisce un segnale di clock sincrono (HCLK) a tutti gli slave e può inviare loro un reset portando a 0 il segnale HRESETn. Il master invia un indirizzo, e il decoder di indirizzi usa i bit più significativi di tale indirizzo per generare il segnale HSEL che seleziona quale dispositivo slave attivare, mentre i dispositivi slave usano i bit meno significativi dell'indirizzo per individuare la cella di memoria o il registro di periferica. Il master invia il segnale HWDATA se vuole scrivere. Ogni slave emette i propri dati nelle linee HRDATA, e un multiplexer seleziona il dato dallo slave indirizzato.

Il master invia un indirizzo a 32 bit in un ciclo, e scrive o legge il dato nel ciclo successivo. La scrittura (o lettura) è detta **trasferimento**. In caso di scrittura, il master attiva HWRITE e invia il dato a 32 bit su HWDATA. In caso di lettura, il master porta HWRITE a 0 e lo slave risponde con il dato a 32 bit su HRDATA. I trasferimenti possono sovrapporsi nel senso che il master può inviare l'indirizzo del prossimo trasferimento mentre sta leggendo o scrivendo il dato del trasferimento in corso. La **Figura e9.46** mostra le temporizzazioni del bus per una scrittura immediatamente seguita da una lettura. Si noti come il dato sia in ritardo di un ciclo rispetto all'indirizzo, e come i due trasferimenti siano parzialmente sovrapposti.

In questo esempio si fa l'ipotesi che il bus trasferisca una sola parola da 32 bit alla volta e che lo slave risponda in un ciclo di clock. AHB-Lite prevede dei segnali aggiuntivi per specificare la dimensione del trasferimento (8-1024 bit) e per trasferire gruppi composti da 4 a 16 elementi. Il master può anche specificare tipi di trasferimento, protezioni e blocco del bus. Gli slave possono portare a 0 il segnale HREADY per segnalare che hanno bisogno di più cicli di clock per rispondere, e possono attivare il segnale HRESP per segnalare un errore. Il lettore interessato può consultare la documentazione *AMBA 3 AHB-Lite Protocol Specification*, disponibile sul Web.

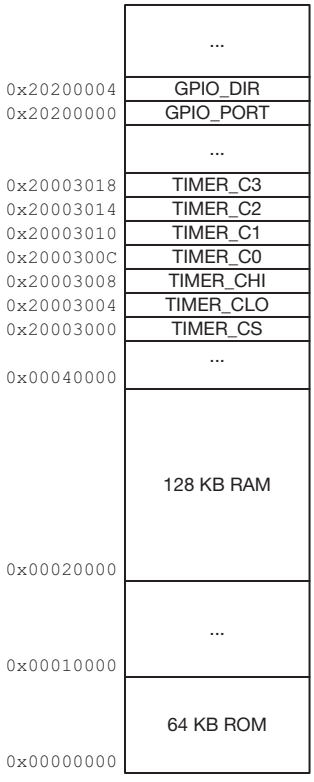


Figura e9.47
Mappa di memoria del sistema.

9.5.2 Esempio di interfaccia a memoria e periferica

Questo paragrafo illustra come collegare una RAM, una ROM, il GPIO e un Timer al processore mediante il bus AHB-Lite. La **Figura e9.47** mostra una possibile mappa di memoria per il sistema della Figura e9.45 con 128 KB di RAM e 64 KB di ROM. Il GPIO controlla 32 piedini di I/O. Il registro a 32 bit GPIO_DIR specifica se ciascun piedino è un'uscita (1) oppure un ingresso (0). Il registro a 32 bit GPIO_PORT può essere scritto per emettere i valori dei piedini di uscita e letto per restituire i valori dei piedini di ingresso. Il modulo Timer assomiglia al contatore BCM2835 descritto nel paragrafo 9.3.5, che contiene un contatore a 64 bit (TIMER_CHI:TIMER_CLO) incrementato

alla frequenza HCLK, quattro canali di confronti (TIMER_C3:0) e un registro di risultato del confronto (TIMER_CS).

