Università degli Studi di Padova
Facoltà di Ingegneria

Corso di Laurea Triennale
in
Ingegneria Meccanica e Meccatronica

Tesi di Laurea

Sistema di Acquisizione Dati per una Line-Scan-Camera del Veicolo Freescale-Cup

Laureando:
Franco Fusco

Relatore:
Ch.mo Prof.
Roberto Oboe

Dipartimento di Tecnica e Gestione dei Sistemi Industriali

Anno Accademico 2014-2015

A Nonna Maia

Indice

Capitolo 1

Introduzione

1.1 Freescale Cup

La Freescale Cup è una competizione nata nel 2003, organizzata dall'azienda *Freescale Semiconductors*, a cui l'Università di Padova ha partecipato, in cluso quest'anno, per tre volte.

La competizione coinvolge diversi team formati da tre studenti universitari, ai quali è richiesto di assemblare un piccolo veicolo e progettare un software che gli permetta di compiere un tracciato assegnato in maniera del tutto autonoma.

La macchina è dotata di due motori in corrente continua per la trazione e di un servomotore per lo sterzo. Su di essa possono essere montati diversi sensori, ad esempio telecamere per l'acquisizione di immagini o encoder per il calcolo della velocità. Per il controllo dei diversi dispositivi si utilizza un microcontrollore tra quelli prodotti dalla Freescale; questo dovrà occuparsi in generale di acquisire i dati dai sensori, elaborarli secondo particolari algoritmi di controllo e trasmettere dunque i segnali opportuni agli attuatori.

Per quanto concerne il tracciato, questo è costituito da un piano a fondo bianco con due linee nere laterali. La pista si compone di parti aventi diverse geometrie: vi sono tratti rettilinei, curve a gomito, dossi, dissuasori, incroci e curve alternate a raggio di curvatura ampio.

Tramite le telecamere, la macchina ha modo di stabilire la propria posizione relativamente al tracciato e di conseguenza di comandare gli attuatori in modo da non fuoriuscire dalla pista. Tra i diversi team, vince quello la cui macchina riesce a completare un giro nel tempo minore, senza uscire di strada.

La competizione permette dunque agli studenti di cimentarsi in un progetto multidisciplinare, i cui contenuti spaziano dalla teoria dei segnali all'informatica, dai controlli digitali all'elettronica analogica e digitale. L'approccio pratico permette poi di imparare ad applicare con elasticità gli strumenti acquisiti durante i corsi universitari, valutando di volta in volta quale particolare approccio adottare per la soluzione di un problema specifico.

1.2 Scopo dell'Elaborato

Durante lo sviluppo del progetto per la Freescale Cup si sono spesso riscontrate limitatezze nell'hardware messo a disposizione dal kit base della competizione. Uno dei principali problemi consiste nell'assenza di dispositivi di input/output adeguati per permettere alla scheda di dialogare con l'utente. Il programma utilizzato per la scrittura del codice, *Code Warrior*, mette a disposizione un tool per il debugging interattivo, tuttavia presenta come problema principale il fatto che il funzionamento di questo sistema richiede la presenza di un cavo di collegamento. Finché si lavora al banco, con la macchina ferma, non sorgono problemi (anche se il debugger non è dei più agevoli da usare). Nel momento in cui si vuole invece mettere la macchina in pista, diventa ostico riuscire a ottenere informazioni, in quanto bisognerebbe seguire la macchina con il computer in mano, al fine di non far scollegare il cavo.

Uno dei motivi per cui si sentiva la necessità di avere un sistema di comunicazione e acquisizione adeguato, risiede nel ruolo fondamentale giocato dalla telecamera. Questa è infatti molto sensibile alle variazioni di luminosità dell'ambiente, e richiede di impostare un parametro, una sorta di tempo di esposizione, in maniera adeguata in funzione della luce incidente al dispositivo. La taratura veniva fatta quasi sempre a mano appena entrati in laboratorio, e quando si riscontravano strani comportamenti del veicolo, si controllava con un oscilloscopio il segnale di uscita dal sensore. Più volte è capitato di scoprire che il tempo di esposizione precedentemente impostato non era più adeguato all'attuale grado di illuminazione, e ogni volta si rendeva necessario procedere per tentativi, fino a quando non si otteneva una lettura soddisfacente.

Per tutti questi motivi, e per altri, si è giunti alla conclusione che per poter lavorare in maniera efficiente con la scheda sarebbe stato necessario sviluppare un sistema che permettesse di interagire con la scheda accedendo alle variabili che essa elabora anche quando questa non è collegata al PC. In secondo luogo, le problematiche della telecamera richiedevano un sistema di acquisizione delle immagini che non richiedesse necessariamente un oscilloscopio.

Un'ulteriore problematica era costituita dalla scheda di controllo motori. Questa è stata progettata dalla stessa Freescale per potersi inserire sopra la Freedom Board, e integra un ponte H per la gestione dei motori. Purtroppo, questo shield di espasione, una volta montato, occupava tutti i pin disponibili del microcontrollore, impedendo l'aggiunta di hardware aggiuntivo.

Alla fine, quello che si è fatto è stato creare una cheda di espansione hardware da montare sulla Freedom Board. Questa scheda monta un display LCD tramite cui la scheda comunica dati all'utente, e rende inoltre disponibili tutti quei pin che non vengono utilizzati dalla scheda di controllo dei motori.

È inoltre presente una memoria di tipo Flash, utilizzata per salvare i dati acquisiti dalla telecamera. Quando il programma termina l'esecuzione, i dati salvati in tale memoria vengono estratti e caricati su una schedina SD. Per far fronte alla necessità di utilizzare un oscilloscopio, si è sviluppato un programma, scritto in linguaggio java, che legge i dati caricati sulla scheda SD e li mostra come se fosse il display di un oscilloscopio. Si sono inoltre sviluppate alcune librerie di utilità per l'elaborazione

e l'interpretazione delle immagini ottenute: in particolare si è definita una classe che definisce i numeri complessi (non presenti nelle librerie standard di java), e una seconda che permette di operare su array di numeri complessi trattandoli come funzioni, ad esempio calcolando la DFT mediante un algoritmo FFT.

1.3 Struttura dell'Elaborato

L'elaborato si compone, esclusa questa introduzione e la conclusione, di cinque capitoli, che ricalcano lo stesso flusso logico con cui si sono sviluppate le diverse parti del progetto.

Rispetto agli anni precedenti, la scheda di controllo del veicolo è cambiata: il capitolo 2 si focalizza sulla descrizione del microcontrollore utilizzato quest'anno, il *FRDM-KL25Z*. Si esporranno le caratteristiche generali di un microcontrollore, concentrandosi poi sui moduli impiegati per il funzionamento generale della macchina.

Nel capitolo 3 si approfondiscono invece le tematiche relative al sensore di visione, una telecamera da 1x128 pixel sensibile all'intensità luminosa, descrivendo brevemente le caratteristiche principali delle immagini acquisite. Verrà presentato il modulo *PIT* utilizzato per l'acquisizione delle immagini della telecamera. Si illustra infine l'insieme di funzioni che permettono di gestire il flusso di dati.

Il capitolo 4 tratta il protocollo di comunicazione *SPI* e il modulo che ne permette l'uso a bordo della scheda. Vengono spiegate le caratteristiche e il funzionamento del chip di memoria, l'integrato *W25Q80BV*, illustrando le librerie realizzate per le operazioni di lettura/scrittura delle celle di memoria.

La scheda di espansione viene descritta nel capitolo 5, spiegando le scelte che si sono attuate per la realizzazione della stessa. Si tratta inoltre il funzionamento del display LCD e del driver *HD44780U* che lo gestisce.

Il capitolo 6 tratta invece il software che permette di trasferire i dati dalla memoria Flash alla scheda SD. Viene quindi presentato il tool *FreescaleCup_Data_Analyzer*, un inseme di classi java utilizzate per la visualizzazione dei dati della telecamera. In conclusione, verrà mostrato come si sono utilizzati tali strumenti per poter simulare alcuni algoritmi di gestione della telecamera.

Capitolo 2

FRDM-KL25Z

2.1 Generalità

Il microcontrollore utilizzato per il controllo del veicolo e delle periferiche hardware è la scheda *FRDM-KL25Z* (anche nota come Freedom Board). Questa presenta alcune differenze significative rispetto al microcontrollore utilizzato negli scorsi anni, e cioè il *TRK MCB5604B* (questo è stato descritto da Giordano Lilli in [14]).

La scheda utilizzata quest'anno fa parte della serie Kinetis L, ed è stata progettata dalla *Freescale Semiconductors* come piattaforma adatta per la prototipazione veloce di progetti basati su microcontrollore. Tra le caratteristiche che la rendono adatta a tale scopo, si ricorda la presenza di alcuni componenti già montati su di essa, tra cui un LED RGB che risulta utile nelle primissime fasi di sviluppo di un progetto: è ad esempio possibile utilizzare i diversi colori emessi per monitorare lo stato di esecuzione del programma caricato nella scheda.

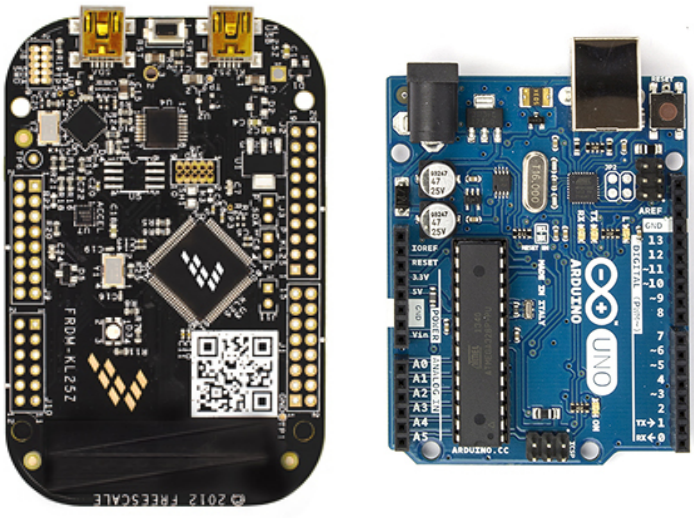
Un altro aspetto che rende la Freedom Board ideale per la prototipazione è la geometria con cui sono stati disposti i connettori pin-strip sulla scheda: essi sono del tutto compatibili con la piedinatura del microcontrollore *Arduino*[1], un progetto open-source divenuto popolare negli ultimi anni e per cui sono stati creati un gran numero di blocchi di espansione. La piedinatura della KL25Z permette così di utilizzare il materiale, di facile reperimento, progettato per Arduino. La figura 2.1 mostra i due microcontrollori dall'alto, permettendo così di apprezzare la similitudine dei contatti pin-strip.

2.1.1 Caratteristiche del Microcontrollore

Il microcontrollore montato nella scheda è il KL25Z; le caratteristiche sono le seguenti[2]:

[1] La scheda Arduino verrà presentata brevemente all'inizio del capitolo 6; per informazioni dettagliate si rimanda alla pagina principale del progetto: `https://www.arduino.cc/`

[2] Si è deciso di mantere tutte le voci in lingua inglese, così come riportate nel datasheet (cfr. [10]), poiché molte di esse non renderebbero se tradotte nel loro equivalente italiano)



(a) Freedom Board (b) Arduino Uno

Figura 2.1: Freedom Board e Arduino Uno a confronto

- Core: ARM Cortex-M0 a 32 bit
  - up to 48 MHz operation
  - single-cycle fast I/O access port
- Memory
  - 128KB flash
  - 16KB SRAM
- System integration
  - power management and mode controllers
  - low-leakage wakeup unit
  - bit manipulation engine for read-modify-write peripheral operations
  - Direct Memory Access (DMA) controller
  - Computer Operating Properly (COP) watchdog timer
- Clocks
  - clock generation module with FLL (Frequency-Locked-Loop) and PLL (Phase-Locked-Loop) for system and CPU clock generation
  - 4 MHz and 32 kHz internal reference clock
  - system oscillator supporting external crystal or resonator
  - low-power 1kHz RC oscillator for RTC (real time clock) and COP watchdog
- Analog peripherals

  - 16-bit SAR ADC w/ DMA support
  - 12-bit DAC w/ DMA support
  - high speed comparator
- Communication peripherals
  - two 8-bit Serial Peripheral Interfaces (SPI)
  - USB dual-role controller with built-in FS/LS transceiver
  - USB voltage regulator
  - two I^2C modules
  - one low-power UART and two standard UART modules
- Timers
  - one 6-channel Timer/PWM module
  - two 2-channel Timer/PWM modules
  - 2-channel Periodic Interrupt Timer (PIT)
  - real time clock (RTC)
  - low-power Timer (LPTMR)
  - system tick timer (SysTick)
- Human-Machine Interfaces (HMI)
  - general purpose input/output controller
  - capacitive touch sense input interface hardware module

2.1.2 Funzionamento di Base

Il primo e fondamentale passo da compiere per poter utilizzare la scheda è comprendere a fondo la struttura e il meccanismo di funzionamento di un microcontrollore. In generale, un *MCU* (*Micro-Controller Unit*) è costituito dall'insieme di:

- CPU
- memoria dati (RAM)
- memoria di programma (ROM, EPROM, FLASH)
- moduli/periferiche:
  - porte di I/O e GPIO (General Purpose Input/Output)
  - gestore di interrupt
  - timer e contatori
  - dispositivi per la conversione analogico-digitale (ADC e DAC)
  - moduli di comunicazione (SPI, I^2C, UART, . . .)

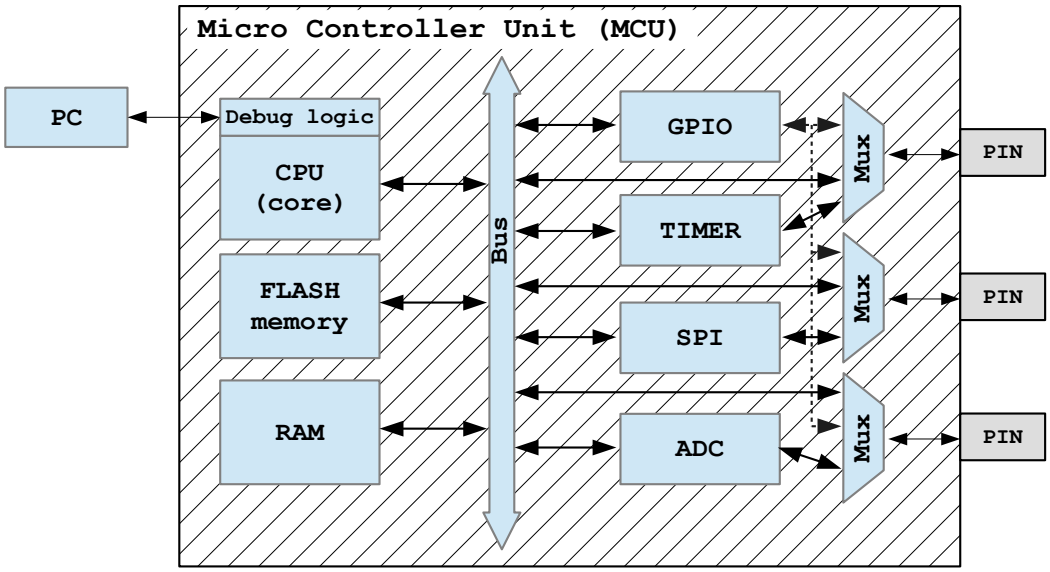


Figura 2.2: Schema semplificato di un microcontrollore

Tutti i suddetti componenti comunicano mediante uno o più bus. La figura 2.2 riporta lo schema funzionale di un generico microcontrollore.

In molti microcontrollori (il KL25Z fa parte di questi) il processore accede ai moduli tramite scritture/letture tramite il bus dati esattamente come se stesse interagendo con un'unità di memoria: tale metodo prende il nome di *Memory-Mapped I/O*. Ogni modulo ha una certa serie di indirizzi assegnati, a ciascuno dei quali corrisponde un registro dati a cui il processore può accedere. Per comunicare un'informazione, la CPU scrive ben determinati bit all'interno dei registri, che la periferica successivamente legge. Lo stesso fanno le periferiche per inviare a loro volta i dati alla CPU, che estrae le informazioni mediante un'operazione di lettura in memoria dal registro desiderato.

Le funzioni specifiche associate a ciascun registro si trovano nei manuali forniti dai costruttori; ad esempio nel caso della Freedom Boardsi legge in [8] che il registro $0\times4003B020$ (*ADC0_SC1A*, visibile in fig. 2.3) è utilizzato dall'ADC come *status and control register 1* (registro di stato e controllo 1), e cioè come registro per l'impostazione di alcuni parametri. Approfondendo la lettura di tale sezione, si trova scritto che ad esempio il settimo bit permette di abilitare gli interrupt dell'ADC: scrivendovi un *1* si attiva la generazione degli interrupt, mentre scrivendovi uno *0* le interruzioni vengono disabilitate.

A livello software i registri sono accessibili mediante delle MACRO dichiarate all'interno di un file fornito dal costruttore. Il meccanismo è molto semplice: si tratta ogni registro come se fosse una variabile intera, e se ne modifica il contenuto mediante l'operatore di assegnazione '='. Bisogna però fare attenzione a modificare esclusivamente i bit strettamente neessari, e non tutti quanti, utilizzando opportune mashere binarie. Come esempio, nel listato 2.1 sono riportate le istruzioni necessarie per abilitare o disabilitare gli interrupt dell'ADC.

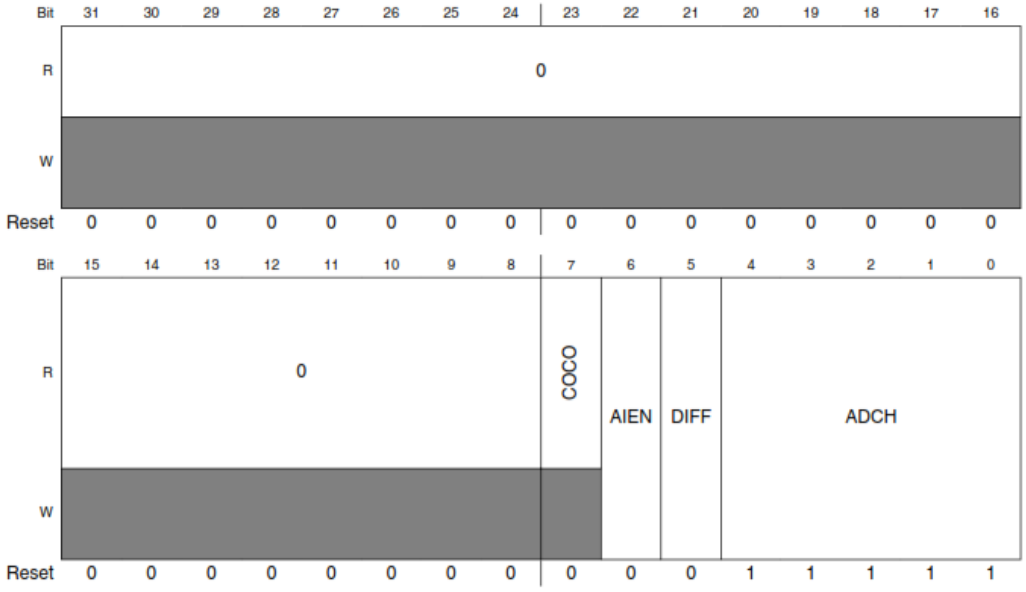


Figura 2.3: Registro ADC0_SC1A

Listato 2.1: Codice di esempio per ADC

```
1 ADC0_SC1A |= (1<<6); // abilita interrupt dell'ADC
2
3 ADC0_SC1A &= ~(1<<6); // disabilita interrupt dell'ADC
```

Nella quasi totalità dei casi, i componenti montati su un microcontrollore sono macchine sincrone, con ciò intendendo che per funzionare necessitano di un segnale di clock. Nei microcontrollori vi sono principalmente due tipi di clock: il *core clock*, avente frequenza F_{core}, alimenta la CPU e alcune periferiche, mentre il *bus clock*, avente frequenza $F_{bus} = 0.5F_{core}$, alimenta altre periferiche. Per impostare i segnali di clock in maniera appropriata bisogna scegliere la sorgente del clock e quindi impostare tutta una serie di parametri che vanno a dividere o a moltiplicare la frequenza base del clock di sistema. Nel microcontrollore KL25Z i diversi segnali di clock sono gestiti dal *Multipurpose Clock Generator Module* (brevemente *MCG*); la figura 2.4 mostra lo schema di tale modulo, mentre il listato 2.2 mostra come impostare il clock di sistema a 48MHz usando il *FLL*.

Listato 2.2: Set del clock di sistema

```
1 //imposta il core clock a 48MHz tramite FLL
2 // imposta il clock su una regolazione "precisa"
3 MCG_C4 |= 1<<MCG_C4_DMX32_SHIFT;
4 // imposta il clock per una frequenza medio-bassa
5 MCG_C4 |= MCG_C4_DRST_DRS(01);
```

Spesso le diverse periferiche sono progettate per interfacciarsi con hardware esterno (ad esempio sensori, schede di controllo per attuatori, PC, . . .) e pertanto necessitano di un'interfaccia di collegamento fisica: tale interfaccia prende il nome di *pin*. Le periferchie non sono tuttavia direttamente collegate ai pin, piuttosto sono collegate a dei *multiplexer* (più semplicemente *mux*). Questi dispositivi permettono di collegare un pin a uno a scelta tra diversi ingressi del mux.

Nel caso del KL25Z l'interfacciamento tra CPU e mux non è diretto, ma viene gestito dal modulo intermedio *PORT*. I registri di quest'ultimo permettono di selezionare per ogni porta la periferica da associare al pin fisico della scheda[3]. Le diverse funzioni associabili a ciascun pin della Freedom Board sono riportate in [8, 7]. Ai registri del modulo PORT si accede tramite Memory-Mapping, come per tutte le periferiche della scheda. Il listato 2.3 mostra ad esempio come selezionare per il pin 33 della Freedom Board (connesso a PORT_A13) la funzione GPIO e per il pin 35 la funzione *SPI0-SCK*.

Listato 2.3: Esempio di utilizzo del modulo PORT

```
1 // imposta PTA13 come GPIO
2 PORTA_PCR13 = (PORTA_PCR13 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
3 // imposta PTA15 come SPI0-SCK
4 PORTA_PCR15 = (PORTA_PCR15 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(2);
```

Le operazioni fin ora descritte da sole non bastano ad abilitare le periferiche della scheda. Per ridurre il consumo energetico e la potenza disspiata, i diversi moduli vengono lasciati inattivi anche dopo l'accensione della Freedom Board. Per poterli utilizzare bisogna pertanto abilitare l'alimentazione tramite la scrittura in un appo-

[3] Esistono in realtà delle porte non associate a un pin fisico.

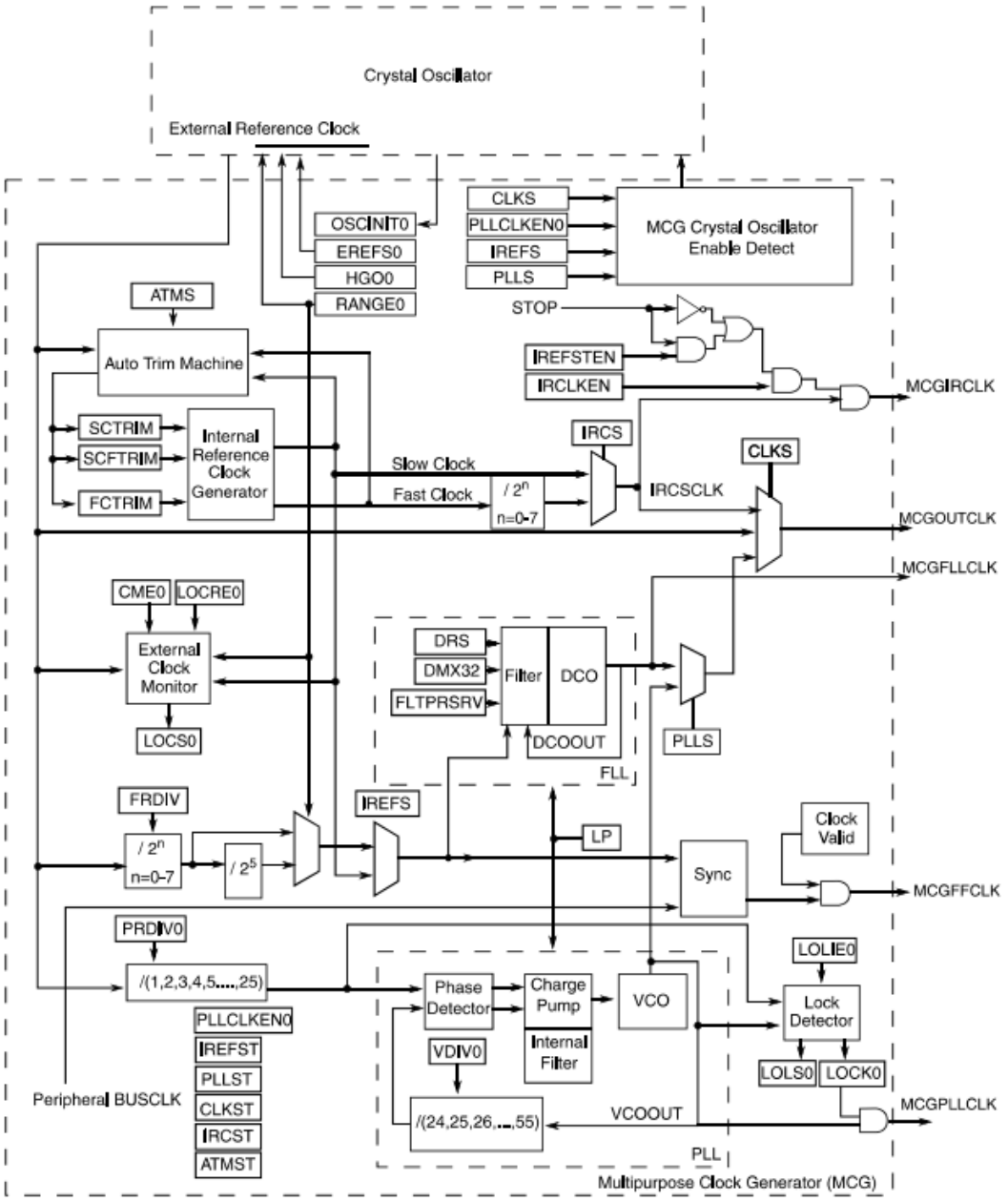


Figura 2.4: Schema del Multipurpose Clock Generator (MCG)

sito registro contenuto nel *System Integration Module* (*SIM*). A tale scopo è necessario editare un codice simile a quello contenuto in 2.4.