L'Esempio HDL e9.1 presenta il codice SystemVerilog del sistema. Il decoder si basa sulla suddetta mappa di memoria. Memorie e periferiche sono interfacciate al bus. I segnali non strettamente necessari sono stati omessi: per

ESEMPIO HDL e9.1

```
module ahb_lite(input logic          HCLK,
                input logic          HRESETn,
                input logic  [31:0]  HADDR,
                input logic          HWRITE,
                input logic  [31:0]  HWDATA,
                output logic [31:0]  HRDATA,
                inout tri    [31:0]  pins);

 logic [3:0] HSEL;
 logic [31:0] HRDATA0,  HRDATA1,  HRDATA2, HRDATA3;
 logic [31:0] pins_dir, pins_out, pins_in;
 logic [31:0] HADDRDEL;
 logic HWRITEDEL;

 // Delay address and write signals to align in time
    with data
 flop #(32) adrreg(HCLK, HADDR, HADDRDEL);
 flop #(1)      writereg(HCLK, HWRITE, HWRITEDEL);

 // Memory map decoding
 ahb_decoder dec(HADDRDEL, HSEL);
 ahb_mux mux(HSEL, HRDATA0, HRDATA1, HRDATA2,
             HRDATA3, HRDATA);

 // Memory and peripherals
 ahb_rom    rom (HCLK, HSEL[0], HADDRDEL[15:2],
            HRDATA0);
 ahb_ram    ram (HCLK, HSEL[1], HADDRDEL[16:2],
            HWRITEDEL, HWDATA, HRDATA1);
 ahb_gpio   gpio (HCLK, HRESETn, HSEL[2],
                  HADDRDEL[2], HWRITEDEL, HWDATA,
                  HRDATA2, pins);
 ahb_timer  timer(HCLK, HRESETn, HSEL[3],
                  HADDRDEL[4:2],
                  HWRITEDEL, HWDATA, HRDATA3);
endmodule

module ahb_decoder(input logic [31:0] HADDR,
                   output logic [3:0] HSEL);

 // Decode based on most significant bits of the
    address
 assign HSEL[0]=(HADDR[31:16]
              ==16'h0000); // 64KB ROM at 0x00000000
                           - 0x0000FFFF
 assign HSEL[1]=(HADDR[31:17]
              ==15'h0001); // 128KB RAM at 0x00020000
                           - 0x003FFFFF
 assign HSEL[2]=(HADDR[31:4]
              ==28'h2020000); // GPIO at 0x20200000
              - 0x20200007
assign HSEL[3]=(HADDR[31:8]
             ==24'h200030); // Timer at 0x20003000
             - 0x2000301B

endmodule
```

```
module ahb_mux(input logic    [3:0]  HSEL,
               input logic    [31:0] HRDATA0, HRDATA1,
               HRDATA2, HRDATA3,
               output logic   [31:0] HRDATA);
 always_comb
  casez(HSEL)
   4'b???1: HRDATA <= HRDATA0;
   4'b??10: HRDATA <= HRDATA1;
   4'b?100: HRDATA <= HRDATA2;
   4'b1000: HRDATA <= HRDATA3;
  endcase
endmodule

module ahb_ram(input  logic        HCLK,
               input  logic        HSEL,
               input  logic [16:2] HADDR,
               input  logic        HWRITE,
               input  logic [31:0] HWDATA,
               output logic [31:0] HRDATA);

 logic [31:0] ram[32767:0];   //128KB RAM organized as
                                32K x 32 bits
 assign HRDATA = ram[HADDR]; // *** check addressing is
                                correct

 always_ff @(posedge HCLK)
  if (HWRITE & HSEL) ram[HADDR] <= HWDATA;
endmodule

module ahb_rom(input logic         HCLK,
               input logic         HSEL,
               input logic  [16:2] HADDR,
               output logic [31:0] HRDATA);
 logic [31:0] rom[16383:0]; // 64KB ROM organized as
                              16K x 32 bits

 // *** load ROM from disk file

 assign HRDATA = rom[HADDR]; // *** check addressing is
                                correct
endmodule

module ahb_gpio(input  logic        HCLK,
                input  logic        HRESETn,
                input  logic        HSEL,
                input  logic [2]    HADDR,
                input  logic        HWRITE,