Listato 2.4: Abilitazione delle porte C mediante il SIM

```
1 // fornisce il clock a PORTC
2 SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK;
```

2.2 General Purpose Input/Output (GPIO)

Tra le molte periferiche presenti nella Freedom Board, si vuole presentare il modulo *General Purpose Input/Output* (*GPIO*). Questo è di fondamentale importanza in generale in un qualsiasi progetto a microcontrollore e in particolare per il progetto descritto in questo elborato. Per avere un'idea dell'importanza di tale componente, si consideri che, fatta eccezione per i pin inerenti alle funzioni di alimentazione, ogni pin della Freedom Board ha la possibilità di essere collegato tramite mux alla funzione GPIO.

Si è anticipato che un MCU può interagire con hardware esterno tramite i pin. Questo vuol dire che un pin potrebbe, ad esempio, essere collegato a un pulsante di cui si vuole "leggere" lo stato (chiuso o aperto), mentre un secondo pin potrebbe invece essere collegato ad un LED, da accendere o spegnere a seconda delle necessità. Per svolgere queste funzioni, entrambi i suddetti pin possono essere collegati al modulo GPIO, che fornisce la base per la "scrittura" e la "lettura" di segnali digitali.

Il funzionamento di questo dispositivo è abbastanza semplice. Innanzitutto un pin può essere impostato in modalità scrittura (output) oppure in modalità lettura (input). Nel primo caso il pin può assumere a piacimento un livello logico *alto*[4], portando il proprio livello di tensione approssimativamente alla tensione di alimentazione[5], oppure un livello logico *basso*[6], che genera in uscita una tensione approssimativamente pari a $0V$. In alternativa il pin, se posto in lettura, può misurare la tensione presente sul pin e stabilire se questa è *HIGH*, cioè vicina a V_{DD}, oppure *LOW* (vicina a 0V)[7].

Per impostare gli stati dei pin si usano alcuni registri (cfr. tabella 2.1). Questi registri sono raggruppati per porte: un gruppo di registri gestisce la funzionalità GPIO per tutte le porte A, un secondo gruppo fa lo stesso con le porte B, e cosìvia. Ogni registro di questi gruppi ha dimensione pari a 32bit, e ciascuno di essi è associato a un pin[8].

[4]Termini equivalenti sono "HIGH" oppure "asserted".

[5]In [11] viene riportato che un livello logico alto in scrittura corrisponde a una tensione compresa tra $V_{DD}-0,5V$ e V_{DD}(con V_{DD}corrispondente alla tensione di alimentazione), mentre un ivello logico basso genera in uscita una tensione compresa tra 0 e $0,5V$.

[6]Termini equivalenti sono "LOW" oppure "deasserted".

[7]Nel datasheet viene dichiarato che una tensione è considerata *HIGH* se superiore a $0,7\times V_{DD}$, mentre è considerata *LOW* se inferiore a $0,35\times V_{DD}$

[8]In realtà sulla scheda non tutte le porte sono associate a un pin, e perciò bisogna aver cura di modificare solamente i bit associati a pin fisicamente presenti sulla scheda.

Nome del registro	Accesso
GPIOx_PDOR Port Data Output Register	R/W
GPIOx_PSOR Port Set Output Register	W
GPIOx_PCOR Port Clear Output Register	W
GPIOx_PTOR Port Toggle Output Register	W
GPIOx_PDIR Port Data Input Register	R
GPIOx_PDDR Port Data Direction Register	R/W

Tabella 2.1: Registri GPIO. La lettera "x" va sostituita, nella pratica, con quella tipica del gruppo di porte (A, B, C, D oppure E).

Per impostare la modalità di un pin (input/output) si utilizzano i registri del tipo *GPIOx_PDDR*, in cui l'acronimo *PDDR* significa *Port Data Direction Register*. Un bit posto a 1 corrisponde a impostare il corrispondente pin in scrittura, viceversa un bit posto a 0 porta il pin in lettura.

Per portare un pin impostato in output a livello alto, si usano i registri *GPIOx_PSOR* (*Port Set Output Register*): quando un bit viene posto a 1, il corrispondente pin viene portato a V_{DD}(se non lo è già), mentre la scrittura di uno 0 non modifica l'attuale stato dei relativi pin. Appena completata l'operazione, il registro viene pulito, in modo che sia possibile utilizzarlo nuovamente in futuro. Allo stesso modo funzionano i registri *GPIOx_PCOR* (*Port Clear Output Register*, porta a 0V i pin selezionati) e *GPIOx PTOR* (*Port Toggle Output Register*, commuta lo stato dei pin indicati).

I registri del tipo *GPIOx_PDOR*, (*Port Data Output Register*), permettono di impostare manualmente lo stato dei pin: scrivendo uno 0 nel registro si porta a terra il pin corrispondente, mentre scrivendo un 1 si impone su di esso la tensione V_{DD}.

Infine, per i pin impostati in input, si può compiere l'operazione di lettura mediante i registri *GPIOx_PDIR* (*Port Data Input Register*). Un bit pari a 1 indica che sul corrispondente terminale è presente una tensione approssimativamente pari a V_{DD}, viceversa se un bit è pari a 0 significa che il pin relativo è stato portato a 0V.

Si è detto all'inizio del presente capitolo che la Freedom Board monta un LED RGB integrato. Come si può osservare nello schema elettrico della scheda (cfr. [9]), i tre LED "elementari" rosso, verde e blu sono rispettivamente collegati alle porte PTB18, PTB19 e PTD1. Il codice contenuto in 2.5 contiene le istruzioni necessarie a accendere i tre LED per ottenere una luce bianca. Si noti che per accendere un LED il corrispondente pin deve essere portato al valore logico 0.

2.3 Hardware-Interrupt

Quando si sviluppa un progetto a microcontrollore, ci si ritrova spesso a dover interrogare le periferiche (sia interne che esterne) sul loro stato. Si pensi ad esempio di avere 3 pulsanti di cui bisogna monitorare lo stato, poiché alla loro pressione si intende far succedere qualcosa. La prima possibilità che si potrebbe prendere in considerazione è quella di sondare a turno i tre dispositivi, fino a quando non viene rilevata la pressione di uno di questi. Questo approccio viene definito *polling*

Listato 2.5: Accensione del LED RGB integrato nella Freedom Board.

```
// macro utili per l'utilizzo delle funzioni GPIO
#define RED_ON GPIOB_PCOR |= 1<<18
#define GREEN_ON GPIOB_PCOR |= 1<<19
#define BLUE_ON GPIOD_PCOR |= 1<<1

void main() {
  ...
  // inizializza le porte per il LED RGB

  // clock alle porte dei LED
  SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK | SIM_SCGC5_PORTD_MASK;

  // GPIO LED rosso
  PORTB_PCR18 = (PORTB_PCR18 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);

  // GPIO LED verde
  PORTB_PCR19 = (PORTB_PCR19 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);

  // GPIO LED blu
  PORTD_PCR1 = (PORTD_PCR1 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);

  // imposta le porte dei led rosso e verde in output
  GPIOB_PDDR |= 1<<18 | 1<<19;

  // imposta la porta del led blu in output
  GPIOD_PDDR |= 1<<1;

  // accende i tre LED
  RED_ON;
  GREEN_ON;
  BLUE_ON;

  ...
}
```

(traducibile in italiano come "sondaggio") e costituisce uno dei meccanismi basilari di gestione, da parte della CPU, delle periferiche esterne.

Un apparente vantaggio del polling è la sua semplicità realizzativa, in quanto a livello di codice non richiede istruzioni complicate. Questa è tuttavia una semplicità apparente, in quanto in progetti complessi il codice può finire col diventare di difficile lettura.

Oltre a questo, vi sono svantaggi rilevanti in termini di prestazioni. Si pensi ad esempio al funzionamento del veicolo da realizzare per la Freescale Cup: il veicolo si trova a dover interagire con diversi dispositivi di input, principalmente due telecamere, due encoder, due potenziometri e due pulsanti, per un totale di 8 periferiche. Innanzitutto va osservato che le telecamere e gli encoder funzionano a frequenze dell'ordine del kHz e oltre, mentre i pulsanti e i potenziometri, in quanto azionati manualmente, aggiornano il loro stato con una frequenza difficilmente superiore a 5Hz. In questo modo, poiché il processore interroga ciclicamente tutti i dispositivi, in una decina di secondi si faranno migliaia di letture dei pulsanti e dei potenziometri "a vuoto". Tutto questo, si capisce, costituisce una perdita di tempo che potrebbe essere utilizzato per eseguire altri compiti.

Un secondo problema, ancora più rilevante, deriva dal fatto che alcuni "eventi" possono verificarsi quando il processore è impegnato a interrogare un altro dispositivo. In tal caso c'è il rischio che l'informazione venga persa, ad esempio alcune letture della telecamera potrebbero essere ignorate, e ne risulterebbe un'immagine distorta. Ancora peggio, una periferica pronta potrebbe causare l'avvio di una routine dispendiosa in termini di tempo, portando a ignorare completamente le altre periferiche per un tempo considerevole, una situazione incompatibile con un qualunque sistema real-time.

Per risolvere i sopracitati problemi e alleggerire la complessità del codice si possono utilizzare gli *interrupt* (interruzioni). Questi sono segnali asincroni[9] che informano il processore che una periferica deve interloquire con la CPU. Ad esempio un ADC potrebbe effettuare una conversione e poi generare un interrupt, per informare il processore che la conversione è completa.

Gli interrupt si dividono in *interrupt software* e *interrupt hardware*. I primi corrispondono di fatto a istruzioni software, che permettono di fermare l'attuale esecuzione per lanciare dei sottoprogrammi. I secondi sono invece utilizzati dalle periferiche per arrestare l'esecuzione del programma principale ed eseguire un sottoprogramma specifico associato a tale dispositivo.

Per quanto riguarda la gestione degli interrupt da parte della Freedom Board, la scheda ricorre a un *Nested Vectored Interrupt Controller* (*NVIC*). Gli interrupt generati dalle diverse periferiche vengono ricevuti dal NVIC, che provvede a comunicare alla CPU di congelare l'esecuzione per gestire l'evento verificatosi. Per funzionare correttamente, il programma deve definire un gestore degli interrupt, e cioè una funzione che venga chiamata ogni volta che si verifica una richiesta di attenzione da parte di una periferica.

[9] Possono cioè verificarsi in qualunque momento.

Tabella 2.2: Nested Vectored Interrupt Controller

Vector	IRQ	IPR	Source	Description
0	–	–	ARM core	Initial Stack Pointer
1	–	–	ARM core	Initial Program Counter
2	–	–	ARM core	Non-maskable Interrupt
3	–	–	ARM core	Hard Fault
4	–	–	–	–
5	–	–	–	–
6	–	–	–	–
7	–	–	–	–
8	–	–	–	–
9	–	–	–	–
10	–	–	–	–
11	–	–	ARM core	Supervisor Call
12	–	–	–	–
13	–	–	–	–
14	–	–	ARM core	Pendable Request for System Service
15	–	–	ARM core	System Tick Timer
16	0	0	DMA	DMA channel 0 transfer complete and error
17	1	0	DMA	DMA channel 1 transfer complete and error
18	2	0	DMA	DMA channel 2 transfer complete and error
19	3	0	DMA	DMA channel 3 transfer complete and error
20	4	1	–	–
21	5	1	FTFA	Command complete and read collision
22	6	1	PMC	Low-voltage detect, low-voltage warning
23	7	1	LLWU	Low Leakage Wakeup
24	8	2	I^2C	
25	9	2	I^2C	
26	10	2	SPI0	Single interrupt vector for all sources
27	11	2	SPI1	Single interrupt vector for all sources
28	12	3	UART0	Status and error
29	13	3	UART1	Status and error
30	14	3	UART2	Status and error
31	15	4	ADC0	
32	16	4	CMP0	

continua nella pagina successiva

continua dalla pagina precedente

Vector	IRQ	IPR	Source	Description
33	17	4	TPM0	
34	18	4	TPM1	
35	19	4	TPM2	
36	20	5	RTC	Alarm interrupt
37	21	5	RTC	Seconds interrupt
38	22	5	PIT	Single interrupt vector for all channels
39	23	5	–	–
40	24	6	USB OTG	
41	25	6	DAC0	
42	26	6	TSI0	
43	27	6	MCG	
44	28	7	LPTMR0	
45	29	7	–	
46	30	7	Port control module	Pin detect (Port A)
47	31	7	Port control module	Pin detect (Port D)

Può accadere che, mentre si sta eseguendo la funzione associata a un determinato interrupt, una seconda periferica faccia richiesta di attenzione da parte della CPU. Per risolvere queste situazioni, a ogni interrupt è associata una priorità: se vengono generati più interrupt, questi vengono gestiti in ordine in accordo con la loro priorità.

Per concludere la descrizione delle interruzioni, si mostra nel listato 2.6 come impostare il modulo ADC in modo che notifichi al processore l'avvenuta conversione tramite la generazione di un interrupt.

Listato 2.6: Interrupt con ADC

```
1  void setupADC() {
2    // abilita il clock per ADC
3    SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;
4    // abilita gli interrupt
5    NVIC_ISER |= NVIC_ISER_SETENA(1<<15);
6  }
7
8  void ADC0_IRQHandler() {
9    /*
10    * Qui si puo' inserire il codice con le istruzioni
11    * da eseguire quando l'ADC genera un interrupt
12    */
13 }
```

Nel caso della Freedom Board, per definire la funzione di gestione degli interrupt di un determinato modulo i programmatori della freescale hanno definito un gestore

di default che interrompe l'esecuzione del codice. Per ogni periferica viene dunque definito un gestore di interruzioni specifico, a cui si associa, inizialmente, il gestore di default. Tale gestore viene però definito `weak` ("debole"), con ciò intendendo che se all'interno del codice il compilatore dovesse trovare una funzione avente lo stesso nome della precedente, il gestore di default verrebbe sostituito con la nuova definzione. A titolo di esempio, la dichiarazione del gestore degli interrupt dell'ADC è riportato nel listato 2.7; la funzione `ADC_IRQHandler()` dichiarata in 2.6 sovrascrive il gestore di default pre-esistente.

Listato 2.7: Defalut Interrupt Handler

```
1 void Default_Handler() {
2   __asm("bkpt");
3 }
4
5 void ADC0_IRQHandler() __attribute__ ((weak, alias("Default_Handler")));
```

2.4 Systick

In molte applicazioni elettroniche, risulta importante riuscire a mettere in pausa per un determinato tempo l'esecuzione del programma; per realizzare una tale funzione risulta indispensabile avere a disposizione un sistema che funzioni da "orologio". Nella Freedom Board è effettivamente presente un timer interno[10], pensato però per contare tempi dell'ordine dei secondi, quando in molte applicazioni è necessario poter gestire tempi molto più ridotti (dell'ordine dei millisecondi o anche meno).

Un primo metodo basilare che permette di mettere in pausa l'esecuzione consiste nell'inserire un ciclo che si ripeta per un certo numero di volte. Più alto è il numero di iterazioni, maggiore sarà il tempo necessario a completarlo. Questo sistema non permette però di avere un controllo preciso del tempo, poichè a priori non si sa esattamente quanto tempo corrisponde a un particolare numero di cicli.

Si potrebbe pensare di misurare sperimentalmente il tempo associato a diverse iterazioni, in modo da poter ottenere una relazione con cui definire un metodo che avvii un ciclo di lunghezza appropriata a seconda del tempo di pausa desiderato.

Questo approccio presenta però due svantaggi non indifferenti. Innanzitutto, la misura dei tempi per la determinazione della relazione sperimentale *cicli–tempo* potrebbe essere attuata tramite un cronometro azionato manualmente oppure con l'impiego di una strumentazione elettronica dedicata. Nel primo caso, le misure introdurrebbero errori dovuti alle imprecisioni dell'operatore umano e inoltre verrebbero fatte su tempi dell'ordine dei secondi, senza poter ottenere informazioni precise sui tempi brevi (intorno ai millisecondi). D'altro canto, l'impiego di un apparato elettronico dedicato complicherebbe il progetto senza motivo.

In secondo luogo, bisogna tenere conto del problema, assai più grave, costituito dalla presenza degli interrupt. Si immagini dunque che una periferica possa gene-

[10]Chiamato *RTC*, *Real Time Clock*

rare degli interrupt, che tipicamente saranno casualmente distribuiti nel tempo. Si supponga di aver scritto una funzione che, una volta chiamata, esegua un ciclo per tutto il tempo richiesto. Ora si pensi a questa situazione: la funzione viene chiamata in modo che termini dopo un certo tempo T_0, ma, dopo un intervallo temporale t_1, la periferica genera un interrupt. Appena possibile, la CPU interrompe l'esecuzione della funzione di pausa, ed esegue invece il codice associato all'interrupt, impiegando un tempo pari a t_{int}. L'esecuzione della funzione iniziale viene ripresa, e dopo un tempo $t_2 = T_0 - t_1$ può infine terminare. In totale, a causa della presenza dell'interrupt, il programma principale rimane inattivo per $t_1 + t_{int} + t_2 = T_0 + t_{int}$. La situazione potrebbe essere anche peggiore se più di un interrupt venisse generato durante l'esecuzione della funzione di pausa.

Un metodo più efficiente consiste invece nello sfruttare gli interrupt stessi per realizzare la funzione desiderata. Determinate periferiche possono infatti generare interrupt a intervalli regolari, e questo permette di creare un rudimentale orologio di sistema.

Il *System Tick Timer*, o brevemente *Systick*, è un modulo integrato nel processore che permette di generare interrupt periodici. Questo ha un contatore interno che parte da un determinato valore e scende progressivamente fino a raggiungere lo zero. La frequenza di conteggio del Systick è la stessa di funzionamento del processore. Quando il contatore passa dal valore 1 allo 0, viene generato un interrupt, mentre il contatore viene ripristinato al valore originario.

Il Systick ha tre registri per il controllo del suo funzionamento: il *Reload Value Register* (*RVR*), il *Current Value Register* (*CVR*) e il *Control and Status Register* (*CSR*). Il primo di questi contiene il valore che deve essere caricato nel contatore ogni volta che si raggiunge lo zero. Il secondo contiene invece l'attuale valore del contatore.

Il CSR contiene 3 bit di impostazione e un bit per la lettura dello stato. Il primo dei tre bit è detto *CLKSOURCE*, e permette di selezionare la sorgente del clock del Systick. Il secondo bit, *TICKINT*, permette di attivare o meno la generazione degli interrupt. Il bit *ENABLE* abilita o disabilita il conteggio. L'ultimo bit è chiamato *COUNTFLAG* e ha come scopo quello di indicare se, dall'ultima lettura, si è verificata una transizione da 1 a 0.

La soluzione che si è deciso di adottare consiste nell'impostare i registri del Systick in modo che venga generato un interrupt ogni $20\mu s$ (cioè con una frequenza pari a $50kHz$). Il Systick utilizza come segnale di clock il core clock, e pertanto per poter generare interrupt alla frequenza desiderata è necessario caricare nel RVR un opportuno numero di conteggi da eseguire prima della generazione del segnale di interruzione. Considerato che il contatore viene incrementato alla frequenza del clock di sistema (core clock), si ha un conteggio ogni $T_{core} = F_{core}^{-1}$. Dopo n_{RVR} conteggi si vuole la generazione dell'interrupt, e cioè $n_{RVR} T_{core} = T_{interrupt}$, avendo indicato con $T_{interrupt}$ il tempo che deve intercorrere tra un interrupt e il successivo. Tale relazione porge: $n_{RVR} = T_{interrupt}/T_{ccore}$, o equivalentemente in termini frequenziali: $n_{RVR} = F_{core}/F_{Systick}$.

Al fine di poter tenere traccia del tempo che trascorre, si è ricorso a una variabile intera chiamata `tick_timer`. Tale valore è dichiarato di tipo `volatile unsigned int`. L'opzione `unsigned` comunica al compilatore che il valore del contatore non

rappresenta mai numeri negativi, e in pratica permette di raddoppiare il valore del massimo intero rappresentabile: una variabile di tipo `int` a 32 bit può contenere valori compresi nell'intervallo $\left[-2^{31}, 2^{31}-1\right]$, mentre una variabile `unsigned int` ha campo di esistenza pari a $\left[0, 2^{32}-1\right]$.

Il tipo `volatile` è invece necessario per forzare la CPU a permettere sempre l'aggiornamento della variabile. Il linguaggio di programmazione C è infatti progettato per ottimizzare il tempo di esecuzione quando possibile: uno dei meccanismi a cui ricorre è quello di individuare segmenti di codice in cui una variabile non viene mai aggiornata, e in tale sequenza tratta il valore al pari di una costante numerica. A titolo di esempio si faccia attenzione alle istruzioni contenute in 2.8. Normalmente, ci si aspetta che, se durante l'esecuzione del ciclo `for` la funzione `interruptFunction()` viene chiamata almeno una volta, il contenuto della variabile `z` alla fine del ciclo sia maggiore di 10. Tuttavia quando inizia l'esecuzione del ciclo, la CPU riconosce che, all'interno di tale blocco di istruzioni, `x` non viene mai aggiornata e decide dunque di considerarla costante. Ne consegue che, anche se durante l'esecuzione del `for` la funzione di interrupt venisse eseguita più volte, il valore di `x` non verrebbe aggiornato nel ciclo, e `z` varrebbe 10. Per risolvere il problema, basta dichiarare la variabile `x` come `volatile int` piuttosto che solamente `int`. In tal caso, il compilatore non farà nessun tipo di assunzione relativamente all'aggiornamento della variabile, e il codice potrà essere eseguito normalmente.

Listato 2.8: Comportamento di variabili non volatili

```
1  int x = 1;
2
3  void main() {
4    int i,z=0;
5    for(i=0; i<10; i++)
6      z+=x;
7    ...
8  }
9
10 void interruptFunction() {
11   x++;
12 }
```

All'interno del codice di gestione del Systick si è quindi definito il gestore degli interrupt in modo che questo incrementi di una unità il valore del `tick_timer`. Il timer inizia il suo aggiornamento praticamente appena avviata la scheda e l'incremento viene eseguito periodicamente, risolvendo il problema dell'orologio. Un vantaggio di questo meccanismo è che il Systick è un modulo appartenente al core, e quindi i suoi interrupt hanno la precedenza su quelli delle periferiche esterne: il conteggio non è quindi praticamente affetto da errori dovuti alla generazione di interruzioni esterne.

L'ultimo passo consiste nell'implementare la funzione di pausa (cfr. listato 2.9). Il concetto che vi sta alla base è estremamente semplice: si indica alla funzione il tempo di pausa desiderato, quindi il metodo salva in una variabile `t_start` l'istante

in cui inizia la propria esecuzione copiando il contenuto del tick_timer. Viene ora avviato un ciclo che continua a controllare il valore del timer implementato: quando il lasso di tempo trascorso eccede il tempo di pausa indicato, il ciclo termina e si esce dalla funzione. Come già evidenziato, questo metodo è abbastanza efficiente anche nel caso in cui si verifichino interrupt esterni. Questo perché il timer del sistema si aggiorna comunque, e pertanto si ha modo di verificare il tempo effettivamente trascorso indipendentemente da quante interruzioni si siano verificate.

Listato 2.9: Setup del Systick e funzione delay

```
// timer di sistema
volatile unsigned int tick_timer = 0;

void setupSystick() {
  // abilita il SYSTICK
  tick_timer = 0;
  SYST_RVR = CORE_CLOCK_FREQUENCY/SYSTICK_FREQUENCY;
  SYST_CVR = 0;
  SYST_CSR = SysTick_CSR_ENABLE_MASK | SysTick_CSR_TICKINT_MASK |
      SysTick_CSR_CLKSOURCE_MASK;
}

// sovrascrive il gestore di default
void SysTick_Handler() {
  // attenzione: il tick timer raggiunge il valore 0xFFFFFFFF
  // circa dopo 85899s (23h 51' 39")
  if(tick_timer < 0xffffffff)
    tick_timer++;
}

// mette in pausa l'esecuzione per ms millisecondi
void delay(unsigned int ms) {
  int t_start = tick_timer;
  while(tick_timer-t_start <= ms*SYSTICK_FREQUENCY/1000);
}
```

2.5 TPM

Durante il tracciato, è opportuno che il veicolo sia in grado di regolare la velocità di rotazione delle ruote motrici. Questo innanzitutto perché, durante una curva, la velocità della ruota interna e di quella esterna sono differenti, in quanto è diverso il raggio di curvatura. Se non si tenesse conto di questo fatto, una delle due ruote finirebbe con lo slittare, riducendo la stabilità del veicolo, oppure risulterebbe frenata dalla forza di attrito, sprecando energia inutilmente.

Vi è poi un secondo motivo, in un certo senso più raffinato, per cui il controllo della rotazione del motore è necessario: a seconda del tipo di ostacolo che si trova davanti al veicolo si può cercare di impostare un tipo di guida appropriato. Ad esempio, se ci si trova su un rettilineo è possibile portare la macchina alla velocità

massima, mentre quando si incontra una curva conviene frenare leggermente e poi mantenere costante la velocità. Ancora, ci si potrebbe trovare di fronte a un dosso: nella prima parte è necessario dare maggiore potenza ai motori per contrastare la gravità, mentre nella seconda conviene limitare la corrente assorbita dal motore, per evitare di acquistare troppa velocità nella discesa.

Inoltre, congiuntamente all'uso di encoder rotativi o altri trasduttori di posizione o velocità angolare, la modulazione della rotazione permette di creare algoritmi di controllo che garantiscano elevate prestazioni per il funzionamento ottimale delle ruote motrici (ad esempio si possono progettare rampe di accelerazione che non portano il motore a slittare in partenza, oppure si possono creare algoritmi in grado di rilevare lo slittamento delle ruote e di impedirlo manipolando la velocità opportunamente).

Il microcontrollore montato sul veicolo deve coordinare infine il controllo di trazione con il funzionamento del servomotore che pilota il meccanismo di sterzo della macchina.

Per tutte queste necessità, inclusa la gestione degli encoder, è possibile fare uso dei *Timer PWM Modules* (*TPM*). Questi moduli mettono a disposizione diversi contatori che permettono di interagire, mediante pin dedicati, con dispositivi esterni. Nel caso specifico della Freescale Cup, sono stati usati per generare i segnali PWM necessari al funzionamento dei motori e per contare il numero di rotazioni degli encoder rotativi montati sulle ruote motrici. Di seguito si spiegherà il funzionamento di questi moduli, concentrandosi sui segnali di interesse. Non si spiegherà invece la struttura e il funzionamento nel dettaglio dei motori in corrente continua e del servomotore[11].

Si vuole ora mostrare la struttura di un modulo TPM. Si faccia riferimento alla figura 2.5 che contiene lo schema di un signolo modulo e alla tabella 2.3, in cui sono riportati invece i registri della periferica.

Tabella 2.3: Registri TPM

Registro/bit	Descrizione
TPMx_SC	**Status and Control Register, contiene i bit usati per configurare le principali impostazioni dei TPM.**
DMA [8]	DMA Enable, abilita il trasferimento dati tramite DMA quando viene settato il bit TOF.
TOF [7]	Timer Overflow Flag, bit che viene settato quando il contatore del TPM raggiunge il valore contenuto nel registro MOD. Per pulire il bit bisogna scrivere un 1.
TOIE [6]	Timer Overflow Interrupt Enable, abilita la generazione degli interrupt quando TOF viene settato.

continua nella pagina successiva

[11] Per approfondimenti in merito si rimanda alla tesi di laurea di un altro studente che ha affrontato esaustivamente tali tematiche [14].

continua dalla pagina precedente

Registro/bit	Descrizione
CPWMS [5]	Center-aligned PWM Select, abilita il conteggio in avanti e indietro.
CMOD [4-3]	Clock Mode Selection, seleziona la sorgente per i conteggi. [CMOD]=00 indica che il timer è disabilitato.
PS [2-0]	Prescale Factor Selection, permette di impostare il fattore di divisione per il divisore di frequenza.
TPMx_CNT	**Counter Register, contiene il valore del contatore del TPM.**
COUNT [15-0]	L'attuale valore del contatore.
TPMx_MOD	**Modulo Register, contiene il modulo del contatore del TPM.**
MOD [15-0]	Modulo impostato per il contatore.
TPMx_CnSC	**Channel (n) Status and Control Register, contiene i bit usati per configurare le principali impostazioni dei diversi canali del TPM.**
CHF [7]	Channel Flag, bit che viene settato quando avviene un evento per il canale. Per pulire il bit bisogna scrivere un 1.
CHIE [6]	Channel Interrupt Enable, abilita la generazione di interrupt quando CHF viene settato.
MSB [5]	Channel Mode Select, usato per selezionare la modalità di funzionamento del canale.
MSA [4]	Channel Mode Select, usato per selezionare la modalità di funzionamento del canale.
ELSB [3]	Edge or Level Select, usato per impostare diverse opzioni relativamente alla modalità selezionata.
ELSA [2]	Edge or Level Select, usato per impostare diverse opzioni relativamente alla modalità selezionata.
DMA [0]	DMA Enable, abilita il trasferimento dati tramite DMA quando viene settato il bit CHF.
TPMx_CnV	**Channel (n) Value Register, contiene il valore di riferimento per gli eventi di canale.**
VAL [15-0]	Channel Value, viene usato come valore di confronto per le modalità PWM e output compare, oppure come valore catturato in modalità input capture.
TPMx_STATUS	**Capture and Compare Status Register, contiene una copia di tutti i bit di stato del modulo.**
TOF [8]	Timer Overflow Flag, copia del bit TOF nel registro TPMx_SC.
CH5F [5]	Channel Flag, copia del bit CHF nel registro TPMx_C5SC.

continua nella pagina successiva

continua dalla pagina precedente

Registro/bit	Descrizione
CH4F [4]	Channel Flag, copia del bit CHF nel registro TPMx_C4SC.
CH3F [3]	Channel Flag, copia del bit CHF nel registro TPMx_C3SC.
CH2F [2]	Channel Flag, copia del bit CHF nel registro TPMx_C2SC.
CH1F [1]	Channel Flag, copia del bit CHF nel registro TPMx_C1SC.
CH0F [0]	Channel Flag, copia del bit CHF nel registro TPMx_C0SC.
TPMx_CONF	**Configuration Register, configura il comportamento del modulo quando la scheda opera in modalità debug o attesa.**
TRGSEL [27-24]	Trigger Select, seleziona il trigger di ingresso per iniziare il conteggio e/o resettare il contatore.
CROT [18]	Counter Reload On Trigger, se è settato, il contatore viene azzerato ogni volta che viene rilevato un fronte di salita del segnale di trigger selezionato.
CSOO [17]	Counter Stop On Overflow, disabilita il conteggio quando il contatore raggiunge il modulo impostato. Il conteggio deve essere riabilitato manualmente, oppure tramite il trigger quando CROT è settato.
CSOT [16]	Counter Start On Trigger, forza il contatore a non iniziare il conteggio quando viene abilitato finché non si verifica un evento di trigger.
GTBEEN [9]	Global Time Base Enable, configura il modulo per utilizzare un contatore esterno per il funzionamento dei canali.
DBGMODE [7-6]	Debug Mode, seleziona il comportamento che deve avere il modulo quando la scheda entra in modalità debug.
DOZEEN [5]	Doze Enable, seleziona il comportamento che deve avere il modulo quando la scheda entra in modalità attesa.

Il modulo prende come ingresso un segnale di clock per incrementare il proprio contatore. È possibile scegliere tra tre diverse sorgenti per tale sorgente: impostando i due bit *CMOD* nel registro *TPMx_SC* a 00, il timer è disabilitato, mentre impostando CMOD a 01 si seleziona il clock della periferica. Impostando CMOD a 10 si seleziona come sorgente un segnale esterno, che viene mandato in un sincronizzatore, in modo che l'incremento avvenga solamente in seguito a un fronte di salita del segnale esterno, e comunque sincronamente al clock della periferica; per un corretto funzionamento, la frequenza massima del segnale esterno deve essere minore della metà del segnale di clock.

Il segnale selezionato viene quindi mandato a un divisore di frequenza (il prescaler). Il segnale in uscita dal prescaler mantiene la stessa forma d'onda di quello in ingresso, ma la frequenza di output è divisa per una potenza di due. Per selezionare il fattore di divisione, bisogna impostare opportunamente i tre bit PS (000 corrisponde a in fattore di divisione 1, 001 corrisponde a 2, 010 a 4, e così via). Questa operazione deve essere eseguita prima di abilitare il segnale di input modificando i bit CMOD.

Il segnale uscente dal divisore di frequenza viene quindi usato come ingresso per il

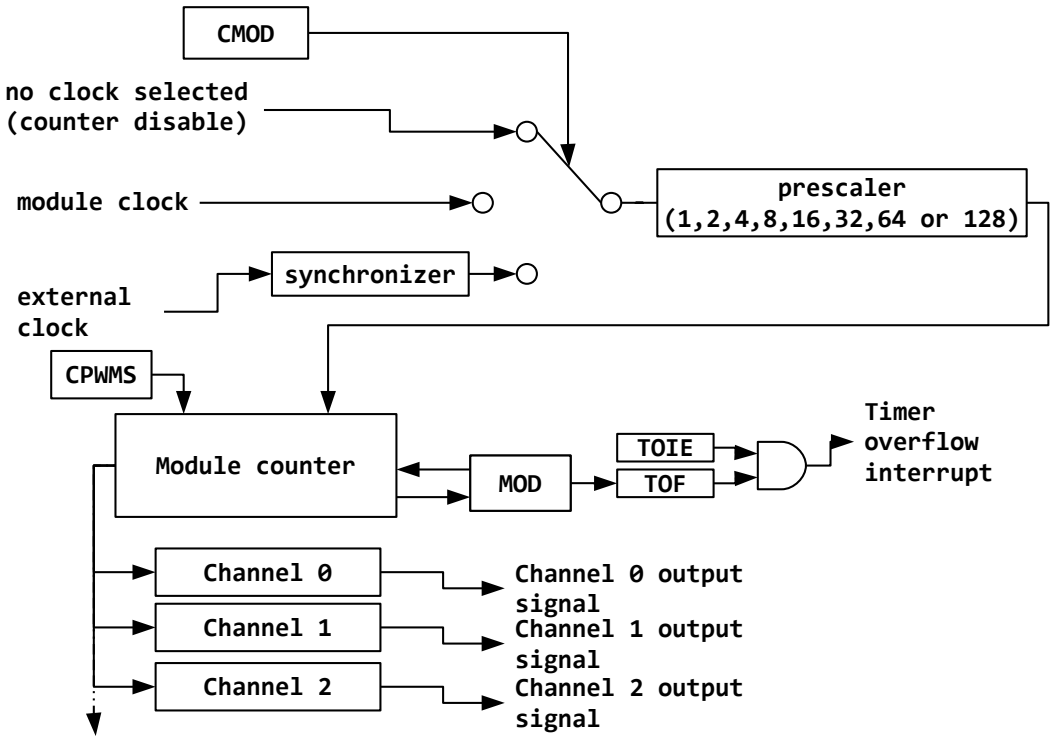


Figura 2.5: Schema del Timer PWM Module (TPM)

modulo contatore, il cuore del TPM. Un parametro fondamentale di questo dispositivo è il *modulo*, un valore numerico compreso tra 1 e 65535 (cioè $2^{16} - 1$), e può essere impostato modificando il contenuto dei 16 bit *MOD* nel registro *TPMx_MOD*.

Il conteggio può avvenire in due modi, a seconda che il bit *CPWMS* del registro TPMx_SC sia 0 o 1. Nel primo caso, il contatore viene incrementato da zero fino al modulo. Raggiunto questo valore viene settato il bit *TOF* (*Timer Overflow Flag*), che indica il raggiungimento del valore limite. Se il bit *TOIE* (*Timer Overflow Interrupt Enabled*) è settato, viene anche generato un interrupt dal modulo TPM. Il contatore viene quindi riportato a zero, e si riparte da capo.

Se invece CPWMS è pari a 1, si abilita il conteggio a salita e discesa. In tal caso, il contatore parte da zero fino a raggiongere MOD, e come prima viene settato il flag di overflow. A questo punto il contatore, invece di essere resettato a 0, viene fatto decrementare contando alla rovescia fino a raggiungere lo zero (non si hanno in questo caso interruzioni). Infine il conteggio riprende in avanti e si ripete continuamente.

Il TPM, per come lo si è visto fin ora, può essere usato come timer, in maniera molto simile al Systick, ma non può invece generare segnali di tipo PWM per il controllo dei motori e del servo. Tale compito è svolto invece dai canali, che si possono trovare a valle del modulo contatore. Ogni modulo TPM ha un certo numero di canali[12] ciascuno dei quali è collegato a un pin fisico della scheda e può operare in

[12]Il TPM0 ne ha sei, mentre il TPM1 e il TPM2 ne hanno due.

diverse modalità.

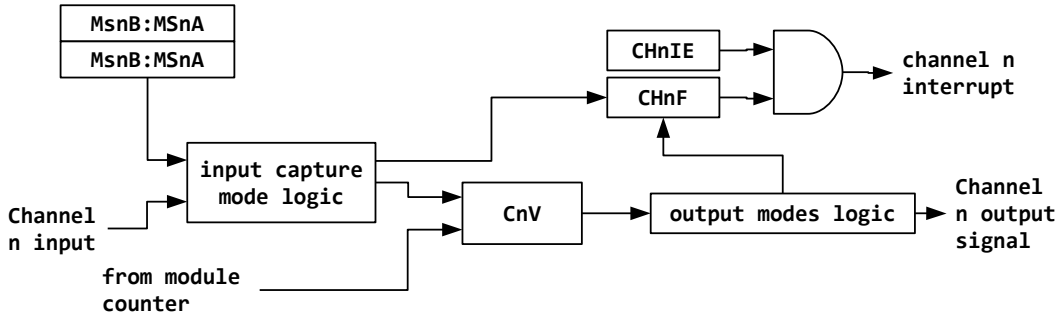


Figura 2.6: Schema di un canale del TPM

La struttura interna di un generico canale è mostrata in figura 2.6. Come si vede, vi sono due diversi segnali in ingresso: uno non è altro che il valore del contatore del TPM, mentre l'altro è un segnale di input specifico per ogni canale.

La modalità di funzionamento viene determinata a seconda dei valori che vengono scritti nei quattro bit *MSnB*, *MSnA*, *ELSnB* e *ELSnB*. Inoltre il comportamento cambia a seconda del contenuto di CPWMS. Quando i quattro bit sono impostati a 0, indipendentemente da CPWMS, il canale è disabilitato.

Di seguito si riportano le modalità di funzionamento dei canali quando CPWMS è pari a 0.

Se [MSnB,MSnA]=00 e almeno uno tra ELSnB e ELSnA è diverso da zero, il canale funziona in modalità "cattura input" (*input capture*). In tale situzione, il pin associato al canale viene usato come input. Quando si verifica un fronte di salita o di discesa del segnale di intput, viene salvato l'attuale valore del contatore TPM all'interno del registro *TPMx_CnV*. A seconda dei valori di ELSnB e ELSnA il canale risulta sensibile ai fronti di salita, a quelli di discesa oppure a entrambi.

Qualora si abbia [MSnB,MSnA]=01 e [ELSnB,ELSnA]≠00 è attiva la modalità *output compare*. In questo caso quando il contatore del TPM raggiunge il valore caricato in TPMx_CnV viene performata un'operazione di *set*, *clear* oppure *toggle* per il pin corrispondente.

Se [MSnB,MSnA]=10 e [ELSnB,ELSnA]≠00 il canale genera in uscita segnali PWM di tipo *edge-aligned*, in cui una delle transizioni avviene all'inizio del periodo del conteggio del TPM, mentre l'altra avviene quando il contatore del TPM raggiunge il valore caricato nel registro CnV. Si può impostare il PWM in modo che all'inizio del periodo l'uscita sia alta e che venga portata a terra raggiunto il valore contenuto in CnV, oppure in modo contrario, iniziando a un livello logico basso e salendo successivamente.

Infine se [MSnB,MSnA]=11 e [ELSnB,ELSnA]≠00 si ha un secondo tipo di modalità output compare. In questo caso infatti al raggiungimento del valore in CnV viene generato un impulso.

Vi è poi un'ulteriore combinazione, in cui però CPWMS è pari a 1, [MSnB,MSnA]=10 e [ELSnB,ELSnA]≠00. In tal caso vengono generati segnali PWM simmetrici rispetto

al centro del periodo del TPM.

In tutte le modalità descritte, quando il contatore eguaglia il valore contenuto in CnV viene settato il bit *CHF* (*Channel Flag*) nel registro *TPMx_CnSC*. Se il bit *CHIE* (*Channel Interrupt Enable*) è settato, viene inoltre generato un interrupt da parte del TPM.

Per far funzionare i motori, si è usato il TPM0, collegando il motore destro al canale 2 e il sinistro al canale 0, entrambi impostati per funzionare in modalità edge-aligned PWM. Il servo viene comandato dal TPM1, canale 0, anch'esso edge-aligned PWM. Gli encoder sono invece collegati ai canali 0 e 1 del TPM2, entrambi funzionanti in modalità input-capture sensibile al fronte di salita. Il codice che permette il setup dei dispositivi conclude questo paragrafo.

Listato 2.10: Setup del TPM0

```
1  void setupMotors() {
2    SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK; // abilita la porta C
3    SIM_SCGC5 |= SIM_SCGC5_PORTE_MASK; // abilita la porta E
4    SIM_SCGC6 |= SIM_SCGC6_TPM0_MASK; // abilita il TPM0
5
6    TPM0_SC = 0; // pulisce il registro per scrivere nuovi valori
7    TPM0_C0SC |= TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK; // imposta MSB=1 e
         ELSB=1 nel registro TPM0_C0SC
8    TPM0_C2SC |= TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK; // imposta MSB=1 e
         ELSB=1 nel registro TPM0_C2SC
9    TPM0_CONF = 0; // pulisce il registro per evitare "preconfigurazioni
         spiacevoli"
10
11   TPM0_SC |= TPM_SC_PS(2); // imposta il valore di prescale a 10
         (divide-by-4): la frequenza dei contatori e' 8kHz
12   TPM0_SC &= ~(1<<TPM_SC_CPWMS_SHIFT); // imposta PWM edge-aligned
13
14   TPM0_MOD = TPM_MOD_MOD(PWM_MOTOR_PERIOD); // imposta il periodo del PWM
15
16   TPM0_C0V = TPM_CnV_VAL(LOW_MOTOR_DUTY); // imposta la velocita' del motore
         al minimo
17   TPM0_C2V = TPM_CnV_VAL(LOW_MOTOR_DUTY); // imposta la velocita' del motore
         al minimo
18
19   // pin per la gestione del ponte H
20   GPIOC_PCOR |= 1<<2; // mette a terra la porta PTC2
21   GPIOC_PDDR |= 1<<2; // imposta la porta PTC2 come output
22   GPIOC_PCOR |= 1<<4; // mette a terra la porta PTC4
23   GPIOC_PDDR |= 1<<4; // imposta la porta PTC4 come output
24
25   GPIOE_PCOR |= 1<<21; // pulisce la porta PTE21
26   GPIOE_PDDR |= 1<<21; // imposta la porta PTE21 come output
27
28   TPM0_SC |= TPM_SC_CMOD(1); // attiva il TPM0
29
30   PORTC_PCR1 = (PORTC_PCR1 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(4); //
         collega PTC1 a FTM0_CH0
31   PORTC_PCR2 = (PORTC_PCR2 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1); //
         collega PTC2 a GPIO
32   PORTC_PCR3 = (PORTC_PCR3 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(4); //
         collega PTC3 a FTM0_CH2
33   PORTC_PCR4 = (PORTC_PCR4 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1); //
         collega PTC4 a GPIO
34   PORTE_PCR21 = (PORTE_PCR21 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1); //
         collega PTE21 a GPIO
35 }
```

Listato 2.11: Setup del TPM1

```
1  void setupServo() {
2    SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK; // abilita il clock per la porta B
3
4    SIM_SCGC6 |= SIM_SCGC6_TPM1_MASK; //abilita il clock per TPM1
5
6    TPM1_SC = 0; // pulisce il registro per scrivere nuovi valori
7    TPM1_C0SC |= TPM_CnSC_MSB_MASK | TPM_CnSC_ELSB_MASK; // imposta MSB=1 e
         ELSB=1 nel registro TPM1_C0SC
8    TPM1_CONF = 0; // pulisce il registro per evitare "preconfigurazioni
         spiacevoli"
9
10   TPM1_SC |= TPM_SC_PS(0); // imposta il valore di prescale a 0: il clock di
         conteggio e' pari a 32kHz
11   TPM1_SC &= ~(1<<TPM_SC_CPWMS_SHIFT); // imposta CPWMS = 0
12
13   TPM1_MOD = TPM_MOD_MOD(PWM_SERVO_PERIOD); // imposta il periodo nel registro
         TPMx_MOD
14
15   TPM1_C0V = TPM_CnV_VAL(NEUTRAL_SERVO_DUTY); // imposta il duty cycle in modo
         da avere il servo in posizione neutra
16
17   TPM1_SC |= TPM_SC_CMOD(1); // abilita il conteggio del TPM1
18
19   // tramite multiplexer imposta per il pin la funzione FTM1_CH0
20   PORTB_PCR0 = (PORTB_PCR0 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(3);
21 }
```

Listato 2.12: Setup del TPM2

```
1  void setupEncoder() {
2    SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK; // abilita il clock per la porta A
3    SIM_SCGC6 |= SIM_SCGC6_TPM2_MASK; //abilita il clock per TPM2
4
5    TPM2_SC = 0; // pulisce il registro per scrivere nuovi valori
6    TPM2_C0SC |= TPM_CnSC_ELSA_MASK; // imposta ELSA=1 nel registro TPM2_C0SC
7    TPM2_C1SC |= TPM_CnSC_ELSA_MASK; // imposta ELSA=1 nel registro TPM2_C1SC
8    TPM2_CONF = 0; // pulisce il registro per evitare "preconfigurazioni
         spiacevoli"
9
10   TPM2_SC |= TPM_SC_PS(0); // imposta il valore di prescale a 0: il clock di
         conteggio e' pari a 32kHz
11   TPM2_MOD = TPM_MOD_MOD(32000); //imposta il periodo nel registro TPMx_MOD
12   TPM2_SC |= TPM_SC_CMOD(1); // abilita il conteggio del TPM2
13   TPM2_C0SC |= TPM_CnSC_CHIE_MASK; // abilita gli interrupt per il canale 0
14   TPM2_C1SC |= TPM_CnSC_CHIE_MASK; // abilita gli interrupt per il canale 1
15
16   PORTA_PCR1 = (PORTA_PCR1 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(3); //
         tramite multiplexer imposta per il pin la funzione FTM2_CH0
17   PORTA_PCR2 = (PORTA_PCR2 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(3); //
         tramite multiplexer imposta per il pin la funzione FTM2_CH1
18   NVIC_ISER |= 1<<19; // abilita la gestione degli interrupt del TPM2
19 }
```

2.6 ADC

Il *Convertitore Analogico-Digitale* (solitamente detto *ADC*, *Analog to Digital Converter*) è un dispositivo fondamentale per il funzionamento di quelle applicazioni elettroniche che devono interagire con l'ambiente in cui operano acquisendo dati da sensori di misura. Come dice il nome, la sua funzione è quella di convertire una tensione analogica in una stringa di bit digitale che rappresenta il livello di tensione presente in ingresso.

Le caratteristiche di un ADC sono molte, e presentarle compiutamente esula dallo scopo di questo paragrafo. Si presenteranno quindi i soli concetti indispensabili per comprenderne il funzionamento al fine di acquisire un segnale analogico tramite la Freedom Board.

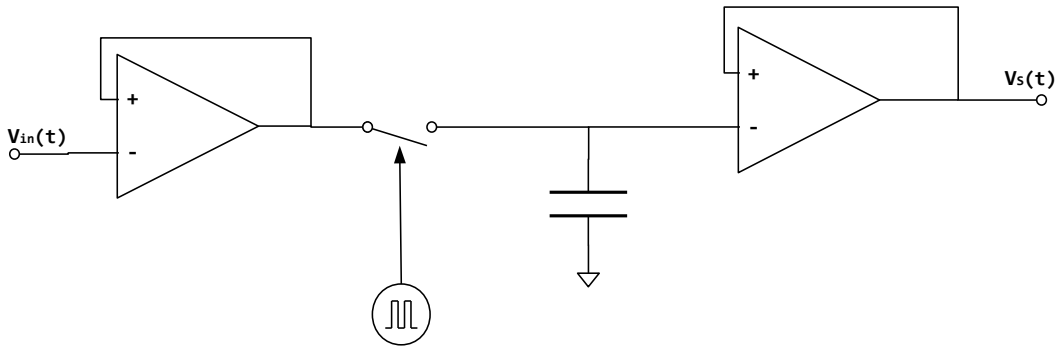


Figura 2.7: Circuito sample & hold

Un convertitore analogico-digitale si compone di due blocchi fondamentali: un circuito *sample & hold* e un circuito quantizzatore. Il primo è visibile in figura 2.7: questa mostra che il circuito si compone di un buffer unitario, che ha lo scopo di separare elettricamente il segnale di ingresso, $V_{in}(t)$, dallo stadio convertitore. Segue un interruttore pilotato: quando il contatto viene chiuso il condensatore si carica, "memorizzando" la tensione di ingresso in quell'istante. Quando il contatto si apre, il condensatore rimane isolato e, trascurando le perdite, mantiene il livello di tensione a un valore costante fino alla successiva chiusura del contatto. L'uscita, indicata in figura come $V_S(t)$, risulta quindi essere, con buona approssimazione, una versione campionata e mantenuta (da qui il nome sample and hold) del segnale in ingresso. Un esempio di segnali di ingresso e di uscita sono mostrati in figura 2.8.

Il segnale campionato e mantenuto viene quindi mandato in ingresso al circuito di quantizzazione. L'architettura del convertitore può essere di diversi tipologie. La Freedom Board utilizza ad esempio un'architettura *SAR* (*Successive Approximation Register*), il cui funzionamento è descritto di seguito. Altre architetture comuni sono ad esempio FLASH, a singola rampa, multirampa, ad integrazione. Per maggiori informazioni su tali architetture si rimanda a [3].

I due parametri fondamentali di un ADC SAR sono il campo di variazione dei valori in ingresso, che verrà indicato con R, e il numero di bit (N). Il primo rappresenta l'intervallo di valori che il dispositivo può accettare in ingresso, ad esempio $[0, 5]\,V$.