                input  logic [31:0] HWDATA,
                output logic [31:0] HRDATA,
                output logic [31:0] pin_dir,
                output logic [31:0] pin_out,
                input  logic [31:0] pin_in);

 logic [31:0] gpio[1:0]; // GPIO registers

// write selected register
```

```
 always_ff @(posedge HCLK or negedge HRESETn)
  if (~HRESETn) begin
   gpio[0] <= 32'b0; // GPIO_PORT
   gpio[1] <= 32'b0; // GPIO_DIR
  end else if (HWRITE & HSEL)
   gpio[HADDR] <= HWDATA;

 // read selected register
 assign HRDATA = HADDR ? gpio[1] : pin_in;

 // send value and direction to I/O drivers
 assign pin_out = gpio[0];
 assign pin_dir = gpio[1];
endmodule

module ahb_timer(input logic        HCLK,
                 input logic        HRESETn,
                 input logic        HSEL,
                 input logic [4:2]  HADDR,
                 input logic        HWRITE,
                 input logic [31:0] HWDATA,
                 output logic[31:0] HRDATA);

 logic [31:0] timers[6:0]; // timer registers
 logic [31:0] chi, clo;    // next counter value
 logic [3:0] match, clr;   //determine if counter
                             matches compare reg

 // write selected register and update tiers and
    match
    always_ff @(posedge HCLK or negedge HRESETn)
 if (~HRESETn) begin
  timers[0] <= 32'b0; // TIMER_CS
  timers[1] <= 32'b0; // TIMER_CLO
  timers[2] <= 32'b0; // TIMER_CHI
  timers[3] <= 32'b0; // TIMER_C0
  timers[4] <= 32'b0; // TIMER_C1
  timers[5] <= 32'b0; // TIMER_C2
  timers[6] <= 32'b0; // TIMER_C3
 end else begin
  timers[0] <= {28'b0, match};
  timers[1] <= (HWRITE & HSEL & HADDR == 3'b000) ?
                 HWDATA : clo
  timers[2] <= (HWRITE & HSEL & HADDR == 3'b000) ?
                 HWDATA : chi;
  if (HWRITE & HSEL & HADDR == 3'b011) timers[3] <=
        HWDATA;
  if (HWRITE & HSEL & HADDR == 3'b100) timers[4] <=
        HWDATA;
  if (HWRITE & HSEL & HADDR == 3'b101) timers[5] <=
        HWDATA;
  if (HWRITE & HSEL & HADDR == 3'b110) timers[6] <=
        HWDATA;
 end
 // read selected register
 assign HRDATA = timers[HADDR];

 // increment 64-bit counter as pair of TIMER_CHI, TIMER_CLO
 assign {chi, clo} = {timers[2], timers[1]} + 1;

 // generate matches: set match bit when counter match-
    es compare register
 // clear bit when a 1 is written to that position of
    the match register
 assign clr = (HWRITE & HSEL & HADDR == 3'b000 & HWDA-
                TA[3:0]);
 assign match[0] = ~clr[0] & (timers[0][0] | (timers[1]
                   == timers[3]));
 assign match[1] = ~clr[1] & (timers[0][1] | (timers[1]
                   == timers[4]));
 assign match[2] = ~clr[2] & (timers[0][2] | (timers[1]
                   == timers[5]));
 assign match[3] = ~clr[3] & (timers[0][3] | (timers[1]
                   == timers[6]));
endmodule

module gpio_pins(input logic [31:0] pin_dir, // 1 = out-
                                               put, 0 =
                                               input
                 input logic [31:0] pin_out, // value to
                                               drive on
                                               outputs
                 output logic [31:0] pin_in, // value read
                                               from pins
                 inout tri [31:0] pin); // tristate pins

 // Individual tristate control of each pin
 // No graceful way to control tristates on a per-bit
    basis in SystemVerilog
 genvar i;
 generate
 for (i=0; i<32; i=i+1) begin: pinloop
  assign pin[i] = pin_dir[i] ? pin_out[i] : 1'bz;
 end
 endgenerate

 assign pin_in = pin;
endmodule
```