Il secondo indica invece quante cifre vengono utilizzate per rappresentare le tensioni digitalizzate. A partire dalla conoscenza di tali informazioni, si può risalire a un ulteriore importante parametro di un convertitore, e cioè il passo di quantizzazione. Questo rappresenta la distanza tra due livelli di transizione successivi, ed è ottenibile come $Q = (R_{max} - R_{min})/2^N$. L'importanza di tale valore risiede nel fatto che ogni livello di tensione quantizzato nelle ampiezze è esprimibile come multiplo intero del passo di quantizzazione e pertanto si può associare alla tensione $V_S \in [nQ, (n+1)Q)$ la rappresentazione binaria del numero n.

Per fare un esempio, se $R = [0, 5]\,V$ e $N = 8$, si ha $Q = 5/2^8 = 0,01953125V$. Se in ingresso fossero presenti $v_{in} = 2,97V$ si avrebbe $V_S \in [152Q, 153Q)$, perciò si potrebbe associare alla tensione in ingresso il numero binario a 8 bit 10011000.

Per attuare la conversione, l'ADC SAR utilizza uno schema di funzionamento come quello indicato in figura 2.9. Come si può vedere, il modulo si compone di un comparatore, del registro ad approssimazioni successive e di un *DAC* (*Digital to Analog Converter*).

Il comparatore fornisce in uscita un valore logico alto se il segnale sul terminale positivo è maggiore del segnale sul terminale negativo, altrimenti si porta a un livello logico basso.

Il DAC converte una stringa digitale a N bit (dove N deve essere anche il numero di bit dell'ADC) in una tensione in accordo con l'espressione:

$$V_{DAC} = \frac{V_0}{2^N}\left(b_0 2^{N-1} + b_1 2^{N-2} + \cdots + b_{N-2}2^1 + b_{N-1}2^0\right)$$

Ove i coefficienti b_i corrispondono alle cifre della stringa digitale. b_{N-1} rappresenta la cifra meno significativa, mentre b_0 corrisponde alla cifra più significativa. Il valore V_0 deve essere scelto in modo da uguagliare $R_{max} - R_{min}$.

Si nota infine che il registro ad approssimazione è una macchina sincrona, e richiede dunque un clock. Tale segnale deve avere frequenza maggiore della frequenza di campionamento, per motivi che si capiranno tra poco.

La conversione avviene per iterazione in più passi, durante i quali il codice di uscita, C_{out}, si avvicina sempre di più alla rappresentazione corretta. Al passo iniziale, nel SAR viene caricata la stringa $[b_0, b_1, b_2, \cdots, b_{N-2}, b_{N-1}] = [1, 0, 0, \cdots, 0, 0]$. Il DAC provvede a convertirla in un valore analogico che risulterà pari a $V_0/2$.

Il comparatore confronta dunque la tensione di ingresso con quella in uscita dal DAC, e si porta allo stato corretto. Il SAR provvede a leggere tale valore e pone b_0 a 1 se l'uscita del comparatore è a livello logico alto, altrimenti imposta tale bit a 0.

Ora viene caricato il valore 1 nel bit b_1, lasciando invariato b_0. Dopo la conversione digitale-analogica si legge il risultato del confronto, e si pone $b_1 = 1$ se l'uscita del comparatore è alta, $b_1 = 0$ altrimenti.

Si continua così fino a che non si è impostato l'ultimo bit, cioè b_{N-1}.

Per comprendere il funzionamento di questo algoritmo, è necessario considerare che, fissato un certo N, ogni numero X compreso tra 0 e 2^N è esprimibile in base 2 mediante la serie:

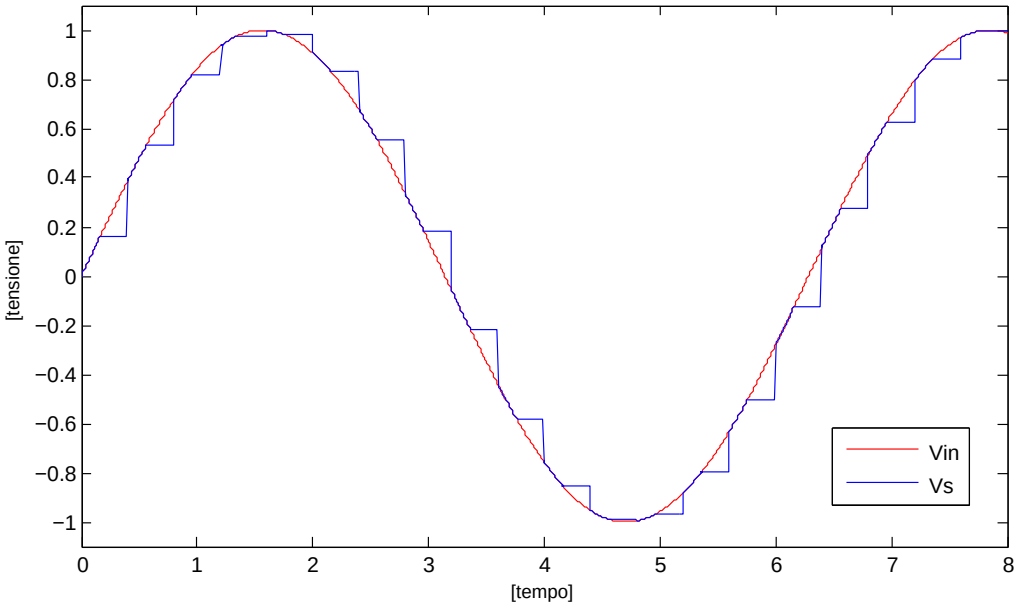


Figura 2.8: Segnali di ingresso e uscita in un circuito sample & hold

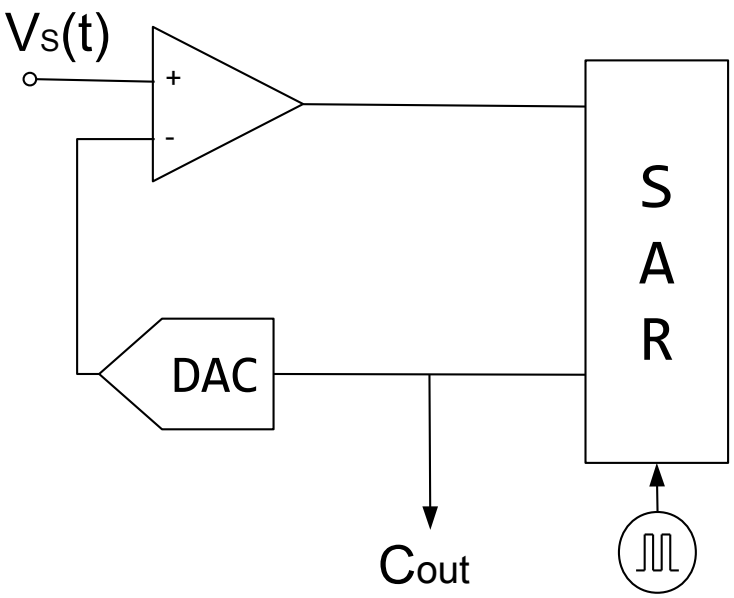


Figura 2.9: Schema del circuito quantizzatore di un ADC SAR

$$X = \sum_{i=0}^{+\infty} x_i 2^{(N-1)-i}$$

Per determinati valori dei coefficienti x_i, che possono essere nulli oppure unitari.

Si può altresì affermare che, dato un numero $M > 0$, qualunque numero X, positivo e minore di M, può esprimersi come:

$$X = \frac{M}{2^N} \sum_{i=0}^{+\infty} x_i 2^{(N-1)-i}$$

In cui i coefficienti x_i non sono in generale uguali a quelli riportati nella prima serie.

Nel caso dell'ADC, si ha che il valore di M corrisponde a V_0, mentre la quantità X corrisponde alla tensione da convertire, V_S. Lo scopo della conversione è di fatto proprio quello di trovare i diversi coefficienti x_i, approssimando il numero incognito con N cifre invece che con una serie infinita. Il funzionamento dell'algoritmo diventa dunque facile da capire. Si supponga che al passo k-esimo della conversione si siano individuati tutti i coefficienti da 0 a k-1. Pertanto la stringa C_{out} sarà uguale a $[x_0, x_1, \cdots, x_{k-2}, x_{k-1}, 0, 0, \cdots, 0, 0]$. Si pone adesso a 1 il coefficiente k-esimo. La tensione in uscita dal convertitore sarà dunque:

$$V_{DAC} = \frac{V_0}{2^N} \sum_{i=0}^{N-1} b_i 2^{(N-1)-i} = \frac{V_0}{2^N} \left[\left(\sum_{i=0}^{k-1} x_i 2^{(N-1)-i} \right) + 2^k \right]$$

Il comparatore valuta quindi la differenza

$$\Delta V = V_S - V_{DAC} = \frac{V_0}{2^N} \left[\left(\sum_{i=0}^{+\infty} x_i 2^{(N-1)-i} \right) - \left(\sum_{i=0}^{k-1} x_i 2^{(N-1)-i} \right) - 2^k \right]$$

Spezzando la prima sommatoria, e cancellando i termini uguali si ottiene:

$$\Delta V = \frac{V_0}{2^N} \left[(x_k - 1)\, 2^k + \sum_{i=k+1}^{+\infty} x_i 2^{(N-1)-i} \right]$$

Il valore di x_k può essere 1 o 0, e perciò si ha:

$$\Delta V = \begin{cases} \frac{V_0}{2^N} \left[\sum_{i=k+1}^{+\infty} x_i 2^{(N-1)-i} \right] & (x_k = 1) \\ \frac{V_0}{2^N} \left[-2^k + \sum_{i=k+1}^{+\infty} x_i 2^{(N-1)-i} \right] & (x_k = 0) \end{cases}$$

Inoltre, considerando che $2^k = \sum_{i=1}^{+\infty} 2^{k-i}$ si ha:

$$\begin{cases} \Delta V > 0 \Leftrightarrow x_k = 1 \\ \Delta V < 0 \Leftrightarrow x_k = 0 \end{cases}$$

E pertanto è possibile impostare $b_k = 1$ se $\Delta V > 0$, in caso contrario sarà $b_k = 0$. Ripetendo il procedimento per tutti i bit rimanenti, si riesce a ottenere la stringa C_{out}.

La conversione di un valore di ingresso richiede in totale N cicli di clock: questi devono avvenire tutti nel lasso di tempo in cui l'interruttore del circuito sample & hold rimane aperto, poichè altrimenti la tensione ai capi del condensatore cambierebbero, e l'algoritmo di conversione non risulterebbe più valido. Questo giustifica un'affermazione fatta in precedenza: il clock fornito al SAR deve avere frequenza maggiore di quella del segnale di controllo dell'interruttore.

Il modulo ADC della Freedom Board monta un ADC come quello descritto, e permette di impostare diversi parametri di funzionamento, come il campo di valori in ingresso e il numero di bit. Si è deciso di utilizzare come campo di ingresso il range di tensioni [0,3.3]V, e un numero di bit per la conversione pari a 8 (a cui corrispondono 255 livelli di quantizzazione). In questo modo è possibile salvare il valore convertito in una variabile di tipo `unsigned char` (intero a 8 bit).

L'ADC si interfaccia con le periferiche esterne mediante diversi canali. Per avviare una conversione è necessario scrivere nel registro *ADC0_SC1A*, nei cinque bit *ADCH*, il canale da selezionare per la conversione. I canali si dividono in due sottotipi, e cioè canali 'a' e canali 'b'. Prima di avviare la conversione, è opportuno selezionare quali canali si vogliono utilizzare, scrivendo nel registro *ADC0_CFG2* nel bit *MUXSEL*.

I listati 2.13 e 2.14 contengono il codice che si è utilizzato per far funzionare l'ADC.

Listato 2.13: Macro e variabili per facilitare l'uso del modulo ADC

```
1  #define ADC_USER_NONE 0
2  #define ADC_USER_CAMDX 1
3  #define ADC_USER_CAMSX 2
4  #define ADC_USER_POT1 3
5  #define ADC_USER_POT2 4
6
7  // macro che seleziona i canali "a" per le conversioni dell'ADC
8  #define SET_ADCa  ADC0_CFG2 &= ~(ADC_CFG2_MUXSEL_MASK)
9  // macro che seleziona i canali "b" per le conversioni dell'ADC
10 #define SET_ADCb  ADC0_CFG2 |= ADC_CFG2_MUXSEL_MASK
11 // macro che avvia la conversione tramite il canale selezionato (ch)
12 #define ADC_CONV(ch) ADC0_SC1A = (ADC_SC1_ADCH(ch) | ADC_SC1_AIEN_MASK);
13
14 // variabili che contengono il risultato della conversione per ogni periferica
15 volatile unsigned char adc_result;
16 volatile unsigned char adc_result_camDX;
17 volatile unsigned char adc_result_camSX;
18 volatile unsigned char adc_result_pot1;
19 volatile unsigned char adc_result_pot2;
20 volatile unsigned char adc_user;
```

Come si può notare, si sono definite delle variabili di supporto per la gestione delle conversioni. I sensori utilizzati sono quattro (due potenziometri e due telecamere)

Listato 2.14: Funzioni per l'uso del modulo ADC

```
1  void setupADC() {
2    // inizializza le variabili adibite per le conversioni dell'ADC
3    adc_user = ADC_USER_NONE;
4    adc_result = 0;
5    adc_result_camSX = 0;
6    adc_result_camDX = 0;
7    adc_result_pot1 = 0;
8    adc_result_pot2 = 0;
9
10   // SETUP DELL' ADC0
11   SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK; // abilita il clock per l'ADC0
12   NVIC_ISER |= NVIC_ISER_SETENA(1<<15); // abilita la gestione degli interrupt
         di ADC0
13 }
14
15 void ADC0_IRQHandler() {
16   switch(adc_user) {
17   case ADC_USER_NONE:
18     adc_result = ADC0_RA;
19   break;
20   case ADC_USER_CAMDX:
21     adc_result_camDX = ADC0_RA;
22     adc_user = ADC_USER_NONE;
23   break;
24   case ADC_USER_CAMSX:
25     adc_result_camSX = ADC0_RA;
26     adc_user = ADC_USER_NONE;
27   break;
28   case ADC_USER_POT1:
29     adc_result_pot1 = ADC0_RA;
30     adc_user = ADC_USER_NONE;
31   break;
32   case ADC_USER_POT2:
33     adc_result_pot2 = ADC0_RA;
34     adc_user = ADC_USER_NONE;
35     break;
36   default:
37     adc_result = ADC0_RA;
38     adc_user = ADC_USER_NONE;
39   break;
40   }
41 }
```

e si sono definite altrettante variabili di tipo `unsigned char`. Questo perchè quando più periferiche fanno richiesta dell'ADC è facile che appena una conversione è terminata ne inizi una immediatamente dopo: se si utilizzasse una sola variabile per tutte le periferiche, si rischierebbe di perdere alcune conversioni.

Il problema si pone in particolare quando entrano in gioco le interruzioni. Ad esempio il programma principale potrebbe richiedere la conversione del valore di un potenziometro, mettendosi in attesa del completamento della conversione. In questo momento di attesa, potrebbe verificarsi un'interruzione, e l'esecuzione passerebbe al gestore dell'interrupt. Se all'interno di tale funzione vi fosse un segmento di codice del tipo *avvia una conversione* → *attendi fino alla fine della conversione* → *leggi il dato e fai qualcosa* il risultato della conversione avviata nel flusso principale andrebbe perso.

Il meccanismo ideato per prevenire queste problematiche si basa sul presupposto che difficilmente diversi segmenti di codice vadano a usare medesime periferiche, e che quindi per evitare che una conversione ne sovrascriva un'altra basta creare variabili dedicate in cui salvare i risultati di ogni singola periferica. Pertanto, la telecamera sinistra può leggere i dati in maniera sicura dalla variabile `adc_result_camSX`, mentre il potenziometro 1 li otterà acedendo al contenuto di `adc_result_pot1`.

L'attuale utilizzatore viene salvato nella variabile `adc_user`, che ha una doppia funzione. Da un lato, indica al gestore degli interrupt del modulo ADC dove andare a salvare il risultato della conversione appena effettuata. Dall'altro, finché la conversione non è terminata, mantiene traccia di chi sta usando in quel momento l'ADC, notificando agli altri dispositivi che non possono richiedere conversioni. Una volta completata la digitalizzazione, `adc_user` viene pulita automaticamente inserendovi il valore `ADC_USER_NONE`.

In conclusione, quando un sensore intende fare richiesta di uso dell'ADC, la prima cosa da fare è aspettare fino a quando `adc_user` non contiene il valore `ADC_USER_NONE`. Immediatamente dopo, bisogna occupare l'ADC impostando opportunamente `adc_user`: in questo modo si è sicuri che nessuno utilizzerà il convertitore. Si procede quindi selezionando i canali *a* o *b* a seconda della propria necessità, e infine dando il via alla conversione. I passaggi appena descritti sono mostrati nel listato 2.15.

Listato 2.15: Esempio di richiesta di conversione A-D

```
...
// attende che l'ADC si renda disponibile
while(adc_user != ADC_USER_NONE);

// occupa l'ADC per una conversione
adc_user = ADC_USER_CAMSX;

SET_ADCb; // seleziona i canali b
ADC_CONV(6); // avvia la conversione sul canale 6b

// attende che la conversione sia completa
while(adc_user != ADC_USER_NONE);

unsigned char cam_value = adc_result_camSX;
...
```

Capitolo 3

Line Scan Camera

Tra i diversi sensori utilizzati nella competizione, quello di maggiore importanza è la fotocamera. Questa permette di acquisire informazioni sulla conformazione del tracciato: le informazioni, appropriatamente elaborate, costituiscono l'ingresso principale dell'intero sistema.

In questo capitolo si vuole presentare il funzionamento del dispositivo in maniera completa, sottolineando di volta in volta gli aspetti problematici che possono portare a un funzionamento scorretto del sistema di acquisizione. Si illustreranno dunque le soluzioni hardware e software adottate per risolvere i problemi.

3.1 Principio di Funzionamento della Fotocamera

Il sensore utilizzato è siglato *TSL1401CL*, e viene prodotto dalla *TAOS* (azienda attualmete di proprietà della *AMS*). Esso consiste in un array di 128 fotodiodi che, se investiti da una radiazione luminosa, generano una corrente. Questa corrente viene integrata da un'apposita circuiteria e la tensione risultante viene utilizzata come indice della luminosità incidente sul fotodiodo.

L'alimentazione può essere a 3 o a 5V, adattandosi alle possibilità della maggior parte dei microcontrollori, inclusa la Freedom Board, che fornisce al sensore una tensone di 3.3V.

Lo schema funzionale del sensore è riportato in figura 3.1. Come si vede dallo stesso, i segnali utilizzati dal sensore sono tre: *CLK* (segnale di clock), *SI* (*Serial Input*) e *AO* (*Analog Output*). Il primo permette di sincronizzare la trasmissione dei dati dei singoli pixel, mentre il secondo indica al sensore di iniziare a trasmettere i dati. Il terzo segnale corrisponde all'uscita analogica del sensore.

In figura 3.2 si possono vedere i segnali digitali nel tempo (l'uscita, si ricorda, è invece analogica; nel relativo diagrama si distinguano la porzione che indica che l'uscita è abilitata e quella in cui il pin viene posto in alta impedenza, *Hi-Z*).

In ogni pixel è presente una circuiteria dedicata per la generazione della tensione di uscita: un fotodiodo genera una corrente, funzione dell'energia luminosa entrante, che viene utilizzata come ingresso per un circuito integratore realizzato con un

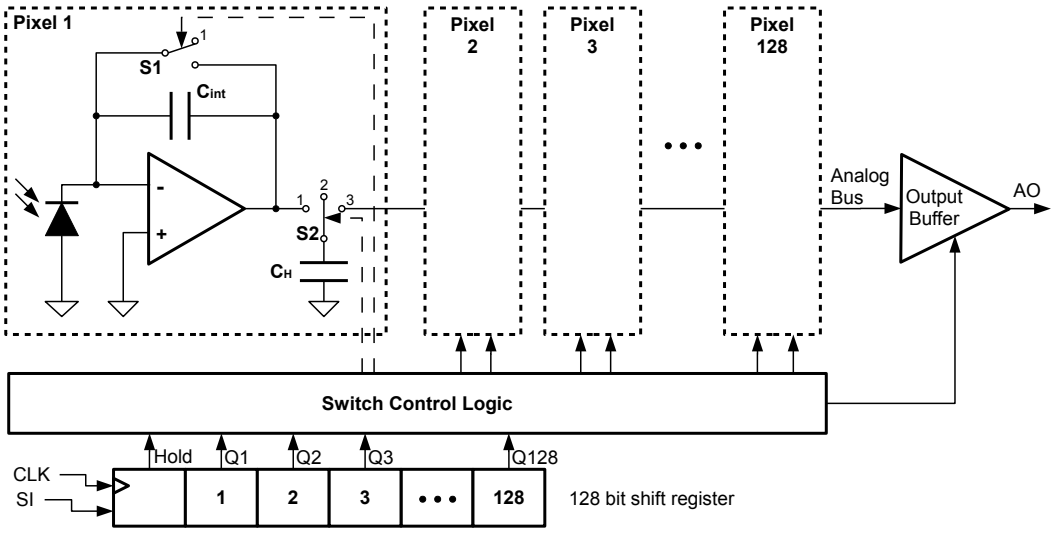


Figura 3.1: Schema funzionale della fotocamera

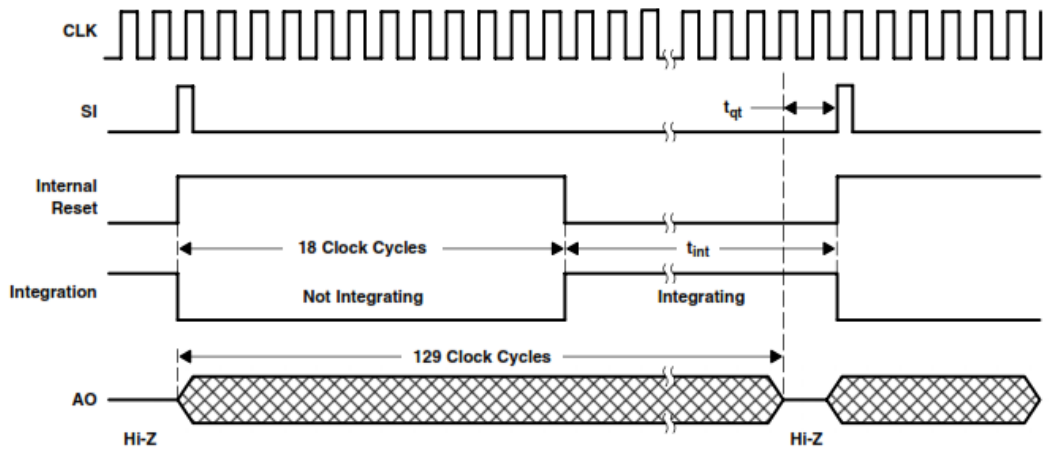


Figura 3.2: Segnali interni alla telecamera

operazionale e un condensatore. La tensione di uscita può essere espressa mediante la relazione lineare:

$$V_{out} = V_{drk} + R_e \, E_e \, t_{int}$$

Si nota che l'uscita presenta un certo offset V_{drk}: questo rappresenta la tensione che si misura sul morsetto analogico quando il sensore non viene irradiato da una fonte luminosa.

Non considerando l'offset, la caratteristica mette in evidenza che vi sono tre parametri che concorrono a determinare il valore di uscita. R_e rappresenta la *responsività* del singolo pixel ad una certa lunghezza d'onda (essa è espressa in $V\,cm^2/\mu J$; si rimanda al manuale tecnico [2] per ulteriori informazioni). È importante notare che la responsività del sensore è funzione dell'ambiente in cui la telecamera opera, e pertanto non è facilmente controllabile. Nonostante ciò, è lecito supporre che questo parametro rimanga, entro certi limiti, invariato durante un giro in pista, in quanto la fonte luminosa (sia essa il Sole o una lampada) difficilmente cambierà durante la singola corsa.

E_e corrisponde alla densità di radiazione incidente (espressa in $\mu W/cm^2$), e corrisponde all'informazione utile che si vuole ottenere mediante il sensore. Infatti, assumendo costanti R_e e t_{int} (ipotesi ragionevole in una singola acquisizione), un pixel che punti a una superficie scura e opaca sarà esposto a una bassa densità di radiazione (in quanto le superfici scure e opache hanno un basso indice di riflessione dell'energia luminosa) mentre un pixel rivolto verso una superficie chiara e opaca verrà investito da molta energia luminosa (le superfici chiare hanno basso indice di assorbimento). Chiaramente il discorso è più complesso, in quanto l'energia riflessa dipende, oltre che dal colore, da altre caratteristiche del materiale. Tuttavia, per quanto concerne il tracciato della Freescale Cup, questo principio è sufficiente al fine di distinguere il fondo bianco della pista dalle linee nere.

L'ultimo parametro, t_{int}, prende il nome di *tempo di integrazione*, e corrisponde al tempo in cui il condensatore di tenuta (quello a valle del circuito integratore) viene caricato dal circuito integratore.

L'acquisizione dei dati inizia quando viene generato un impulso sul terminale SI. Pioché il clock è di tipo *PET* (*Positive Edge Triggered*), il segnale di controllo deve essere portato a livello logico alto prima di un fronte di salita. In particolare, si legge in [2] che tale evento deve verificarsi almeno $20\,ns$ prima del fronte. Non vi sono invece restrizioni sull'istante in cui l'impulso deve terminare, a patto che ciò avvenga prima del successivo ciclo. Poiché, come riportato nel manuale, il periodo minimo di clock è pari a $100\,ns$, se si sincronizza SI in modo da generare l'impulso a metà del semiperiodo basso e tenendolo in tale stato fino alla metà del semiperiodo alto, non ci saranno problemi indipendentemente dalla frequenza di clock scelta.