esempio, la memoria ROM ignora le operazioni di scrittura. Il modulo GPIO è collegato anche a 32 piedini che possono comportarsi da ingressi o da uscite.

9.6 ■ SISTEMI DI I/O DEL PC

I personal computer (PC) usano un'ampia gamma di protocolli di I/O per vari scopi, inclusi i collegamenti a memoria, a disco, in rete, a schede interne di espansione, a dispositivi esterni. I protocolli sono evoluti nel tempo per offrire prestazioni molto elevate e per facilitare l'aggiunta di dispositivi da parte degli utenti. Tali risultati sono stati pagati in termini di elevata complessità dei protocolli di I/O. Questo paragrafo esplora i principali standard di I/O usati nei PC ed esamina alcune opzioni disponibili per collegare un PC a logica digitale progettata dall'utente o altri dispositivi esterni.

La **Figura e9.48** mostra la scheda madre di un PC con processore Core i5 o i7. Il processore è inscatolato in un *land grid array* (matrice a griglia di contatti) con 1156 areole dorate di contatto elettrico usate per fornire alimentazione e massa al processore e per collegarlo alla memoria e ai dispositivi di I/O. La scheda madre contiene gli alloggiamenti (*slot*) per la memoria DRAM, i connettori per un'ampia gamma di dispositivi di I/O, il connettore per l'alimentatore, regolatori di tensione e condensatori. Due moduli DRAM sono collegati mediante l'interfaccia DDR3. Periferiche come tastiere e webcam sono collegate mediante USB. Schede di espansione ad alte prestazioni come le schede grafiche usano l'alloggiamento PCI Express X16, mentre schede con prestazioni inferiori possono usare l'alloggiamento PCI Express x1 o gli alloggiamenti PCI meno recenti. Il PC si collega alla rete mediante il *jack* Ethernet. Il disco rigido si collega a una porta SATA. Nel resto del paragrafo si presenta una panoramica di tutti questi standard di I/O.

Uno dei progressi più significativi negli standard di I/O dei PC è stato lo sviluppo di collegamenti seriali ad alta velocità. Fino a qualche tempo fa, la maggior parte dell'I/O era realizzata sulla base di collegamenti paralleli costituiti da un ampio bus dati e da un segnale di clock. Al crescere del tasso di trasmissione dati, i diversi ritardi nei segnali nei diversi fili del bus hanno posto un limite alla velocità del bus stesso. Inoltre i bus connessi a più dispositivi soffrono di problemi sulle linee di trasmissione come riflessioni e

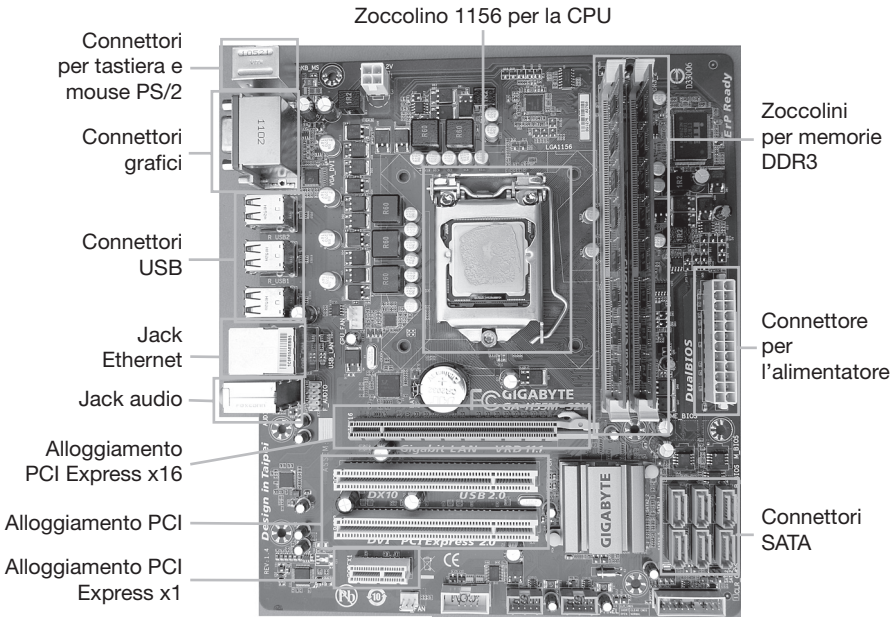


Figura e9.48
Scheda madre Gigabyte GA-H55M-S2V.

diversi tempi di arrivo a carichi diversi. Infine, il rumore può alterare i dati. I collegamenti seriali punto-a-punto eliminano molti di questi problemi. Il dato viene normalmente trasmesso su una coppia differenziale di fili, quindi un rumore esterno che influisce in modo uguale su entrambi i fili della coppia risulta innocuo. È facile inserire terminazioni elettriche alle linee di trasmissione, quindi si hanno piccole riflessioni (*vedi* il paragrafo A.8 sulle linee di trasmissione). Non viene trasmesso esplicitamente il segnale di clock: il sincronismo viene ricostruito da parte del dispositivo ricevente sulla base della temporizzazione delle transizioni di dato. La progettazione di collegamenti seriali ad alta velocità è un lavoro da specialisti, ma si hanno risultati superiori ai 10 Gb/s su fili di rame e ancora maggiori su fibre ottiche.