Non appena ha luogo l'impulso, la logica di controllo provvede a commutare tutti i deviatori S2 (cfr. fig. 3.1) portandoli dalla posizione 1 (che permette la carica del condensatore di tenuta) alla posizione 2 (un capo del condensatore rimane scollegato e pertanto mantiene, trascurando le perdite, la carica accumulata). Viene dunque attivata la procedura di reset, che consiste nella chiusura di tuttigli interruttori S1:

questi cortocircuitano i capi dei condensatori presenti nei circuiti di integrazione, scaricandoli completamente. Tale procedura dura 18 cicli di clock, dopo i quali gli interruttori S1 vengono aperti avviando l'integrazione.

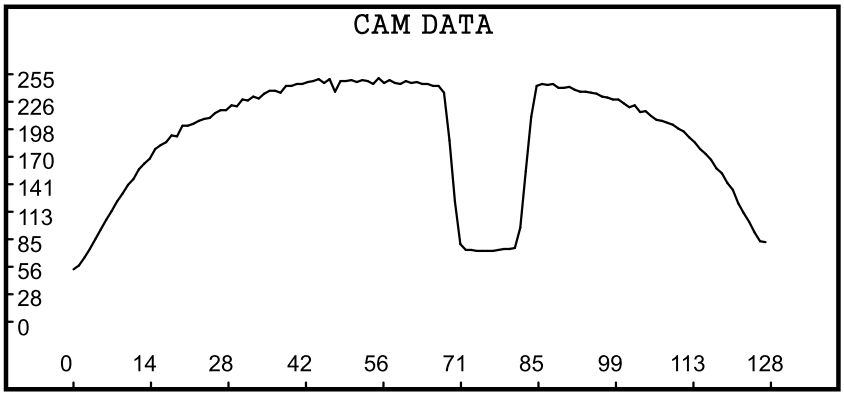
Nel frattempo, il segnale SI viene fatto passare in un registro a scorrimento a 128 bit (uno per ogni pixel del sensore). In ogni istante, il registro conterrà degli 0 logici, ad eccezione dell'impulso che viene propagato. Questo permette di comandare in successione i deviatori S2, che si portano in posizione 3 (collegandosi al bus analogico) quando il corrispondente bit nello shift-register è alto, e tornando alla posizione 2 quando il bit viene fatto scorrere. Pertanto sul terminale di uscita saranno via via presenti le tensioni generate da ciascun pixel, che verranno acquisite dal microcontrollore tramite l'ADC.

Dopo 128 fronti di salita del clock, la procedura di acquisizione è terminata e, con il successivo ciclo, il pin AO viene posto in stato di alta impedenza. I deviatori S2 vengono portati in posizione 1, collegando la capacità di tenuta all'uscita dell'integratore. Per il corretto funzionamento del dispositivo, è necessario attendere ancora un tempo minimo pari a 20 μs per la carica dei condensatori (indicato con t_{qt}), dopo dopo di che è possibile avviare una nuova conversione.

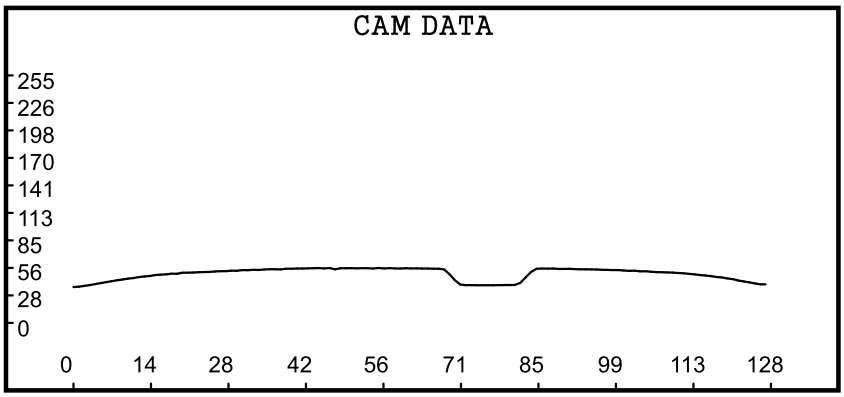
Il tempo che intercorre tra il 19° ciclo di clock e un nuovo impulso del segnale SI corrisponde al tempo di integrazione t_{int}, che, come illustrato, è uno dei parametri fondamentali che concorre alla generazione della tensione in uscita dal dispositivo.

La regolazione di tale valore è uno dei passi fondamentali per poter acquisire in maniera corretta l'immagine del tracciato. Impostando un tempo di integrazione troppo basso, i pixel non hanno tempo per caricare a sufficienza le capacità di tenuta, e pertanto tutti i livelli di tensione saranno molto bassi, ottenendo così un'immagine "scura" (figura 3.3(b)), ovvero in cui tutti i valori di tensione sono bassi. Al contrario, se il tempo è troppo alto, i circuiti integratori si portano in saturazione. I valori di uscita saranno quindi tutti prossimi alla tensione di alimentazione, perdendo dunque informazioni utili sul tracciato che verrà visto come se fosse interamente bianco (figura 3.3(c)).

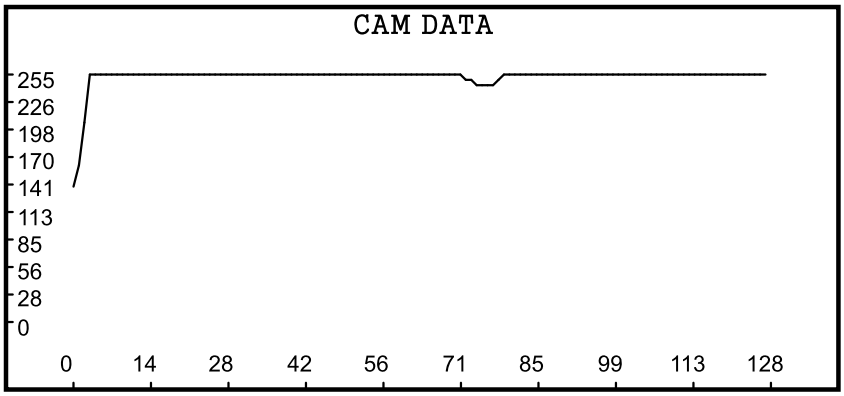
Un'osservazione importante va fatta in riferimento ai bordi dell'immagine 3.3(a). La figura rappresenta un'acquisizione fatta su una superficie bianca con una linea nera al centro: pertanto i livelli di grigio dei pixel dovrebbero essere tutti vicini al valore di saturazione, eccezion fatta per la zona centrale. Come si può invece osservare, i valori ai lati sono inizialmente bassi, e crescono fino a raggiungere la saturazione. Ciò è dovuto a quello che si può definire un "effetto bordo": alle estremità dell'array, i pixel ricevono una scarsa illuminazione a causa della presenza dei bordi della telecamera, che assorbono parte della radiazione luminosa in arrivo. Questo causa una perdita significativa di informazione, e pertanto i dati letti in corrispondenza dei bordi andrebbero ignorati in quanto non rilevanti ai fini dell'identificazione del tracciato.



(a) Immagine originale



(b) Immagine scura



(c) Immagine chiara

Figura 3.3: Acquisizioni con diversi tempi di integrazione: la prima immagine mostra il risultato di un'acquisizione corretta fatta su un tracciato bianco con una linea nera al centro; nelle altre due immagini si vedono acquisizioni fatte con tempo di integrazione o troppo piccolo o troppo grande.

3.2 Periodic Interrupt Timer

Per l'utilizzo delle fotocamere si è scelto di utilizzare il modulo *PIT* (*Periodic Interrupt Timer*) interno alla Freedom Board. Questo modulo permette di generare periodicamente degli interrupt, e permette ai sensori di visione di funzionare continuamente senza che l'esecuzione del programma principale possa influenzarne il comportamento.

Il PIT consiste principalmente di due contatori a 32 bit, chiamati *Timer 0* e *Timer 1*. Tali moduli utilizzano come sorgente sincrona il bus clock, pertanto il loro valore viene decrementato a quella frequenza. Una volta che i contatori raggiungono lo zero, viene generato un interrupt.

La tabella 3.1 mostra i registri del modulo, indicando la funzione dei biti in esso contenuti.

Tabella 3.1: Registri PIT

Registro/bit	Descrizione
PIT_MCR	**PIT Module Control Register, contiene i bit di controllo per l'intero modulo.**
MDIS [30]	Module Disable, permette di disabilitare il clock per i timer del PIT scrivendo un 1, o di abilitarlo scrivendo uno 0.
FRZ [31]	Freeze, imposta il comportamento dei contatori quando si entra in modalità debug.
PIT_LTMR64H	**PIT Upper Lifetime Timer Register, utilizzato congiuntamente al registro PIT_LTMR64L quando si congiungono i due contatori per formarne uno unico a 64 bit.**
LTH [0-31]	Life Timer Value, contiene il valore del Timer 1, e cioè i 32 bit più significativi del contatore a 64 bit.
PIT_LTMR64L	**PIT Lower Lifetime Timer Register, utilizzato congiuntamente al registro PIT_LTMR64H quando si congiungono i due contatori per formarne uno unico a 64 bit.**
LTL [0-31]	Life Timer Value, contiene il valore del Timer 0, e cioè i 32 bit meno significativi del contatore a 64 bit.
PIT_LDVALn	**Timer Load Value Register, permette di impostare per ogni contatore il periodo del conteggio.**
TSV [0-31]	Timer Start Value, il valore che deve essere caricato nel corrispondente timer quando questo raggiunge lo zero.
PIT_CVALn	**Current Timer Value Register, il registro serve per indicare il valore attuale del timer.**

continua nella pagina successiva

continua dalla pagina precedente

Registro/bit	Descrizione
TVL [0-31]	Current Timer Value, rappresenta l'attuale valore del timer, se il conteggio è abilitato.
PIT_TCTRLn	**Timer Control Register, contiene i bit di controllo per ciascun contatore.**
CHN [29]	Chain Mode, permette di congiungere questo timer al precedente, in modo da abilitare un unico contatore a più bit.
TIE [30]	Timer Interrupt Enable, abilita la generazione degli interrupt quando il contatore raggiunge il valore 0.
TEN [31]	Timer Enable, permette di abilitare o di disabilitare il contatore.
PIT_TFLGn	**Timer Flag Register, registro in cui è memorizzato il flag per gli interrupt.**
TIF [31]	Timer Interrupt Flag, indica se il contatore ha raggiunto lo 0, e in tal caso genera un interrupt (se TIE è settato). Per pulire tale bit è necessario scrivere un 1.

Per il funzionamento della telecamera si è deciso di utilizzare il Timer 0; il codice che permette di impostare correttamente il PIT è riportato nel listato 3.1.

Per la gestione dei segnali di clock e SI delle telecamere si utilizzano ripettivamente le porte PTE1 e PTD7 condivise per i due sensori (entrambe le porte sono collegate alla funzione GPIO, in modalità output). Per le uscite analogiche, si utilizzano invece i pin PTD5 e PTD6, collegandoli alla funzione ADC0. Il codice relativo è mostrato nel listato 3.2.

Listato 3.1: Codice per il setup del Timer 0

```
1  void PITsetup() {
2    // SETUP DEL PIT0 (Periodic Timer Interrupt)
3    // abilita il clock per il PIT
4    SIM_SCGC6 |= SIM_SCGC6_PIT_MASK;
5    // evita di disabilitare il clock quando entra in modalit debug
6    PIT_MCR &= ~(PIT_MCR_MDIS_MASK);
7    // imposta il contatore del PIT0 in modo da avere come frequenza
         PIT_INTERRUPT_FREQUENCY
8    PIT_LDVAL0 = (BUS_CLOCK_FREQUENCY/PIT_INTERRUPT_FREQUENCY);
9    // abilita la generazione degli interrupt del PIT0
10   PIT_TCTRL0 |= PIT_TCTRL_TIE_MASK | PIT_TCTRL_TEN_MASK;
11   // abilita la gestione da parte della CPU degli interrupt del PIT
12   NVIC_ISER |= NVIC_ISER_SETENA(1<<22);
13 }
```

Listato 3.2: Codice per il setup dei segnali della telecamera

```
void setupSignals() {
  // SETUP DEL SEGNALE DI CLOCK
  // abilita il clock per PTE
  SIM_SCGC5 |= SIM_SCGC5_PORTE_MASK;
  // imposta la funzione GPIO per PTE1
  PORTE_PCR1 |= PORT_PCR_MUX(1);
  // imposta il pin in scrittura
  GPIOE_PDDR |= 1<<1;
  // resetta il valore del pin
  GPIOE_PCOR |= 1<<1;

  // SETUP DEL SEGNALE SI (INIZIO ACQUISIZIONE)
  // abilita il clock per PTD
  SIM_SCGC5 |= SIM_SCGC5_PORTD_MASK;
  // imposta la funzione GPIO per PTD7
  PORTD_PCR7 |= PORT_PCR_MUX(1);
  // imposta il pin in scrittura
  GPIOD_PDDR |= 1<<7;
  // resetta il valore del pin
  GPIOD_PCOR |= 1<<7;

  // SETUP DELLE PORTE PER LA LETTURA DEL SEGNALE
  setupADC();
  // abilita il clock per PTD
  SIM_SCGC5 |= SIM_SCGC5_PORTD_MASK;
  // imposta ALT0 (ADC) come funzione di PORTD5
  PORTD_PCR5 = (PORTD_PCR5 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(0);
  // imposta ALT0 (ADC) come funzione di PORTD6
  PORTD_PCR6 = (PORTD_PCR6 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(0);
}
```

Ogni chiamata del gestore corrisponde a un quarto di periodo del segnale di clock, la cui frequenza corrisponde quindi a un quarto di quella del PIT. Come si vede nel frammento di codice 3.1, il valore di reset del contatore viene posto a *BUS_CLOCK_FREQUENCY/PIT_INTERRUPT_FREQUENCY*: questo permette di avere come frequenza per il PIT quella desiderata. Nel caso specifico, si ha che il bus clock ha frequenza pari a 24 MHz, mentre si è scelto di impostare la frequenza del PIT a 50 kHz (a cui corrisponde una frequenza di lavoro della telecamera pari a 12.5 kHz). Ne consegue che il valore da caricare in PIT_LDVAL0 è pari a 480.

3.3 Algoritmo di Gestione delle Fotocamere

Si è deciso di utilizzare il PIT in modo che gestisca le telecamere procedendo attraverso diversi stati identificati da un numero intero e postivo (da 0 a 9). A ogni chiamata del gestore degli interrupt generati da tale modulo, viene eseguita un'operazione specifica a seconda dello stato in cui ci si trova, quindi quest'ultimo viene aggiornato a seconda dell'operazione che dovrà essere eseguita successivamente.

Come ausilio alla comprensione del funzionamento dell'algoritmo di gestone delle fotocamere, si faccia riferimento al diagramma riportato in figura 3.4. Una variabile di tipo `volatile unsigned char`, chiamata `stato`, memorizza l'attuale stato all'interno dell'algoritmo, inizialmente posto a zero. Un'altra variabile `volatile unsigned char selezione_pixel` permette di tenere traccia del pixel che deve essere convertito.

Infine, i pixel sono memorizzati in due buffer separati: `volatile unsigned char bufferDX[128]` e `volatile unsigned char bufferSX[128]`. Questi verranno poi elaborati da un appropriato algoritmo con lo scopo di estrarre le informazioni utili all'identificazione del tracciato.

Per tenere traccia del tempo che passa, vengono utilizzate due ulteriori variabili: `volatile unsigned int cicli_effettuati` indica quanti cicli di clock sono trascorsi dalla ricezione dell'ultimo segnali di SI, mentre `volatile unsigned int tempo_integrazione` memorizza il tempo di integrazione scelto per la conversione. In verità la variabile non memorizza realmente il tempo di integrazione adottato, ma piuttosto il numero di cicli di clock che si fanno intercorrere tra un segnali di SI e l'altro. Di fatto, posto che sia `tempo_integrazione > 128`, semplicemente si utilizza una differente definizione di tempo di integrazione, che non va a inficiare la riuscita dell'acquisizione dei dati.

Alla prima interruzione del PIT, viene eseguito il blocco corrispondente allo stato 0. Questo provvede a inizializzare il segnale di clock ponendolo a 1, mentre il segnale SI viene posto a zero. Lo stato viene cambiato in 1, e l'esecuzione del gestore termina. Alla seconda chiamata, viene eseguito il blocco corrispondente allo stato 1, che lascia inalterati i due segnali e incrementa lo stato di 1. Il primo semiperiodo di clock è dunque terminato. Alla terza esecuzione, essendo lo stato pari a 2, il clock viene portato a 0, e lo stato viene incrementato.

Viene dunque eseguito il 4° blocco (stato=3), in cui il clock rimane inalterato, mentre SI viene portato a 1. Allo stato successivo, viene modificato il clock: tale

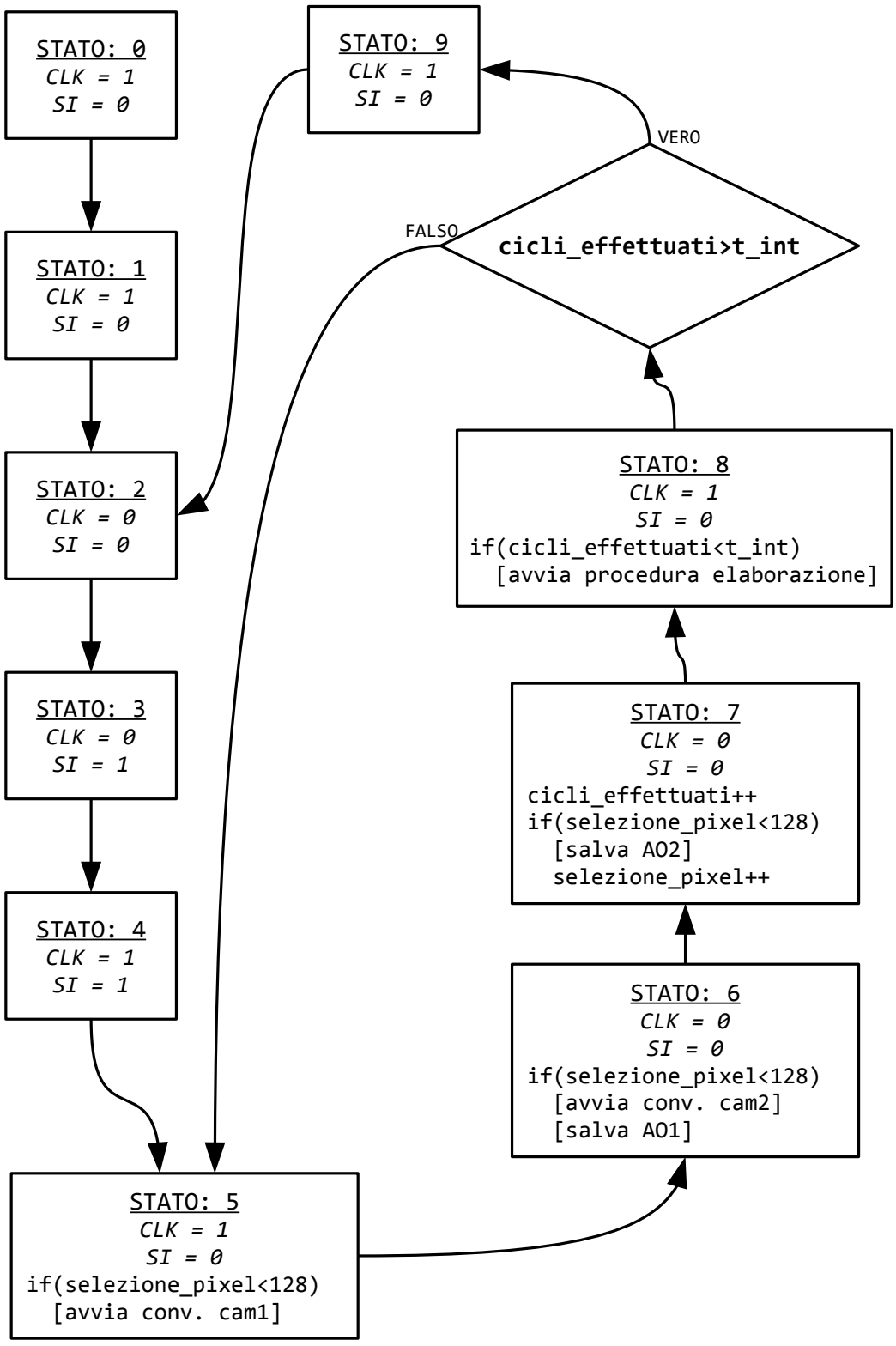


Figura 3.4: Diagramma dell'algoritmo di gestione delle fotocamere

fronte di salita avviene quando SI si trova in uno stato logico alto, e pertanto inizia l'acquisizione dei dati.

Nello stato 5 viene portato a 0 il segnale di inizio acquisizione. Per come è stato strutturato l'algoritmo, ciò avviene in corrispondenza della metà del semiperiodo in cui il clock si trova in uno stato logico alto: come era stato già accennato nel paragrafo 3.1, questo permette di rispettare in maniera semplice le specifiche di sincronizzazione tra SI e clock contenute nel manuale tecnico. In questo stato si provvede inoltre ad avviare la conversione dell'uscita della prima fotocamera tramite l'ADC[1], a patto che il pixel attualmente selezionato sia "valido" (in altre parole, se `selezione_pixel < 128`). Ovviamente tale condizione al momento è sicuramente verificata, considerato che si sta acquisendo il primo pixel. Tuttavia come si nota osservando lo schema degli stati, questo blocco fa parte di un loop che si ripeterà varie volte, e in generale più di 128 cicli saranno performati.

Nello stato 6 il clock viene portato a 0, e si avvia la conversione del successivo pixel. La bassa frequenza di lavoro delle fotocamere (50 kHz per il PIT) dovrebbe assicurare che l'ADC, operante a una frequenza molto maggiore (in quanto il clock di base è quello del bus), sia in grado di completare il proprio compito in tempi sufficientemente brevi. Nonostante ciò, il meccanismo software ideato per la gestione delle conversioni assicura che non verrà avviata alcuna conversione fintanto che quella precedente non è completa. Dopo aver avviato la conversione del pixel della seconda fotocamera, il codice memorizza in `bufferSX` il pixel precedentemente convertito dall'ADC.

Passati dunque allo stato 7, si incrementa il numero di cicli effettuati (istruzione `cicli_effettuati++`). Si procede poi con la memorizzazione in `bufferDX` del pixel convertito e con l'incremento unitario della variabile `selezione_pixel`.

Il codice contenuto nel blocco corrispondente allo stato 8 provvede a verificare se si è raggiunto il tempo di integrazione limite e, nel caso in cui la condizione sia verificata, provvede ad avviare la procedura di elaborazione dei dati. L'ultimo compito che viene portato a termine è la valutazione dello stato successivo: se la disuguaglianza `cicli_effettuati > tempo_integrazione` non è verificata, ciò corrisponde a dire che la procedura di acquisizione non è terminata. In tal caso lo stato successivo sarà il 5°, permettendo di convertire i successivi pixel o, nel caso in cui tutti i pixel siano già stati acquisiti, di far trascorrere il tempo fino a che non si raggiunge il tempo di integrazione desiderato. Nel caso in cui il numero di cicli effettuati sia sufficientemente alto, il ciclo di acquisizione può dirsi terminato, e le variabili `cicli_effttuati` e `selezione_pixel` vengono azzerate. Lo stato successivo sarà il 9°.

L'ultimo stato ha l'unica funzione di completare il semiperiodo di clock iniziato allo stato 8, mantenendo i due segnali di controllo (CLK e SI) invariati, e impostando come stato futuro lo stato 2.

[1]Si tiene a precisare che la conversione analogico-digitale avviene utilizzando le funzioni e le modalità descritte in 2.6 (si faccia riferimento in particolare ai listati 2.13 e 2.14).

3.4 Funzioni Ausiliarie

Il codice per l'elaborazione delle immagini verrà descritto approfonditamente alla fine (cfr. 6.7). Qui si illustreranno invece una serie di funzioni ausiliarie che permettono di eseguire alcune operazioni senza interferire con l'acquisizione dei dati.

Le operazioni che si possono voler compiere sono principalmente la modifica del tempo di integrazione, la disabilitazione temporanea delle telecamere e l'accesso diretto alle immagini acquisite.

Tutte queste operazioni, se venissero eseguite direttamente, potrebbero andare a interferire con il funzionamento dei sensori. Ad esempio, se si disabilitasse direttamente il timer che genera gli interrupt, si potrebbe bloccare a metà un'acquisizione dati. Una volta che questa dovesse riprendere, un'immagine conterrebbe informazioni altamente distorte, in quanto una parte dei pixel sarebbe stata acquisita prima dell'arresto dell'acquisizione, mentre l'altra parte conterrebbe pixel che sarebbero, con buona probabilità, tutti in saturazione, o che potrebbero riferirsi a un'immagine diversa.

Un altro problema si riscontrerebbe nell'accesso diretto alle immagini contenute nei buffer: se le telecamere fossero attive, i pixel potrebbero cambiare mentre un'altra funzione utilizza i buffer. Chiaramente questo costituisce un problema, e bisogna dunque trovare un modo per eseguire correttamente tali operazioni.

Il meccanismo chiave con cui si è affrontata la questione consiste in buona sostanza nel rimandare tutte le operazioni al momento in cui ci si trova nello stato 8 (cfr. figura 3.4), eseguendo le istruzioni necessarie contemporaneamente alla procedura di elaborazione.