9.6.1 USB

Fino alla metà degli anni ’90, aggiungere una periferica a un PC richiedeva una certa esperienza tecnica. Aggiungere schede di espansione implicava aprire il PC, sistemare i ponticelli (*jumper*) sulla scheda nel modo opportuno, installare manualmente il driver software. Aggiungere un dispositivo RS-232 richiedeva di scegliere il cavo opportuno e configurare correttamente baud rate e bit di dato, di stop e di parità. USB (*Universal Serial Bus*, bus seriale universale) sviluppato da Intel, IBM, Microsoft e altre aziende, ha semplificato enormemente il lavoro necessario per aggiungere periferiche standardizzando i cavi di collegamento e la procedura di configurazione software. Al giorno d’oggi si vendono ogni anno miliardi di dispositivi USB.

USB 1.0 è stato rilasciato nel 1996. Usa un semplice cavo con quattro fili: 5 V, massa e una coppia di fili differenziali per trasportare i dati. Il connettore è realizzato in modo che non sia possibile collegare il cavo in modo sbagliato. USB 1.0 lavora a 12 Mb/s. Un dispositivo può assorbire fino a 500 mA dalla porta USB, quindi le tastiere, i mouse e altre periferiche possono essere alimentate dalla porta USB invece di richiedere batterie o cavi di alimentazione separati.

USB 2.0 è stato rilasciato nel 2000, portando la velocità massima a 480 Mb/s con un uso molto più evoluto dei fili differenziali. Grazie a questa velocità, USB è stato in grado di collegare al PC webcam e dischi rigidi esterni. Le chiavette USB con memorie flash hanno rapidamente sostituito i floppy come supporto per il trasferimento di file tra calcolatori diversi.

USB 3.0, rilasciato nel 2008, ha ulteriormente incrementato la velocità massima a 5 Gb/s. Il connettore ha la stessa forma, ma usa più fili che lavorano a velocità molto elevata. Consente di collegare dischi rigidi ad alte prestazioni. Circa nello stesso periodo USB ha introdotto anche una Specifica per Ricarica Batterie che incrementa significativamente la potenza elettrica inviata tramite la porta USB per velocizzare la ricarica dei dispositivi mobili.

La semplicità per l’utente è stata ottenuta a spese di una struttura hardware e software molto più complessa. Costruire un’interfaccia USB da zero è una grossa impresa. Anche solo scrivere un semplice driver di dispositivo USB è piuttosto difficile.

9.6.2 PCI e PCI Express

Il bus PCI (*Peripheral Component Interconnect*, interconnessione di componenti periferici) è uno standard di bus di espansione sviluppato da Intel divenuto molto diffuso intorno al 1994. Era utilizzato per aggiungere schede di espansione come porte seriali o USB aggiuntive, interfacce di rete, schede audio, modem, controllori di dischi, schede video. Il bus parallelo a 32 bit opera a 33 MHz, con un tasso di trasferimento dati di 133 MB/s.

La richiesta di schede di espansione PCI è andata scemando nel tempo.

Porte standard come Ethernet e SATA sono oggi integrate nella scheda madre. Molti dispositivi che un tempo richiedevano schede di espansione possono oggi essere collegati mediante USB 2.0 o 3.0. E le schede video attuali richiedono tassi di trasferimento dati superiori a quelli che PCI può offrire.

Le schede madre di oggi spesso hanno ancora qualche alloggiamento di espansione PCI, ma i dispositivi veloci come le schede video sono collegati mediante ***PCI Express*** (PCIe). PCIe fornisce una o più "corsie" (*lane*) di collegamento seriale ad alta velocità. In PCIe 3.0 ogni corsia può lavorare fino a 8 GB/s. Molte schede madre forniscono un alloggiamento x16 con 16 corsie per un tasso di trasferimento dati totale di 16 GB/s, utile per dispositivi ad alto traffico come le moderne schede video.

9.6.3 Memoria DDR3

La DRAM si collega al processore mediante un bus parallelo. Nel 2015 tale bus era lo standard DDR3, la terza generazione di bus di memoria *Double Data Rate* (a doppio tasso di trasferimento dati) che lavora a 1.5 V. Le schede madre più diffuse presentano due canali DDR3 per consentire l'accesso simultaneo a due banchi di memoria. Sta cominciando a diffondersi DDR4, che lavora a 1.2 V e a velocità superiore.