Vengono innanzitutto dichiarate delle variabili ausiliarie e definite delle macro di supporto:

- `volatile unsigned char camera_enabling_request`: questa variabile notifica al sistema che si vuole cambiare lo stato di funzionamento delle telecamere (attivandole se non lo sono già, o disattivandole se sono in funzione)
- `volatile unsigned int nuovo_tempo_integrazione`: se diverso da zero, indica il valore del tempo di integrazione che si vuole impostare
- `volatile unsigned char camBufferDX[128]` e `volatile unsigned char camBufferSX[128]`: servono per creare una copia delle immagini acquisite
- `volatile unsigned char camBufferDX_request` e `volatile unsigned char camBufferSX_request`: servono per poter copiare i dati delle acquisizioni dentro ai buffer ausiliari, notificando lo stato della richiesta
- `#define CAM_BUFFER_REQUEST_READY 0`: questa macro serve per indicare che il buffer è pronto per la copia
- `#define CAM_BUFFER_REQUEST_REQUEST 1`: indica che la richiesta di copia è stata fatta, ma che ancora non è stata portata a termine

- `#define CAM_BUFFER_REQUEST_COMPLETE 2`: indica che la copia del buffer è completa

Sono quindi implementate le funzioni che fanno richiesta di modificare un parametro. Esse sono riportate nei listati 3.3 e 3.4. Queste si occupano di impostare correttamente i valori delle variabili ausiliarie (ad esempio, se si vuole fare una copia del buffer sinistro, la relativa funzione imposterà `camBufferSX_request` al valore `CAM_BUFFER_REQUEST_REQUEST`).

Dopo la richiesta, il compito vero e proprio non viene eseguito immediatamente, aspettando di arrivare alla fine di un'acquisizione. In tale punto si provvede a:

- modificare il tempo di integrazione (ovviamente si è fatto in modo che la procedura di acquisizione non cicli ulteriormente a causa del cambiamento)
- disabilitare il timer (resettando il bit TEN nel registro PIT_TCTRL1)
- copiare il buffer sinistro, se la richiesta è attiva, e di conseguenza impostare il valore di `camBufferSX_request` a `CAM_BUFFER_REQUEST_COMPLETE`; lo stesso viene fatto con il buffer destro

Due precisazioni sono ora necessarie. Qualora il timer venisse disabilitato, non sarebbe possibile riattivarlo all'interno del gestore del PIT. La riattivazione viene allora compiuta direttamente dalla relativa funzione di richiesta, che provvede anche a resettare lo stato a 0. Inoltre, siccome la prima acquisizione sarebbe quasi sicuramente in saturazione, provvede a settare una variabile che indica alla funzione di elaborazione delle immagini di scartare la prima lettura.

Per quanto riguarda la copia dei buffer, il meccanismo è simile a quello dei metodi di gestione dell'ADC: la richiesta di copia può essere effettuata solamente se il valore attuale di `camBufferDX_request` (o `camBufferSX_request`) è pari a `CAM_BUFFER_REQUEST_READY`. Questo implica anche che, anche una volta completata la copia, non sarà possibile fare ulteriori richieste fino a che il segmento di codice che utilizza l'array clonato non modifica il valore da `CAM_BUFFER_REQUEST_COMPLETE` a `CAM_BUFFER_REQUEST_READY`.

Listato 3.3: Funzioni ausiliarie per le Telecamere (parte 1)

```
1  void disableCamera() {
2    camera_enabling_request = 1;
3  }
4
5  void enableCamera() {
6    // verifica che il Timer 0 sia disabilitato
7    if(PIT_TCTRL0 | PIT_TCTRL_TEN_MASK) {
8      // abilita il Timer 0
9      PIT_TCTRL0 |= PIT_TCTRL_TEN_MASK;
10     // segnala di scartare la prima immagine acquisita
11     discard_flag = 1;
12     // resetta lo stato
13     stato = 0;
14   }
15 }
16
17 void setTempoIntegrazione(unsigned int t) {
18   // controlla se il tempo passato ha un valore valido
19   if(t>130 && t<TEMPO_INTEGRAZIONE_MASSIMO)
20     nuovo_tempo_integrazione = t;
21   else
22     nuovo_tempo_integrazione = 0;
23 }
```

Listato 3.4: Funzioni ausiliarie per le Telecamere (parte 2)

```
1  void makeBufferSXrequest() {
2    if(camBufferSX_request == CAM_BUFFER_REQUEST_READY)
3      camBufferSX_request = CAM_BUFFER_REQUEST_REQUEST;
4  }
5
6  void clearBufferSXrequest() {
7    if(camBufferSX_request == CAM_BUFFER_REQUEST_COMPLETE)
8      camBufferSX_request = CAM_BUFFER_REQUEST_READY;
9  }
10
11 void makeBufferDXrequest() {
12   if(camBufferDX_request == CAM_BUFFER_REQUEST_READY)
13     camBufferDX_request = CAM_BUFFER_REQUEST_REQUEST;
14 }
15
16 void clearBufferDXrequest() {
17   if(camBufferDX_request == CAM_BUFFER_REQUEST_COMPLETE)
18     camBufferDX_request = CAM_BUFFER_REQUEST_READY;
19 }
20
21 unsigned char getBufferSXrequest() {
22   return camBufferSX_request;
23 }
24
25 unsigned char getBufferDXrequest() {
26   return camBufferDX_request;
27 }
```

Capitolo 4

Comunicazione SPI e WINB25Q

L'ambiente utilizzato per la programmazione del microcontrollore è chiamato *Code Warrior*. Questo IDE mette a disposizione dei programmatori un'interessante funzionalità di debug, che permette di controllare la correttezza del codice in fase di esecuzione.

Tra le diverse azioni possibili, una di grande utilità consiste nell'impostazione di semafori di interruzione del codice. Quando il programma viene lanciato, Code Warrior ne monitora l'esecuzione, e, qualora venga raggiunto un *break-point* (letteralmente, *punto di interruzione*), il microcontrollore entra in uno stato di congelamento: le istruzioni non vengono più eseguite, e i valori interni rimangono invariati. Manualmente si può dunque ripristinare il funzionamento normale del processore, che riprende a eseguire le istruzioni in serie fino a che non raggiunge un altro punto di stop. In alternativa si possono iniziare a eseguire le istruzioni una alla volta, avendo così la possibilità di valutarne gli effetti in maniera agevole e semplice.

Altre due funzioni molto utili, sono quelle di monitoraggio delle variabili e dei registri. La prima permette di specificare il nome di alcuni valori di cui si vuole studiare l'evoluzione (ad esempio potrebbe essere il valore attualmente impostato per controllare l'angolo di sterzo del veicolo). La seconda serve invece a mostrare il contenuto dei singoli registri della Freedom Board, e permette di controllare in maniera rapida e sicura se sono stati commessi errori nella scrittura dei bit di impostazione.

Tutte le tecniche descritte presentano però un limite fondamentale: non sono adatte a monitorare il codice che si interfaccia con i dispositivi collegati. Ad esempio, se si utilizzasse tale approccio per il funzionamento delle telecamere, le immagini acquisite sarebbero sempre in saturazione (in quanto il programmatore è troppo lento per seguirle in tempo reale).

Vi è poi anche una questione di percezione che rende inadeguati gli strumenti messi a disposizione da Code Warrior: non sempre le variabili riescono a esprimere efficacemente il loro significato se visualizzate nel formato numerico originario. Si consideri ad esempio la sequenza di valori seguente:

```
 54  58  66  75  85  95 105 114 124 132 141 147 157 163 168 178
182 185 192 191 202 202 204 207 209 210 215 218 218 223 222 229
228 232 230 235 238 238 236 243 243 245 245 247 248 250 246 250
237 248 248 249 247 249 248 245 251 246 249 246 245 248 246 247
245 245 243 243 236 187 124  80  74  74  73  73  73  73  74  75
 75  76  97 155 211 243 245 244 245 241 241 242 239 237 237 236
235 232 231 229 229 225 221 223 216 217 212 208 207 205 203 199
196 190 185 178 173 167 158 153 143 136 122 112 103  92  83  82
```

Questa non è altro che l'immagine 3.3(a) riportata sotto forma di sequenza numerica. Come si capisce, l'interpretazione in questo caso è abbastanza ardua, a differenza di quanto accade con la sua corrispondente versione grafica.

Per tutte le ragioni elencate, si è cercato di trovare un modo per ampliare le possibilità della scheda in termini di analisi dei dati raccolti, concentrandosi in particolare sull'esigenza di poter estrarre le immagini acquisite dalle telecamere in modo da visualizzarle in un secondo momento. Tra le varie opzioni prese in considerazione, si è scelto di ricorrere all'uso di una memoria di tipo Flash, grazie alla quale i dati possono essere immagazzinati a lungo termine anche dopo l'arresto della scheda. Il dispositivo comunica mediante il protocollo SPI, che verrà adesso illustrato nel dettaglio.

Prima di passare oltre, si vuole però fare una precisazione: uno dei motivi per cui si è scelto un simile componente, è che questo può essere utilizzato in molti modi, oltre che come strumento di debug. Un primo esempio consiste nella possibilità di memorizzare diversi set di taratura dei parametri caratteristici dei controllori digitali, in modo da adattare il controllo alle diverse situazioni che si possono presentare. In aggiunta, si potrebbe impostare la macchina in modo tale che, mentre affronta il tracciato, tenga in memoria i dati relativi ai tratti già percorsi, per poi riutilizzarli in un secondo momento al fine di migliorare le prestazioni di gara a un successivo giro di pista.

4.1 Serial Peripheral Interface

Il protocollo di comunicazione SPI costituisce una semplice ed efficace interfaccia per la trasmissione di dati digitali tra diversi dispositivi. Essa prevede la presenza di un circuito denominato *master*, che gestisce tutti gli aspetti di sincronizzazione, e di uno o più circuiti denominati *slave*.

Un limite di questo protocollo consiste nell'impossibilità di due dispositivi slave di comunicare direttamente tra loro: in ogni istante lo scambio dei dati può avvenire solamente tra il master e uno slave specifico. Inoltre, per come è strutturata l'interfaccia, i dispositivi non possono trasmettere le informazioni nel momento in cui qeste sono disponibili, ma devono aspettare che il master decida di scambiare informazioni con questi.

4.1.1 Schema dell'Interfaccia e Funzionamento

In figura 4.1 si mostra lo schema con cui si interfacciano i dispositivi all'interno della comunicazione SPI. Come si può vedere, la comunicazione utilizza tre linee comuni per collegare tutti i dispositivi, e in aggiunta è presente una connessione tra ogni dispositivo slave e il master.

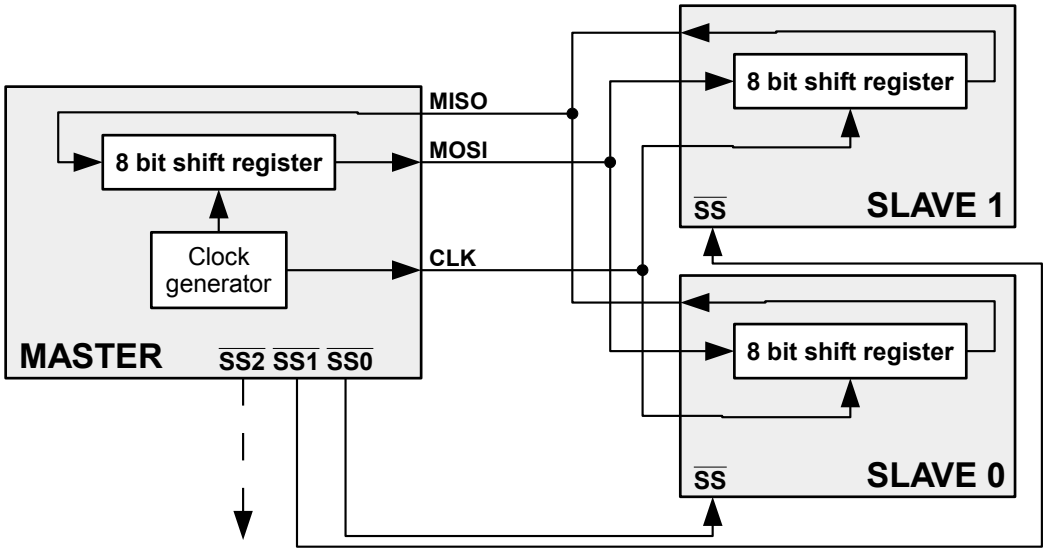


Figura 4.1: Schema di interfacciamento del protocollo SPI

I terminali *SS*, acronimo per *Slave Select*, permettono di comunicare alle diverse periferiche se si intende comunicare con loro. Si noti che il segnale è utilizzato in logica negata, con ciò intendendo che per l'attivazione della corrispondente funzione la tensione deve essere portata a 0, mantenendola invece a V_{DD} fintanto che un dispositivo deve rimanere inattivo.

La ragione di questo meccanismo trova spiegazione nel modo in cui le altre linee di comunicazione sono progettate: esse connettono diversi dispositivi slave, ciascuno dei quali, in fase di comunicazione, potrebbe voler inviare informazioni utili. Facendo riferimento allo schema di figura 4.1, si supponga ad esempio che lo slave 0 voglia inviare uno "0" al master, mentre lo slave 1 intenda trasmettere un "1". Le tensioni, come verrà illustrato in seguito, vengono imposte dalle periferiche sulla linea denominata *MISO*: su tale collegamento i due dispositivi slave risulterebbero cortocircuitati, con il conseguente dannegiamento dei componenti. Tuttavia, tramite gli Slave Select, il master potrà imporre che solo una periferica alla volta possa trasmettere informazioni, costringendo tutti gli slave non selezionati a porre i terminali di comunicazione in stato di alta impedenza.

Una volta che il master abilita uno slave, i dati possono essere scambiati in contemporanea tramite le due linee dati *MOSI* e *MISO*. Il primo corrisponde alla linea che il master utilizza per inviare i bit alle periferiche (l'acronimo sta infatti per *Master-Out, Slave-In*); al contrario il secondo (come già accennato) corrisponde al terminale di uscita dei dati degli slave (*Master-In, Slave-Out*).

Un terzo segnale che, come i due precedenti, collega tutti i dispositivi tra loro, è il segnale di clock (*CLK* in figura), il cui scopo consiste nella sincronizzazione dello scambio dei dati.

I moduli che possono comunicare tramite l'interfaccia SPI sono provvisti, oltre che dei terminali descritti, di un registro a scorrimento a 8 bit, fondamentale per il funzionamento della comunicazione. Quando il master porta a 0 uno dei pin SS, i registri a scorrimento dei due moduli comunicanti risultano connessi in modo tale da formare un registro a scorrimento circolare da 16 bit. In tali registri devono essere dunque caricate le informazioni da inviare, e la trasmissione può iniziare. Il master pilota il segnale di clock in maniera tale che tutti gli otto bit dei registri vengano shiftati nel registro dell'altra periferica, quindi (se non è richiesto nessun'altro scambio di informazioni) il pin SS viene riportato alla tensione di alimentazione, concludendo la procedura di comunicazione.

4.1.2 Sincronizzazione

Durante la comunicazione, i bit vengono fatti scorrere da un registro all'altro sincronamente al segnale di clock. A seconda delle scelte costruttive dei produttori dei diversi dispositivi, i registri a scorrimento sono progettati per essere sensibili a ben determinate transizioni del segnale di sincronismo. Pertanto, in fase di progettazione di un sistema di comunicazione è importante conoscere a priori come deve essere gestita la trasmissione dei dati, in modo da sincronizzare correttamente i registri del master e dello slave quando avviene la comunicazione.

Per il clock vengono dunque definiti due parametri fondamentali: *CPOL* e *CPHA*. Il primo rappresenta la *polarità* del segnale, e cioè indica se la trasmissione del dato avviene in corrispondenza di una transizione positiva oppure negativa, e che valore deve avere il clock prima e dopo la trasmissione.

Il valore CPHA indica invece se il singolo dato viene trasmesso in corrispondenza del periodo del clock, oppure se tale evento si verifica con un certo sfasamento temporale rispetto al segnale di sincronismo (avvenendo in pratica a metà del periodo del clock).

Si guardi ora la figura 4.2. Essa mostra le tracce temporali dei segnali coinvolti nella trasmissione di un byte nel caso in cui sia CPHA=0.

In questa immagine si distinguono diversi segnali: i primi due mostrano il clock nei due casi in cui CPOL valga 0 oppure 1. Il terzo segnale Mostra invece l'istante in cui il dato di un registro viene salvato per essere trasferito. Seguono poi i due segnali MOSI e MISO. Infine si mostra il segnale di controllo SS.

Come si può osservare, nel caso in cui si abbia CPHA pari a 0 non appena il segnale SS viene portato a 0 inizia la procedura di trasmissione dei dati. Al primo fronte di clock (positivo nel caso in cui CPOL=0, negativo altrimenti) il dato viene prelevato, e al successivo fronte viene completata l'operazione di scorrimento.

Si noti in questo caso come il clock sia sfasato rispetto ai segnali MOSI e MISO.

Facendo attenzione ai segnali SS (nel caso in cui la scheda si comporti da slave) ci si accorge inoltre di un fatto molto rilevante: dopo la trasmissione del byte,

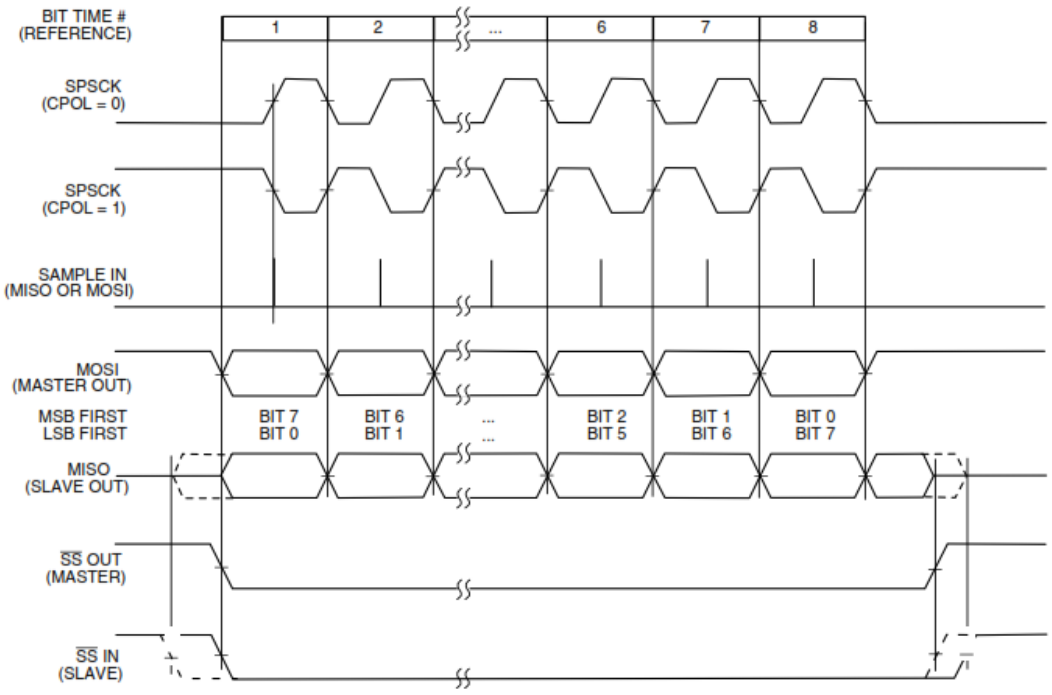


Figura 4.2: Sincronizzazione SPI (CPHA=0)

tale pin non deve rimanere basso, ma deve necessariamente ritornare allo stato di congelamento del dispositivo. Questo significa, nella pratica, che non è possibile compiere trasferimenti multipli lasciando il segnale SS sempre a zero.

Con riferimento alla figura 4.3, si consideri ora CPHA=1. Qui si nota che, nell'istante in cui viene selezionata la periferica per lo scambio delle informazioni, i bit presenti nei registri non vengono ancora prelevati. Questo accade invece in concomitanza con la prima transizione di clock (che può essere positiva o negativa a seconda di CPOL). Con la successiva transizione il dato viene campionato, e all'inizio del nuovo periodo lo shift viene portato a termine, rendendo disponibile il successivo bit.

A differenza del caso precedente, non è necessario disattivare la periferica tra la ricezione di un byte e l'altro, permettendo così il trasferimento di più informazioni senza dover cambiare lo stato di SS.

In letteratura, le quattro possibili combinazioni di CPOL e CPHA sono nominate con un numero progressivo da 0 a 3, e sovente nei datasheet dei componenti si trova una dicitura del tipo

"Il dispositivo supporta per la comuniczione SPI le modalità 0 e 2"

La tabella 4.1 mostra dunque le diverse modalità del clock.

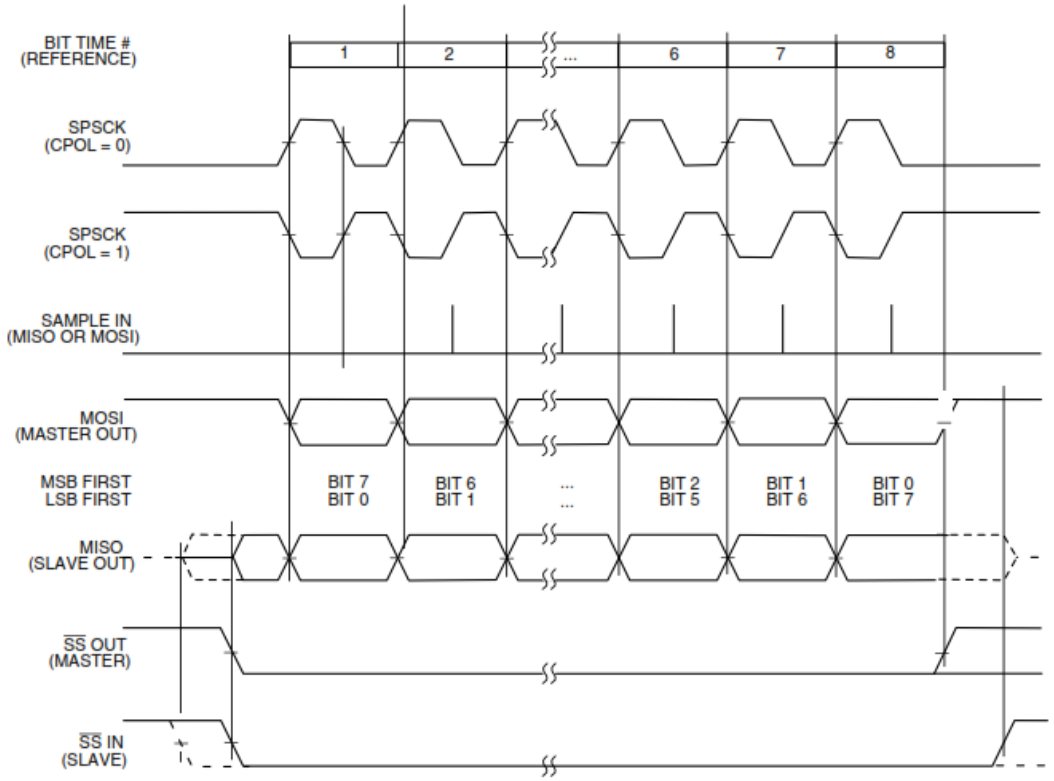


Figura 4.3: Sincronizzazione SPI (CPHA=1)

Modalità	**CPOL**	**CPHA**
mode 0	0	0
mode 1	0	1
mode 2	1	0
mode 3	1	1

Tabella 4.1: Modalità di clock nel protocollo SPI

4.2 Modulo SPI su KL25Z

La Freedom Board integra al suo interno un modulo dedicato per la comunicazione SPI. La versatilità di questo microcontrollore permette di implementare due canali separati per lo scambio di informazioni, ciascuno con i suoi terminali MOSI, MISO e CLK. Sebbene non sia necessaria la presenza di più linee (poiché come si è visto basta un'unica interfaccia per comunicare con diversi dispositivi slave), tale opzione permette di utilizzare al meglio le funzioni implementate dalla scheda, in quanto i pin dedicati a un modulo potrebbero essere già occupati per altre funzioni.

Segue una descrizione che presenta la periferica innanzitutto dal punto di vista dell'hardware; successivamente si mostrerà la libreria di utilità scritta per permettere di utilizzare in maniera agevole l'interfaccia di comunicazione.

4.2.1 Descrizione del Modulo

Il microcontrollore KL25Z monta due moduli identici che realizzano l'interfaccia SPI. La struttura di tali moduli è più complessa del semplice schema di figura 4.1, in quanto oltre al generatore del clock e al registro a scorrimento sono presenti tutta una serie di registri per l'impostazione di diversi parametri. Tali parametri sono impostabili, come al solito, tramite i registri del microcontrollore. Questi sono visibili in tabella 4.2.

Tabella 4.2: Registri SPI

Registro/bit	Descrizione
SPIx_C1	**SPI Control Register 1, contiene alcuni bit di controllo per il modulo.**
SPIE [7]	SPI Interrupt Enable, permette di abilitare la generazione degli interrupt quando avvengono gli eventi *SPI receive buffer full* o *mode fault*.
SPE [6]	SPI System Enable, utilizzato per attivare/disabilitare il funzionamento del modulo.
SPTIE [5]	SPI Transmit Interrupt Enable, bit che consente di abilitare la generazione di un interrupt qualora si verifichi l'evento *SPI transmit buffer empty*.
MSTR [4]	Master/Slave Mode Select, configura il ruolo del microcontrollore durante la comunicazione (master oppure slave).
CPOL [3]	Clock Polarity, permette di impostare la polarità del clock (cfr. 4.1.2).
CPHA [2]	Clock Phase, utilizzato per decidere se la trasmissione deve avvenire in fase con il clock o meno (cfr. 4.1.2).

continua nella pagina successiva

continua dalla pagina precedente

Registro/bit	Descrizione
SSOE [1]	Slave Select Output Enable, seleziona la funzione da associare al pin SS (il comportamento del modulo dipende anche dal valore dei bit SPIx_C1_MSTR e SPIx_C2_MODFEN.
LSBFE [0]	LSB First, permette di decidere se il byte da trasmettere deve iniziare dal bit meno significativo o dal più significativo.
SPIx_C2	**SPI Control Register 2, contiene alcuni bit di controllo per il modulo.**
SPMIE [7]	SPI Match Interrupt Enable, abilita la generazione degli interrupt quando si verifica l'evento *SPI receive data buffer hardware match*.
TXDMAE [5]	Transmit DMA Enable, abilita la trasmissione mediante Direct Memory Access per questo modulo.
MODFEN [4]	Master Mode-Fault Function Enable, permette di modificare la funzone del pin SS (il comportamento del modulo dipende anche dal valore dei bit SPIx_C1_MSTR e SPIx_C2_SSOE.
BIDIROE [3]	Bidirectional Mode Output Enable, il bit seleziona la funzione della linea di trasmissione quando è attiva la comunicazione bidirezionale, e cioè una modalità di funzionamento particolare in cui si utilizza un solo terminale di collegamento per il trasferimento dei dati.
RXDMAE [2]	Receive DMA Enable, abilita la ricezione mediante Direct Memory Access per questo modulo.
SPISWAI [1]	SPI Stop in Wait Mode, configura il comportamento del modulo quando il dispositivo entra in modalità di attesa.
SPC0 [0]	SPI Pin Control 0, permette di abilitare la comunicazione bidirezionale.
SPIx_BR	**SPI Baud Rate Register, registro per l'impostazione della frequenza del clock di trasmissione.**
SPPR [6-4]	SPI Baud Rate Prescaler Divisor, imposta il fattore di prescale per il clock.
SPR [3-0]	SPI Baud Rate Divisor, imposta il fattore di divisione per il clock.
SPIx_S	**SPI Status Register, contiene 4 bit di stato**
SPRF [7]	SPI Read Buffer Full Flag, flag che viene settato quando la ricezione del byte è completa.
SPMF [6]	SPI Match Flag, flag che viene settato quando il dato ricevuto coincide con il byte scritto nel registro SPIx_ M.

continua nella pagina successiva

continua dalla pagina precedente

Registro/bit	Descrizione
SPTEF [5]	SPI Transmit Buffer Empty Flag, flag che viene settato quando il byte è stato inviato.
MODF [4]	Master Mode Fault Flag, questo bit viene settato quando un altro dispositivo porta il pin SS a zero, situazione che si verifica nel caso in cui per qualche ragione un altro dispositivo si comporta da master.
SPIx_D	**SPI Data Register, il registro serve a contenere il byte di uscita prima della trasmissione, nonché quello di entrata quando la trasmissione è completa.**
Bits [7-0]	Data, il byte da trasmettere o ricevuto.
SPIx_M	**SPI Match Register, il registro serve a contenere il byte utilizzato per la funzione di matching.**
Bits [7-0]	Hardware Compare Value, il byte ivi contenuto viene utilizzato per controllare se i bit ricevuti coincidono a questo valore particolare, e, in tal caso, viene settato il bit SPIx_S_SPMF.

Per impostare il funzionamento dei moduli, si utilizzano principalmente i registri *SPIx_C1* e *SPIx_C2*. Nel primo, vi sono i bit *CPOL* e *CPHA* che permettono di impostare il formato del clock secondo gli schemi presentati in precedenza. Il bit *MSTR* permette invece di decidere se il dispositivo si comporterà da master o da slave.

Il bit *SPE* permette invece di disabilitare temporaneamente le diverse funzioni del modulo (si faccia attenzione che il modulo rimane comunque attivo finché il bit corrispondente nel System Integration Module non viene resettato).

I due bit *SPTIE* e *SPIE*, insieme con il bit *SPMIE* nel registro SPIx_C2, permettono di abilitare gli interrupt al verificarsi di diversi eventi. Come si è accennato, la struttura interna del modulo SPI è più complessa dello schema generale di un dispositivo di interfaccia SPI. In particolare, il modulo contiene due registri interni aggiuntivi per la trasmissione dei dati: il primo corrisponde a un buffer di trasmissione (*Tx BUFFER*), mentre il secondo a un buffer di ricezione (*Rx BUFFER*). Un ulteriore registro viene utilizzato per caricare e prelevare i dati di trasmissione: questo prende il nome di *SPI Data Register*, o, in forma abbreviata, *SPIx_D*.

Il programmatore, per avviare una trasmissione, deve innanzitutto caricare i dati nel registro dati (SPIx_D), e il byte viene quindi trasferito all'interno del buffer di trasmissione dal modulo. Appena possibile, il dato viene trasferito direttamente nel registro a scorrimento principale, pronto per trasmettere i dati alla periferica. Quando tale buffer si svuota, il flag SPTEF viene settato, e, se SPIE è pari a 1, viene generato un interrupt. A questo punto è possibile caricare un nuovo dato nel buffer di trasmissione, anche se non si è ancora completata la precedente procedura di

trasmissione.

I dati, una volta shiftati, vengono invece trasferiti nel buffer di ricezione. Questo causa il settaggio del flag *SPRF* e la generazione di un interrupt (sempre che SPIE sia settato). I dati sono quindi disponibili leggendo dal registro SPIx_D (in cui viene trasferito il contenuto del buffer di ricezione quando viene richiesta la lettura dei dati).

In aggiunta, non appena i dati entrano nel RxBUFFER, il byte acquisito viene confrontato con l'eventuale valore caricato nel registro *SPIx_M* (*SPI Match Register*). Se i due byte coincidono, viene settato il flag *SPMF*, e viene generato l'interrupt corrispondente nel caso in cui sia stato precedentemente settato SPMIE.

Il modulo si occupa inoltre di gestire la funzione associata allo Slave Select mediante la selezione opportuna dei bit *SSOE* e *MODFEN* (il comportamento dipende anche dal ruolo giocato dalla Freedom Board, se master oppure slave). In linea di principio, si potrebbe utilizzare come SS un qualunque pin della scheda collegandolo alla funzione GPIO, e pilotandolo in maniera opportuna durante le comunicazioni. Questo è infatti quello che si fa normalmente, e tuttavia i prgettisti della Freescale hanno pensato di associare al modulo SPI un pin SS dedicato. Questo può essere utilizzato in meniera "classica" come Slave Select collegandolo a una periferica e pilotandolo tramite funzione GPIO, oppure vi si può associare qualche funzione particolare.

Tra le modalità di utilizzo presenti, ve ne è una grazie a cui il pin funziona a tutti gli effetti come SS, ma con la particolare caratteristica che il suo stato viene impostato automaticamente dalla scheda a seconda che sia stato richiesto di inviare un dato o meno. Questa funzione si attiva quando MSTR=1, SSOE=1, MODFEN=1.

Una seconda funzione particolare è quella denominata *Mode Fault*, che si seleziona impostando MSTR=1, SSOE=0 e MODFEN=1. In questo caso, il pin SS viene utilizzato per controllare se un dispositivo ad esso collegato per qualche ragione si stia comportando da master: in tal caso sarebbe opportuno commutare immediatamente allo stato di slave onde evitare che le linee MISO, MOSI e CLK vengano utilizzate come output da più dispositivi, creando un cortocircuito potenzialmente distruttivo. Pertanto, quando un dispositivo porta il pin SS a zero, il modulo provvede subito a portarsi in modalità slave resettando MSTR (interompendo inoltre qualunque comunicazione in corso), e a notificare l'evento settando il flag *MODF*. Infine viene generato un interrupt, a patto che il bit SPIE sia pari a 1.

4.2.2 Libreria C per la Comunicazione SPI

La comunicazione tramite SPI può essere avviata semplicemente utilizzando le macro di accesso e modifica dei registri contenute nel file header fornito dal costruttore, tuttavia questo procedimento risulta scomodo e relativamente complesso qualora si debba utilizzare il modulo di frequente.

La soluzione software adottata è consistita nella scrittura di una libreria di utilità che semplifica il codice da scrivere al fine di avviare la trasmissione dei dati. Tale libreria consiste in un file header in cui sono riportati i prototipi delle diverse funzioni, e in

un file sorgente contenente l'implementazione delle stesse. Il file header è riportato nel listato 4.1.

Prima di procedere alla descrizione dettagliata dei metodi implementati, si vogliono riportare qui le scelte progettuali che si sono fatte. Innanzitutto, la libreria permette l'utilizzo della scheda solamente come dispositivo master. Questo in quanto durante la realizzazione del progetto non si è presentata la necessità di far funzionare la Freedom Board come slave.

In secondo luogo, si è scelto di limitare il numero di periferiche collegabili a quattro. La scelta deriva dal semplice fatto che conoscendo in anticipo i pin da riservare come SS lo sviluppo dei metodi si rendeva meno intricato. Inoltre, non si volevano occupare troppe porte, in quanto costituiscono risorse preziose. Quattro dispositivi - e quindi quattro pin - sono sembrati un compromesso ragionevole tra funzionalità e semplicità.

Infine, si è usato come formato del clock *mode 0*. Questo poichè il dispositivo Flash adoperato permetteva l'uso delle sole modalità 0 e 3. La differenza sostanziale tra di essi risiede nel fatto che nel secondo caso sia possibile effettuare diversi cicli di comunicazione (con la scheda in modalità slave) senza dover ogni volta riportare in alto il pin SS. La scelta di utilizzare la Freedom Board unicamente come master annulla però queste differenze, e si è scelto arbitrariamente di utilizzare il primo dei due formati.

Listato 4.1: File header spi.h

```
1  #ifndef SPI_H_
2  #define SPI_H_
3
4  #define SPI_SLAVE1 0x01
5  #define SPI_SLAVE2 0x02
6  #define SPI_SLAVE3 0x03
7  #define SPI_SLAVE4 0x04
8
9  void setupSPI();
10 unsigned char ss_error_occurred();
11 unsigned char sendSPIdata(unsigned char data, unsigned char slave);
12 void sendSPIbuffer(unsigned char *buffer, unsigned int buffer_length, unsigned
       char slave);
13
14 #endif
```

Passando all'analisi del file header contenuto in 4.1, si trova che all'inzio vengono definite quattro macro: `SPI_SLAVE1`, `SPI_SLAVE2`, `SPI_SLAVE3` e `SPI_SLAVE4`. Queste permettono di identificare i quattro dispositivi che possono essere connessi alla Freedom Board.

Segue poi il metodo `setupSPI`, che inizializza correttamente i bit del modulo (si faccia riferimento al codice 4.2). I pin utilizzati per la funzione Slave Select sono `PTD0`, `PTD2`, `PTD3` e `PTD4`; questi vengono subito posti a V_{DD} in modo da deselezionare tutte le periferiche da subito. La scheda viene impostata come master, con modalità di

clock pari a 0. Per quanto riguarda la frequenza del clock, il fattore di prescale viene posto pari a 6, mentre il fattore di divisione è paria 8. Considerato che il modulo riceve in ingresso il clock di bus (24 MHz), si ha $CLK_{SPI} = CLK_{BUS}/48 = 500\,kHz$.

Listato 4.2: Funzione `setupSPI`

```
1  void setupSPI() {
2    ss_error_flag = 0;
3
4    SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK; // clock alle porte C
5    SIM_SCGC5 |= SIM_SCGC5_PORTD_MASK; // clock alle porte D
6
7    PORTC_PCR5 |= PORT_PCR_MUX(2); // imposta PTC5 come CLK
8    PORTC_PCR6 |= PORT_PCR_MUX(2); // imposta PTC6 come MOSI
9    PORTC_PCR7 |= PORT_PCR_MUX(2); // imposta PTC7 come MISO
10
11   // SET DEL PIN SS1 (PTD0)
12   PORTD_PCR0 |= PORT_PCR_MUX(1); // imposta PTD0 come SS1
13   GPIOD_PDDR |= 1<<0; // imposta PTD0 (SS1) in output
14   SS1_HIGH; // deseleziona lo slave1
15
16   // SET DEL PIN SS2 (PTD2)
17   PORTD_PCR2 |= PORT_PCR_MUX(1); // imposta PTD2 come SS2
18   GPIOD_PDDR |= 1<<2; // imposta PTD2 (SS2) in output
19   SS2_HIGH; // deseleziona lo slave2
20
21   // SET DEL PIN SS3 (PTD3)
22   PORTD_PCR3 |= PORT_PCR_MUX(1); // imposta PTD3 come SS3
23   GPIOD_PDDR |= 1<<3; // imposta PTD3 (SS3) in output
24   SS3_HIGH; // deseleziona lo slave3
25
26   // SET DEL PIN SS4 (PTD4)
27   PORTD_PCR4 |= PORT_PCR_MUX(1); // imposta PTD4 come SS4
28   GPIOD_PDDR |= 1<<4; // imposta PTD4 (SS4) in output
29   SS4_HIGH; // deseleziona lo slave4
30
31   SIM_SCGC4 |= SIM_SCGC4_SPI0_MASK; // clock al modulo SPI0
32   SPI0_C1 = 0x10; // master, no interrupt, mode 0, SS=GPIO
33   SPI0_BR |= SPI_BR_SPPR(5); // bus_clock/6 (=4MHz)
34   SPI0_BR |= SPI_BR_SPR(2); // (bus_clock/SPPR)/8 (=500kHz)
35   SPI0_C1 |= SPI_C1_SPE_MASK; // abilita il modulo SPI
36 }
```

La seconda funzione che si incontra nell'header è chiamata `ss_error_occurred`. Questa restituisce 1 se l'ultimo tentativo di invio dei dati non è stato completato in quanto il dispositivo slave passato alle funzioni `sendSPIdata` o `sendSPIbuffer` differiva dal codice dei quattro dispositivi slave selezionabili. Al contrario, restituisce 0 se il dispositivo era stato indicato correttamente.

Si incontra dunque la funzione `sendSPIdata` (listato 4.3). Questa permette di inviare il byte `data` (un valore a 8 bit) al dispositivo `slave` indicato (per il valore di quest'ultimo, si consiglia di utilizzare le quattro macro definite all'inizio dell'header).

Innanzitutto, viene pulita la variabile `ss_error_flag`, che permette di tenere traccia degli errori dovuti al passaggio di un valore non concesso per `slave`. Successivamente, si utilizza la funzione ausiliaria `selectSlave` per portare a 0 lo Slave Select del dispositivo indicato (la funzione è mostrata in 4.5). Questo metodo restituisce il valore zero se lo slave passato non è valido: in tal caso `sendSPIdata` setta il flag di errore ed esce dal metodo restituiendo il valore 0.

Si entra dunque nel cuore del metodo: qui innanzitutto si attende che il buffer di trasmissione sia vuoto, quindi si inserisce il byte `data` nel registro SPI0_D, dando inizio alla trasmissione. Si attende che i byte siano stati scambiati, quindi vengono estratti dal registro dati e salvati nella variabile `read_data`. Il metodo ausiliario `deselectSlave` riporta il pin SS a V_{DD}, e alla fine viene restituito il dato ricevuto.

Listato 4.3: Metodo `sendSPIdata`

```
1  unsigned char sendSPIdata(unsigned char data, unsigned char slave) {
2    ss_error_flag = 0; // resetta ss_error_flag
3
4    // seleziona lo slave; se il parametro passato e' invalido,
5    // setta ss_error_flag e interrompe il metodo restituendo 0
6    if(selectSlave(slave)==0) {
7      ss_error_flag=1;
8      return 0x00;
9    }
10
11   // PROCEDURA PER L'INVIO DEI DATI TRAMITE SPI
12   // aspetta che il registro dei dati sia pronto per essere scritto
13   while((SPI0_S & SPI_S_SPTEF_MASK)==0);
14   // immette i dati nel registro, avviando la comunicazione con lo slave
15   SPI0_D = SPI_D_Bits(data);
16   // aspetta che i dati dello slave siano stati
17   // trasferiti nel buffer di ricezione dati
18   while((SPI0_S & SPI_S_SPRF_MASK)==0);
19   // recupera i dati ricevuti dallo slave
20   unsigned char read_data = SPI0_D;
21
22   deselectSlave(slave);
23
24   return read_data; // restituisce i dati ottenuti
25 }
```

Il metodo `sendSPIbuffer` esegue un compito molto simile a quello di `sendSPIdata`, con la differenza che permette di scambiare una sequenza di byte senza riportare SS a zero tra un dato e l'altro. La sequenza di dati viene passata mediante il puntatore `buffer`, di cui va indicata la lunghezza tramite la variabile `buffer_length`. Come si può vedere nel listato 4.4, la prima parte del metodo coincide quasi esattamente con quella di `sendSPIdata`.

Dovendo inviare una serie di dati, il metodo avvia un ciclo che invia in sequenza tutti i byte contenuti in `buffer`. Ogni dato ricevuto viene posizionato nello stesso array, nella posizione che occupava il dato appena scambiato. Finito il ciclo, viene

deselezionato lo slave. I dati sono quindi disponibili mediante lo stesso array che si era utilizzato come variabile di ingresso.

Listato 4.4: Funzione `sendSPIbuffer`

```
1  void sendSPIbuffer(unsigned char *buffer, unsigned int buffer_length, unsigned
       char slave) {
2    ss_error_flag = 0; // resetta ss_error_flag
3
4    // seleziona lo slave; se il parametro passato e' invalido,
5    // setta ss_error_flag e interrompe il metodo restituendo 0
6    if(selectSlave(slave)==0) {
7      ss_error_flag=1;
8      return;
9    }
10
11   // PROCEDURA PER INVIARE I DATI DEL BUFFER
12   int i;
13   for(i=0; i<buffer_length; i++) {
14     // aspetta che il registro dei dati sia pronto per essere scritto
15     while((SPI0_S & SPI_S_SPTEF_MASK)==0);
16     // immette i dati nel registro, avviando
17     // la comunicazione con lo slave
18     SPI0_D = SPI_D_Bits(buffer[i]);
19     // aspetta che i dati dello slave
20     // siano stati trasferiti nel registro D
21     while((SPI0_S & SPI_S_SPRF_MASK)==0);
22     buffer[i] = SPI0_D; // recupera i dati ricevuti dallo slave
23   }
24
25   deselectSlave(slave);
26 }
```

Si riportano per completezza i codici delle funzioni ausiliarie `selectSlave` e `deselectSlave` (rispettivamente in 4.5 e 4.6). Nel codice compaiono delle macro ausiliarie (ad esempio `SS1_LOW`) che permettono di portare i segnali SS corrispondenti a 1 o a 0 mediante GPIO.

4.3 WINB25Q

Dal momento in cui ci si è resi conto che era necessario trovare un modo per estrarre i dati dalla Freedom Board, il primo interrogativo è stato quale fosse il modo migliore. Si è valutata la possibilità di far comunicare la scheda con una periferica USB, ma dopo un po' di ricerca ci si è resi conto che il compito era abbastanza arduo, in quanto richiedeva conoscenze informatiche abbastanza approfondite. Si è pensato di utilizzare un sistema wireless, come descritto dallo studente Francesco Dal Santo in [15], oppure di collegare una scheda SD al microcontrollore.

Alla fine, si è giunti alla decisione di utilizzare un dispositivo di memoria Flash. I motivi principali sono due: innanzitutto la semplicità di funzionamento di un di-

Listato 4.5: Funzione ausiliaria selectSlave

```
unsigned char selectSlave(unaigned char slave) {
  switch(slave) {
    case SPI_SLAVE1:
      SS1_LOW; // seleziona slave1
      break;
    case SPI_SLAVE2:
      SS2_LOW; // seleziona slave2
      break;
    case SPI_SLAVE3:
      SS3_LOW; // seleziona slave3
      break;
    case SPI_SLAVE4:
      SS4_LOW; // seleziona slave4
      break;
    default:
      return 0;
      break;
  }
  return 1;
}
```

Listato 4.6: Funzione ausiliaria deselectSlave

```
unsigned char deselectSlave(unaigned char slave) {
  switch(slave) {
    case SPI_SLAVE1:
      SS1_HIGH; // deseleziona slave1
      break;
    case SPI_SLAVE2:
      SS2_HIGH; // deseleziona slave2
      break;
    case SPI_SLAVE3:
      SS3_HIGH; // deseleziona slave3
      break;
    case SPI_SLAVE4:
      SS4_HIGH; // deseleziona slave4
      break;
    default:
      return 0;
      break;
  }
  return 1;
}
```

spositivo di questo tipo permette di ottenere ottimi risultati senza dover impiegare troppo tempo nella realizzazione di una libreria software di gestione. In secondo luogo, il chip può essere utilizzato durante la competizione Freescale Cup, a differenza di altri dispositivi di acquisizione dati wireless che sono proibiti durante la gara.

Una volta deciso il tipo di dispositivo, ci si è preoccupati di trovare qualcosa di adatto, in termini di capacità di memorizzazione, costo e tecnologia di funzionamento. Alla fine della ricerca, il dispositivo che ha soddisfatto tutte queste esigenze è l'integrato *WINB25Q*. I motivi di questa scelta sono i seguenti:

- Il costo ridotto permetteva di acquistarne più di uno, nel caso in cui il primo si fosse danneggiato
- L'organizzazione delle celle di memoria è tale da permettere di salvare in maniera ottimale le immagini acquisite dalle due telecamere (infatti l'unità minima di memoria in scrittura, corrisponde a 256 byte, esattamente la dimensione di due immagini estratte dalle telecamere)
- La programmazione del dispositivo avviene tramite il protocollo SPI, che come si è visto è compatibile con la Freedom Board
- È disponibile un manuale che ne descrive dettagliatamente il funzionamento, in maniera chiara e semplice

Una volta acquistato il componente, si è proceduto con l'implementazione della libreria SPI descritta sopra, e in seguito studiando il funzionamento del dispositivo Flash, che verrà descritto ora. Infine, si è scritta una libreria, basta sulla precedente, per interfacciare il dispositivo con la Freedom Board.

A coloro che volessero utilizzare tale dispositivo in un dato progetto, si consiglia caldamente di leggere il manuale tecnico [16] (scaricabile gratuitamente dal sito del costruttore: `https://www.winbond.com/`).

4.3.1 Schema e Funzionamento del Dispositivo

Il dispositivo siglato W25Q80BV, prodotto dall'azienda Winbond, è un dispositivo di memoria Flash seriale della capacità di 1 MB, alimentato da una tensione continua compresa tra i 2.5 V e i 3.6 V.

Il dispositivo permette di salvare e leggere dati sotto forma di byte dalle sue celle di memoria, e prevede diverse funzioni atte a proteggere determinate locazioni di memoria, impedendone la cancellazione o la sovrascrittura.

In figura 4.4 viene mostrata la piedinatura dell'integrato. Questo ha otto terminali, denominati rispettivamente *CS*, *DO*, *WP*, *GND*, *DI*, *CLK*, *HOLD* e *VCC*.

Le funzioni di ciascun pin sono le seguenti:

- **CS** (**Chip Select**): corrisponde allo Slave Select del protocollo SPI.

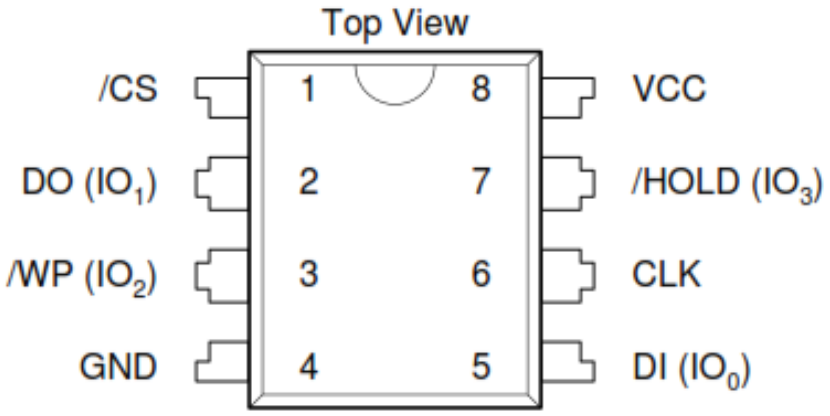


Figura 4.4: Piedinatura della memoria Flash

- **DO** (**Data Output**): terminale di output per le informazioni, corrisponde alla linea MISO.
- **WP** (**Write Protect**): pin per la funzione di protezione da scrittura. Questa permette, sotto particolari condizioni, di impedire la modifica dei dati contenuti nel registro di stato. Funziona in logica negata: la protezione si attiva quando la tensione sul terminale scende a zero.
- **GND** (**Gorund**): corrisponde alla massa del dispositivo (o meglio, costituisce il riferimento a $0\,V$).
- **DI** (**Data Input**): terminale di input per le informazioni, corrisponde alla linea MOSI.
- **CLK** (**Clock**): corrisponde alla linea omonima dell'interfaccia SPI.
- **HOLD**: pin per la funzione di sospensione temporanea della comunicazione. Funziona in logica negata.
- **VCC**: pin di alimentazione del dispositivo.

Per quanto concerne la struttura interna del chip, si osservi il diagramma a blocchi di figura 4.5. Il dispositivo si compone, principalmente, di tre blocchi fondamentali: i registri di memoria (in figura sono visibili nella parete destra, raggruppati in sedici blocchi), i registri di stato (denotati con il termine *Status Register*) e la logica di controllo e di comando (*SPI Command & Control Logic*).

Oltre ai suddetti, vi sono altri blocchi necessari per il corretto funzionamento del dispositivo, ad esempio i decodificatori di colonna e di riga (necessari per selezionare le locazioni di memoria corrette).

I registri di memoria hanno una configurazione particolare, che raggruppa le singole celle in unità di memorizzazione innestate.