La **Figura e9.49** mostra un modulo di memoria DDR3 *dual inline* (DIMM): il modulo ha 120 contatti su ciascun lato, per un totale di 240 contatti, che includono un bus dati a 64 bit, un bus indirizzi a 16 bit multiplexato nel tempo, segnali di controllo, oltre a numerosi contatti di alimentazione e di massa. Nel 2015, i moduli DIMM contenevano da 1 a 16 GB di DRAM, ma si deve tenere conto che la capacità di memoria raddoppia ogni 2-3 anni.

Attualmente la DRAM opera a una frequenza di clock di 100-266 MHz. DDR3 fa lavorare il bus di memoria quattro volte più velocemente della DRAM, inoltre trasferisce dati sia sul fronte di salita sia sul fronte di discesa del clock: è quindi in grado di trasferire 8 parole di dato per ogni ciclo del clock della memoria. Con parole da 64 bit questo corrisponde a 6.4-17 GB/s di tasso di trasferimento dati. Per esempio, DDR3-1600 usa un clock di memoria a 200 MHz e un clock di I/O a 800 MHz per inviare 1600 milioni di parole al secondo, o 12 800 MB/s. Da qui il nome dei moduli di memoria, chiamati PC3-12800. Purtroppo la latenza della DRAM rimane alta, con circa 50 ns di attesa da una richiesta di lettura all'arrivo del primo dato.

9.6.4 Interconnessione in rete

I calcolatori si collegano a Internet tramite un'interfaccia di rete in grado di supportare il protocollo **TCP/IP** (*Transmission Control Protocol/Internet Protocol*). La connessione fisica può essere un cavo Ethernet oppure un collegamento wireless Wi-Fi.

Il protocollo **Ethernet** è definito dallo standard IEEE 802.3. È stato sviluppato allo Xerox Palo Alto Research Center (PARC) nel 1974. Originariamente lavorava a 10 Mb/s (il cosiddetto 10 Mbit Ethernet) ma sono ora diffuse le versioni a 100 Mbit (Mb/s) o 1 Gbit (Gb/s) con cavi in categoria 5 contenenti quattro coppie di doppini elettrici attorcigliati (*twisted pair*). 10 Gbit Ethernet

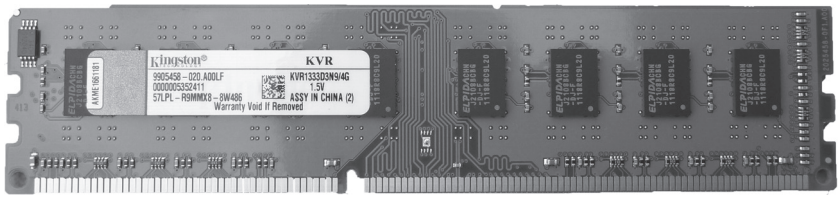


Figura e9.49
Modulo di memoria DDR3.

Tabella e9.11 Protocolli Wi-Fi 802.11.

Protocollo	Data di rilascio	Banda di frequenza (GHz)	Tasso di trasferimento dati (MB/s)	Distanza (m)
802.11b	1999	2.4	5.5-11	35
802.11g	2003	2.4	6-54	38
802.11n	2009	2.4/5	7.2-150	70
802.11ac	2013	5	433+	variabile

su connessioni in fibra ottica è sempre più diffuso per i calcolatori server e per altre macchine ad alte prestazioni, e sta cominciando a diffondersi anche 100 Gbit Ethernet.

Wi-Fi è il nome comunemente usato per indicare lo standard 802.11 per reti wireless. Opera nelle bande wireless senza autorizzazione a 2.4 e 5 GHz, nel senso che non serve l'autorizzazione di un radio operatore per trasmettere a bassa potenza in queste bande. La **Tabella e9.11** riassume le potenzialità di tre generazioni di Wi-Fi; lo standard emergente 802.11ac promette di spingere le comunicazioni wireless ben oltre 1 Gb/s. Le crescenti prestazioni derivano da avanzamenti nella modulazione e nell'elaborazione dei segnali, da antenne multiple e da maggiori ampiezze di banda dei segnali.

9.6.5 SATA

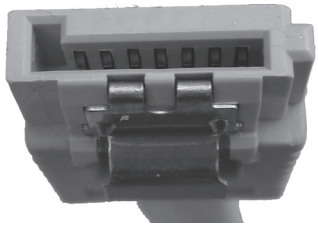


Figura e9.50
Cavo SATA.

I dischi rigidi interni richiedono un'interfaccia veloce al PC. Nel 1986, la Western Digital ha introdotto l'interfaccia IDE (*Integrated Digital Interface*) evoluta in seguito nello standard ATA (*AT Attachment*). Lo standard utilizza un ingombrante cavo a nastro con 40 o 80 fili con una lunghezza massima di 18 pollici (45.72 cm) per trasferire dati a 16-133 MB/s.

ATA è stato in seguito soppiantato da SATA (*Serial ATA*), che usa collegamenti seriali ad alta velocità per trasferire dati a 1.5, 3 o 6 GB/s con il più comodo cavo a 7 fili mostrato nella **Figura e9.50**. I più veloci dischi a stato solido del 2015 superavano i 500 MB/s di tasso di trasferimento dati, traendo il massimo vantaggio da SATA.

Un altro standard è SAS (*Serial Attached SCSI*), evoluzione di SCSI parallelo (*Small Computer System Interface*). SAS offre prestazioni paragonabili a SATA ma supporta connessioni più lunghe; è diffuso soprattutto nei calcolatori server.

9.6.6 Interfacciamento a un PC

Tutti gli standard finora descritti sono ottimizzati per elevate prestazioni e facilità di collegamento ma sono difficili da realizzare in hardware. Ingegneri e scienziati hanno spesso la necessità di trovare un modo per collegare al PC della circuiteria esterna, come sensori, attuatori, microcontrollori, dispositivi FPGA. La connessione seriale descritta nel paragrafo 9.3.4.2 è sufficiente per realizzare una connessione a basse prestazioni con un microcontrollore dotato di UART. In questo paragrafo si descrivono due ulteriori metodi: sistemi di acquisizione dati e collegamenti USB.

9.6.6.1 Sistemi di acquisizione dati

I sistemi di acquisizione dati (DAQ, *Data AcQuisition system*) collegano un calcolatore al mondo esterno usando più canali di I/O analogico e/o digitale. I DAQ sono oggi generalmente disponibili come dispositivi USB, quindi molto facili da installare. La National Instruments (NI) è il leader di mercato nella produzione di DAQ.

I prezzi dei DAQ ad alte prestazioni possono facilmente arrivare alle migliaia di dollari, soprattutto perché è un mercato di nicchia con scar-

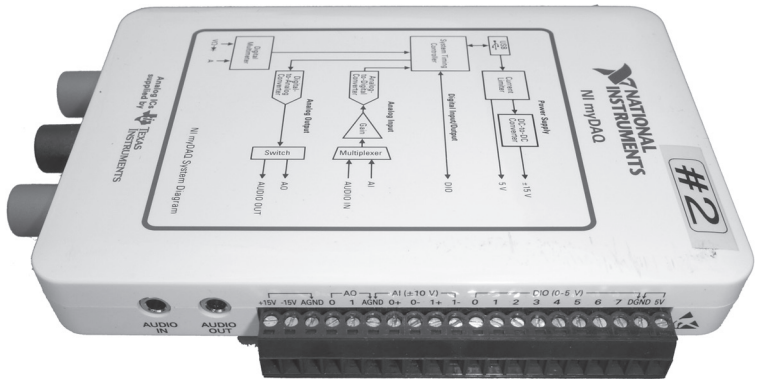


Figura e9.51
Il myDAQ della NI.

sa competizione. Fortunatamente la NI vende il suo maneggevole sistema myDAQ insieme al proprio software LabVIEW a prezzi scontati per studenti (200 dollari in America). La **Figura e9.51** mostra il myDAQ. È dotato di due canali analogici di ingresso e uscita a 200 kcampioni/sec con risoluzione di 16 bit e intervallo dinamico di tensione ±10 V, che possono essere configurati per lavorare come un oscilloscopio e un generatore di segnale. Ha inoltre otto linee di ingresso e uscita digitali compatibili con i sistemi a 3.3 e 5 V. È anche in grado di generare uscite di alimentazione a 5, 10 e –15 V e contiene un multimetro digitale in grado di misurare tensioni, correnti e resistenze. Quindi myDAQ è in grado di sostituire un intero banco di misura e di collaudo offrendo anche funzioni automatiche di memorizzazione dati.

La maggior parte dei DAQ di NI è controllata con LabVIEW, il linguaggio grafico di NI per il progetto di sistemi di misura e di controllo. Alcuni DAQ possono essere controllati anche da programmi in C usando l'ambiente LabWindows, da applicazioni Microsoft.NET usando l'ambiente Measurement Studio, o da Matlab usando il Data Acquisition Toolbox.

9.6.6.2 Collegamenti USB

Sono sempre più numerosi i prodotti che forniscono semplici ed economici collegamenti tra PC e dispositivi esterni mediante USB. Questi prodotti contengono librerie e driver preconfezionati, per consentire di scrivere facilmente programmi sul PC in grado di scambiare dati con un dispositivo FPGA o un microcontrollore.

FTDI (*Future Technology Devices International*) è un leader di mercato per questi sistemi. Per esempio, il cavo FTDI C232HM-DDHSL da USB a MPSSE (*Multi-Protocol Synchronous Serial Engine*) mostrato nella **Figura e9.52** ha un connettore USB da una parte e un'interfaccia SPI dall'altra parte operante fino a 30 Mb/s, più l'alimentazione a 3.3 V e quattro contatti di I/O di uso generale. La **Figura e9.53** mostra un esempio di connessione di un dispositivo FPGA al PC usando tale cavo, che se necessario può fornire l'alimentazione a 3.3 V al

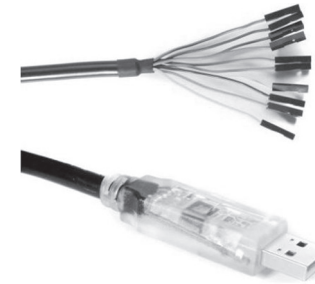


Figura e9.52
Il cavo FTDI da USB a MPSSE.
(© 2012 FTDI. Con permesso di riproduzione)

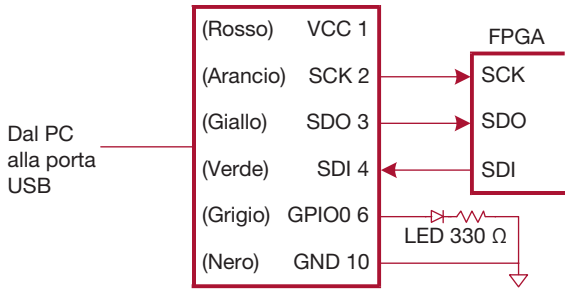


Figura e9.53
Interfaccia tra PC e FPGA usando il cavo C232HM-DDHSL da USB a MPSSE.

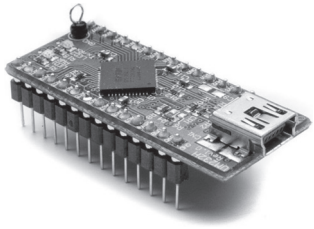


Figura e9.54
Il modulo FTDI UM232H.
(© 2012 FTDI. Con permesso di riproduzione)

dispositivo FPGA. I tre contatti SPI si collegano a un dispositivo FPGA slave come quello dell'Esempio e9.4. La figura mostra anche uno dei quattro contatti GPIO usato per pilotare un LED.

Il PC richiede che sia installata la libreria `.dll` (*dynamic linked library*) D2XX. A quel punto si può scrivere un programma C che usa la libreria per scambiare dati sul cavo.

Se serve un collegamento ancora più veloce si può usare il modulo FTDI UM232H mostrato nella **Figura e9.54**, che collega una porta USB del PC a un'interfaccia parallela sincrona operante fino a 40 MB/s.

9.7 ■ RIASSUNTO

Molti processori usano I/O mappato in memoria per comunicare con il mondo esterno. I microcontrollori offrono una gamma di periferiche di base, compresi I/O di uso generale, seriali, analogici, oltre a timer. I personal computer e i microcontrollori più evoluti supportano anche I/O più complessi come USB, Ethernet e SATA.

Questo capitolo ha mostrato numerosi esempi specifici di I/O utilizzando il Raspberry Pi. I progettisti di sistemi embedded si trovano continuamente di fronte a nuovi processori e nuove periferiche. Per semplici I/O nei sistemi embedded si procede consultando la documentazione tecnica per scoprire quali sono le periferiche disponibili, e quali contatti e registri di I/O mappati in memoria sono utilizzabili. A quel punto è relativamente semplice scrivere un driver che inizializza la periferica e trasmette o riceve dati.

Per standard più complessi come USB, scrivere un driver di dispositivo è un'attività molto specialistica, da lasciar fare agli esperti che abbiano una conoscenza approfondita del dispositivo e del protocollo USB. I progettisti occasionali si devono invece orientare su processori forniti di driver di dispositivi già collaudati e con esempi di codice per i dispositivi che interessa pilotare.