La memoria principale è suddivisa in 16 gruppi da 64 KB, denominati blocchi. Le informazioni immagazzinate in ciascun blocco sono poi ripartite a loro volta in 16

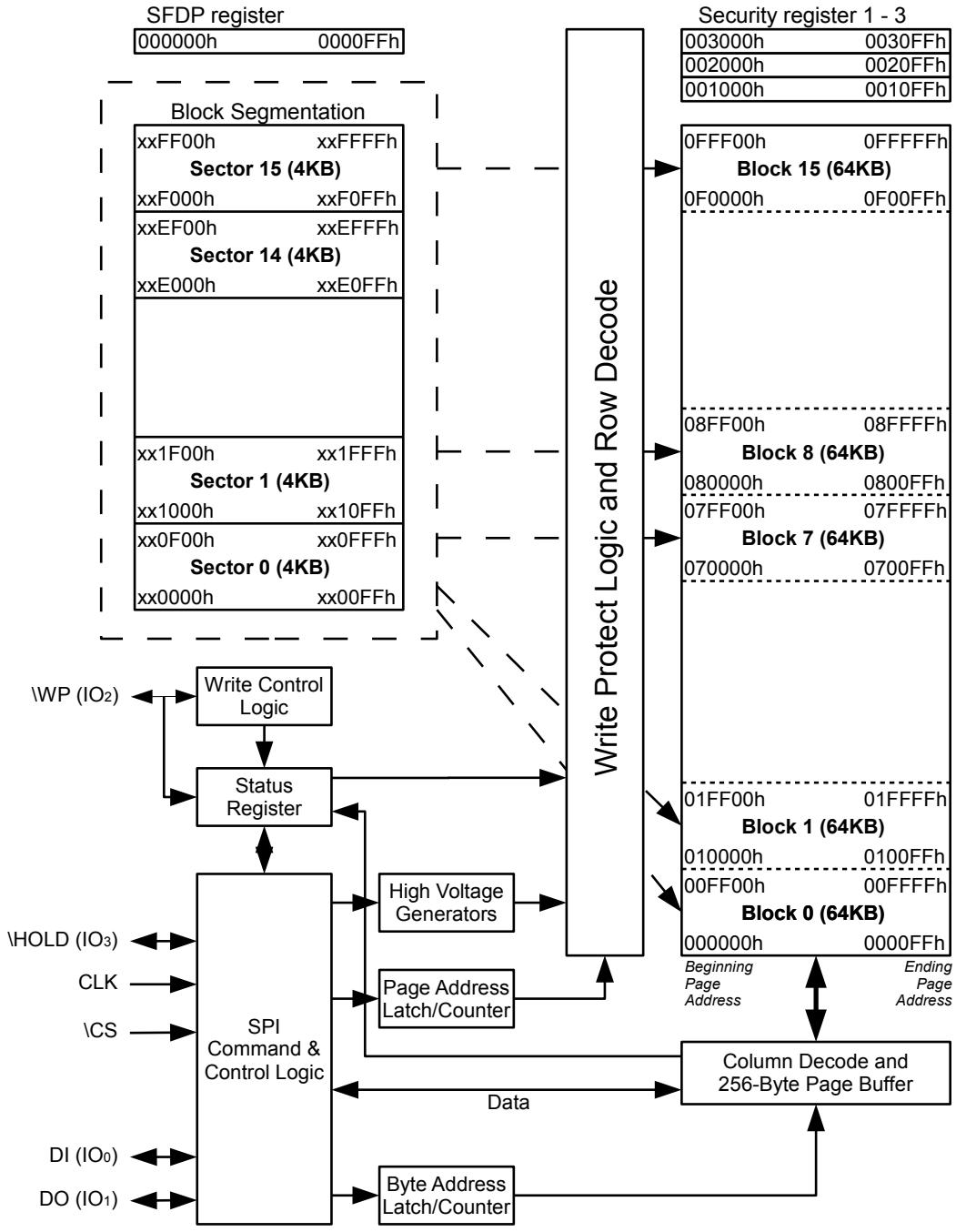


Figura 4.5: Schema a blocchi della memoria Flash W25Q80BV

sotto-gruppi detti settori (ciascuno di essi avente capacità pari a 4 KB). Un ulteriore sezionamento prevede che all'interno di ogni settore vi siano 16 pagine, ciascuna delle quali raggruppa in sé 256 registri elementari a 8 bit.

Per poter risalire al singolo byte viene utilizzato un indirizzo a 24 bit. I 4 bit più significativi devono essere posti sempre uguali a zero, mentre i 4 successivi identificano uno specifico blocco (ovviamente il valore binario `0000` identifica il blocco zero, il numero `0001` corrisponde al blocco 1, e così via). Seguono altri 4 bit per designare i diversi settori all'interno di un blocco, e 4 che permettono di identificare una pagina. Infine, le ultime 8 cifre binarie permettono di selezionare un byte specifico tra i 256 conenuti nelle singole pagine.

Questa struttura permette di segmentare in maniera agevole un indirizzo in 3 sezioni da un byte, ciascuna delle quali potrà essere inviata separatamente al dispositivo tramite la linea MOSI. La logica di controllo del chip si occuperà di ricompattare l'indirizzo in modo da poter selezionare le locazioni di memoria desiderate.

Tra le pagine presenti, ve ne sono tre speciali, denominate *Security Registers*, i cui byte hanno indirizzi del tipo `0x0010nn`, `0x0020nn` e `0x0030nn`. Vi è la possibilità di impedire la modifica di tali porzioni di memoria agendo su ben determinati bit del registro di stato 2.

Volendo adesso passare allo studio dei registri di stato, si osservi la figura 4.6: in essa sono rappresentati i due registri a 8 bit per la gestione della memoria. Alcuni dei bit ivi contenuti hanno riportata la dicitura "non-volatile": ciò sta a indicare che tali valori restano memorizzati anche dopo lo spegimento del dispositivo.

La funzione associata ad ogni bit dei registri di stato è riportata nella tabella 4.3.

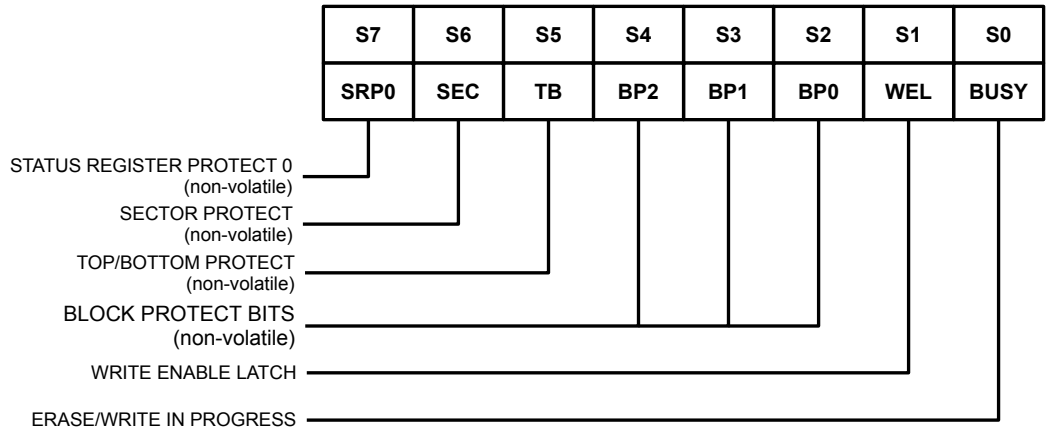
Per concludere la presentazione del dispositivo, si vuole spiegare il meccanismo con cui le diverse istruzioni vengono inviate ed elaborate dal chip, per poi presentare le operazioni più comunemente utilizzate.

Le istruzioni vengono eseguite innanzitutto selezionando la periferica tramite il terminale CS, e quindi inviando il codice dell'istruzione da eseguire (cfr. tabella 4.4); si tenga presente che il primo dato ricevuto dal dispositivo non dovrà mai essere considerato, in quanto all'inizio il terminale DO viene mantenuto in stato di alta impedenza.

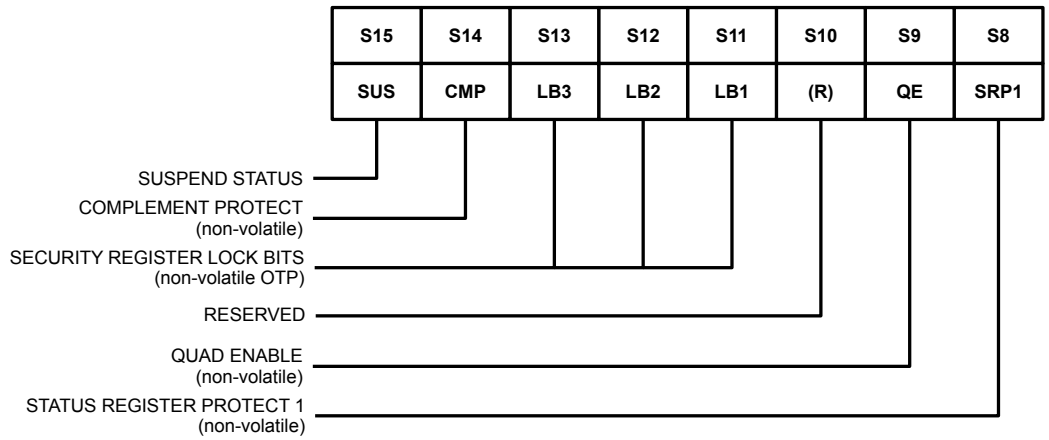
Dopo aver comunicazto l'operazione da esegure, si devono inviare/leggere i dati necessari per completare l'istruzione. Queste, e in generale tutte le informazioni scambiate, devono essere passate a partire dal bit più significativo.

Le due istruzioni "Read Status Register 1" e "Read Status Register 2" permettono di ottenere le impostazioni correnti per il dispositivo. Per completare la lettura, dopo aver abilitato il dispositivo portando a zero CS è necessario inviare il codice dell'istruzione (`0x05` nel primo caso, `0x35` nel secondo), ignorando il dato che verrà ricevuto in contemporanea in quanto non significativo. A questo punto, il dispositivo provvede a inviare continuamente in uscita un byte che rappresenta il contenuto del registro richiesto (ignorando invece ciò che il master invia) fino a quando il chip non viene deselezionato.

Per poter scrivere nei registri di stato si utilizza invece l'istruzione "Write Status



(a) Registro di Stato 1



(b) Registro di Stato 2

Figura 4.6: Registri di Stato della memoria Flash.

Bit	Funzione
BUSY	Bit di sola lettura che assume valore 1 quando è in corso un'operazione di scrittura o di cancellazione dei dati oppure quando vengono scritti i registri di stato.
WEL	Bit di sola lettura che viene settato a 1 quando viene eseguita l'operazione "write enable". Viene resettato automaticamente a 0 dopo l'accensione del dispositivo o in seguito a un'istruzione di scrittura o cancellazione dei dati, quando vengono scritti i registri di stato oppure a causa dell'istruzione "write disable".
BP [0-2]	Questi bit permettono di bloccare la scrittura o la cancellazione di determinate porzioni di memoria (ma non proteggono i registri di stato). Nel manuale tecnico è possibile trovare una tabella che mostra le regioni protette in funzione dei valori di BP [0-2], TB, SEC e CMP.
TB	Viene utilizzato per specificare ulteriori informazioni in merito al comportamento della funzione di protezione da scrittura/cancellazione della memoria.
SEC	Come per TB, viene usato per modificare il comportamento della funzione di protezione.
SRP [0-1]	Questi bit permettono di modificare il comportamento della funzione di protezione dei registri di stato. Nel manuale tecnico è possibile trovare una tabella che mostra il comportamento del dispositivo in funzione dei valori di SRP [0-1] e della tensione presente sul pin WP.
QE	Bit che permette di attivare o disattivare una funzione di comunicazione particolare (detta *Quad SPI Mode*) che utilizza, invece delle sole linee MOSI e MISO, quattro terminali di input/output.
LB [1-3]	Questi bit possono essere programmati una volta sola (con ciò intendendo che, una volta settati, non possono essere più modificati). Ciascuno di essi, una volta portato a 1, blocca la scrittura o la concellazione dei tre *security registers*.
CMP	Come per TB e SEC, viene usato per modificare il comportamento della funzione di protezione.
SUS	Questo bit, quando settato, indica che un'operazione di scrittura o cancellazione della memoria è stata temporaneamente sospesa. Viene resettato in seguito all'accensione, oppure dopo un'istruzione "erase/program resume".

Tabella 4.3: Funzioni dei bit dei Registri di Stato

Istruzione	Codice (hex)	Descrizione
Write Enable	`0x06`	Permette di settare il bit WEL nel registro di stato 1.
Write Disable	`0x04`	Annulla l'effetto dell'istruzione "Write Enable".
Read Status Register 1	`0x05`	Permette di leggere il registro di stato 1.
Read Status Register 2	`0x35`	Permette di leggere il registro di stato 2.
Write Status Register	`0x01`	Permette di scrivere i due registri di stato.
Read Data	`0x03`	Permette di leggere il contenuto della memoria a partire da un byte specificato.
Page Program	`0x02`	Permette di salvare 256 byte all'interno di una pagina.
Sector Erase (4KB)	`0x20`	Elimina i dati contenuti nel settore indicato.
Block Erase (64KB)	`0xD8`	Elimina i dati contenuti nel blocco indicato.
Chip Erase (1MB)	`0xC7`	Elimina i dati contenuti nell'intero chip di memoria.

Tabella 4.4: Elenco delle istruzioni di uso più comune (W25Q80BV). Il corrispondente byte da inviare al dispositivo è riportato in formato esadecimale.

Register". Questa richiede che sia stato preventivamente settato il bit WEL mediante "Write Enable". L'istruzione di scrittura inizia shiftando il relativo codice attraverso la linea MOSI, e dunque inviando in sequenza due byte, uno per ogni registro da scrivere (e iniziando dal registro 1). Eventuali dati che dovessero essere passati successivamente alla periferica verrebbero ignorati. Il processo viene completato deselezionando il dispositivo. In questo caso, nessun dato viene inviato al master.

"Read Data" permette di leggere i dati immagazzinati nel chip. Inizialmente devono essere passati all'integrato 4 byte (mentre questo non restituirà alcuna informazione): l'istruzione stessa più 24 bit di indirizzo che identifichino un byte specifico. A partire dal quinto dato, il dispositivo ignora le informazioni inviate dal master, e provvede invece a fornire uno dopo l'altro tutti i dati contenuti nella pagina indicata. Dopo l'invio di un byte, l'indirizzo viene automaticamente incrementato, rendendo disponibile il dato successivo (questo vuole dire che con questa istruzione si può leggere anche l'intera memoria, a partire da un byte specificato, semplicemente mantenendo CS basso e continuando a inviare il clock). La lettura dei dati termina una volta che si riporta CS a V_{DD}.

Per quanto riguarda la scrittura dei dati, l'istruzione da utilizzare è "Page Program", che permette di inserire i dati in un'intera pagina (che, si ricorda, contiene 256B). Come nel caso di "Write Status Register", è necessario che sia WEL=1. Dopo aver comunicato il comando al dispositivo, si invia l'indirizzo (24 bit) della pagina in cui si vogliono caricare i dati. Si eseguono quindi tanti cicli di invio dati, quanti sono i byte che si vogliono scrivere. Se il numero di valori inviati eccede i 256 (massima capacità della singola pagina) i primi dati inviati vengono progressivamente cancellati (in modo da non "sconfinare" in una pagina adiacente). Riportando CS a V_{DD}, si conclude la procedura.

Una cosa estremamente importante da tenere a mente, è che una pagina può essere scritta solamente dopo essere stata cancellata tramite un'istruzione del tipo "Sector Erase", "Block Erase" o "Chip Erase". In caso ciò non venisse fatto, la programmazione della pagina fallirebbe. I comandi di cancellazione si possono avviare semplicemente trasmettendone il codice, senza la necessità di indicare alcun ulteriore parametro.

Un ultimo aspetto importantissimo riguarda il tempo di esecuzione delle istruzioni di scrittura e cancellazione: queste operazioni possono richiedere tempi molto lunghi, se paragonati al periodo del clock di comunicazione. Per utilizzare correttamente il dispositivo, è necessario attendere ogni volta che la precedente istruzione sia stata completata. Ciò può essere fatto aspettando fino a che il bit BUSY nel registro di stato 1 non risulti pari a zero.

4.3.2 Libreria C per la Gestione della Memoria Flash

In quest'ultimo paragrafo viene riportato e commentato il codice che permette alla Freedom Board di comunicare in maniera corretta con la memoria Flash. Poiché il dispositivo comunica tramite protocollo SPI, viene fatto uso delle funzioni definite precedentemente nei file `spi.h` e `spi.c`.

Per comodità, si è creato il file denominato `W25Q80BV_INSTRUCTION_SET.h`, in cui sono

riportate diverse macro. La prima parte del file è mostrata nel listato 4.7, in cui sono contenute le macro che definiscono i codici delle diverse istruzioni interpretabili dal chip. Seguono nel file diverse righe di codice che facilitano l'estrazione dei singoli bit dei registri di stato mediante delle maschere binarie, e, proseguendo, sono dichiarate delle macro utili per la gestione degli indirizzi di memoria. Tali parti verranno omesse, in quanto non sono state usate per l'implementazione del codice che verrà mostrato (sono state pensate per facilitare la programmazione a chi dovesse utilizzare la libreria). Il file termina con il codice 4.8, che ridefinisce alcuni comandi del primo listato tramite nomi brevi e facili da ricordare (tali abbreviazioni sono state usate nel file C sorgente, e si è ritenuto opportuno riportarle qui).

Si procede dunque presentando l'header del progetto, mostrato in 4.9, in cui sono visibili i prototipi delle funzioni create.

Come si vede, all'inizio del file viene definita la macro `W25Q_SLAVE` in modo da indicare che la memoria verrà trattata come il quarto dispositivo slave della libreria SPI.

La funzione `setupW25Q` (cfr. listato 4.10) inizializza come output i pin della Freedom Board che andranno a collegarsi con i terminali HOLD e WP della memoria, e porta le relative tensioni a V_{DD} (disattivando le due funzioni). In seguito, inizializza la comunicazione SPI richiamando la funzione `setupSPI`.

Seguono le due funzioni `winb_writeEnable` e `winb_writeDisable` (listato 4.11). Le due inviano tramite il protocollo SPI rispettivamente le istruzioni "Write Enable" e "Write Disable".

I due metodi `winb_readRegister1` e `winb_readRegister2` permettono invece di accedere ai contenuti dei registri di stato 1 e 2. Come si può notare, per eseguire i due comandi si utilizza la funzione `sendSPIbuffer`, in cui il buffer contiene l'istruzione di lettura e un byte vuoto. Dopo la comunicazione, al secondo posto dell'array vi sarà un byte che incapsula i bit di stato.

Al fine di soddisfare le specifiche temporali descritte nel manuale tecnico, sono state definite due funzioni di utilità: `winb_busy` legge il contenuto del primo registro di stato e ne isola il bit BUSY[1], restituendo 1 se il dispositivo è occupato, e 0 altrimenti. Il metodo `winb_wait_if_busy` utilizza il precedente codice per mettere in pausa l'esecuzione fintanto che la memoria non completa l'operazione in corso. Le funzioni sono riportate nel listato 4.13.

Le tre funzioni `winb_sectorErase`, `winb_blockErase` e `winb_chipErase` permettono di cancellare dati pre-esistenti in una specifica porzione di memoria. Come si vede nel listato 4.14, dell'indirizzo passato vengono estratti solamente i bit che realmente compongono l'indirizzo desiderato, scartando eventuali bit che non avrebbero senso per tali operazioni (ad esempio, se si deve eliminare un blocco intero, specificare il settore e la pagina al chip sarebbe inutile). Tale meccanismo permette però di compiere un'operazione interessante dal punto di vista del programmatore: se si indica uno specifico byte all'interno della memoria, l'effetto finale è di eliminare l'intera porzione di memoria che lo contiene, sia esso un settore (come avverrebbe

[1] Per farlo, viene usata una di quelle maschere di cui non si è riportata la definizione. Per amore di completezza, si riporta qui la definizione di tale maschera: `#define WINB_S1_BUSY_MASK 0x01`

Listato 4.7: Macro per le istruzioni supportate dal W25Q80BV

```
// ERASE & PROGRAM INSTRUCTIONS
#define WINB_WRITE_ENABLE 0x06
#define WINB_WRITE_ENABLE_FOR_VOLATILE_REGISTERS 0X50
#define WINB_WRITE_DISABLE 0x04
#define WINB_READ_STATUS_REGISTER_1 0X05
#define WINB_READ_STATUS_REGISTER_2 0X35
#define WINB_WRITE_STATUS_REGISTER 0X01
#define WINB_PAGE_PROGRAM 0x02
#define WINB_QUAD_PAGE_PROGRAM 0x32
#define WINB_SECTOR_ERASE_4KB 0x20
#define WINB_BLOCK_ERASE_32KB 0x52
#define WINB_BLOCK_ERASE_64KB 0xD8
#define WINB_CHIP_ERASE 0xC7
#define WINB_ERASE_PROGRAM_SUSPEND 0x75
#define WINB_ERASE_PROGRAM_RESUME 0x7A
#define WINB_POWER_DOWN 0xB9
#define WINB_CONTINUOUS_READ_MODE_RESET 0xFF
// READ INSTRUCTIONS
#define WINB_READ_DATA 0x03
#define WINB_FAST_READ 0x0B
#define WINB_FAST_READ_DUAL_OUTPUT 0x3B
#define WINB_FAST_READ_QUAD_OUTPUT 0x6B
#define WINB_FAST_READ_DUAL_IO 0xBB
#define WINB_FAST_READ_QUAD_IO 0xEB
#define WINB_WORD_READ_QUAD_IO 0xE7
#define WINB_OCTAL_WORD_READ_QUAD_IO 0xE3
#define WINB_SET_BURST_WITH_WRAP 0x77
// ID & SECURITY INSTRUCTIONS
#define WINB_RELEASE_POWER_DOWN_DEVICE_ID 0xAB
#define WINB_MANUFACTURER_DEVICE_ID 0x90
#define WINB_MANUFACTURER_DEVICE_ID_BY_DUAL_IO 0x92
#define WINB_MANUFACTURER_DEVICE_ID_BY_QUAD_IO 0x94
#define WINB_JEDEC_ID 0x9F
#define WINB_READ_UNIQUE_ID 0x4B
#define WINB_READ_SFDP_REGISTER 0x5A
#define WINB_ERASE_SECURITY_REGISTERS 0x44
#define WINB_PROGRAM_SECURITY_REGISTERS 0x42
#define WINB_READ_SECUTIRY_REGISTERS 0x48
}
```

Listato 4.8: Ridefinizione delle istruzioni del chip W25Q80BV

```
// SHORT-NAMED INSTRUCTIONS FOR WINBOND W25Q80BV
#define WINB_WEN WINB_WRITE_ENABLE
#define WINB_WDIS WINB_WRITE_DISABLE
#define WINB_RSR1 WINB_READ_STATUS_REGISTER_1
#define WINB_RSR2 WINB_READ_STATUS_REGISTER_2
#define WINB_WSR WINB_WRITE_STATUS_REGISTER
#define WINB_PP WINB_PAGE_PROGRAM
#define WINB_SECER  WINB_SECTOR_ERASE_4KB
#define WINB_BLER WINB_BLOCK_ERASE_64KB
#define WINB_CHER WINB_CHIP_ERASE
#define WINB_SLEEP WINB_POWER_DOWN
#define WINB_WAKEUP WINB_RELEASE_POWER_DOWN_DEVICE_ID
#define WINB_READ WINB_READ_DATA
```

Listato 4.9: File `W25Q80BV.h`

```
#ifndef W25Q80BV_H_
#define W25Q80BV_H_

#define W25Q_SLAVE SPI_SLAVE4

void setupW25Q();

void winb_writeEnable();
void winb_writeDisable();

unsigned char winb_readRegister1();
unsigned char winb_readRegister2();

unsigned char winb_busy();
void winb_wait_if_busy();

void winb_sectorErase(unsigned int address);
void winb_blockErase(unsigned int address);
void winb_chipErase();

void winb_writePage(unsigned char *data, unsigned int address);
unsigned char winb_readByte(unsigned int address);

#endif /* W25Q80BV_H_ */
```

Listato 4.10: Funzione `setupW25Q`

```
// definizioni ausiliarie per i pin HOLD e WRITE_PROTECT
#define HOLD_PIN_HIGH GPIOC_PSOR |= 1<<12
#define HOLD_PIN_LOW GPIOC_PCOR |= 1<<12
#define WRITE_PROTECT_PIN_HIGH GPIOC_PSOR |= 1<<16
#define WRITE_PROTECT_PIN_LOW GPIOC_PCOR |= 1<<16

void setupW25Q() {
  // clock alle porte C
  SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK;

  // Set del pin di "HOLD-ON"
  PORTC_PCR12 = (PORTC_PCR12 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
  GPIOC_PDDR |= 1<<12;
  HOLD_PIN_HIGH;

  // Set del pin di "WRITE-PROTECT"
  PORTC_PCR16 = (PORTC_PCR16 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
  GPIOC_PDDR |= 1<<16;
  WRITE_PROTECT_PIN_HIGH;

  // abilita il funzionamento della comunicazione SPI
  setupSPI();
}
```

Listato 4.11: Metodi `winb_writeEnable` e `winb_writeDisable`

```
void winb_writeEnable() {
  sendSPIdata(WINB_WEN, W25Q_SLAVE);
}

void winb_writeDisable() {
  sendSPIdata(WINB_WDIS, W25Q_SLAVE);
}
```

Listato 4.12: Metodi `winb_readRegister1` e `winb_readRegister2`

```
unsigned char winb_readRegister1() {
  unsigned char buff[] = { WINB_RSR1, 0x00 };
  sendSPIbuffer(buff, 2, W25Q_SLAVE);
  return buff[1];
}

unsigned char winb_readRegister2() {
  unsigned char buff[] = { WINB_RSR2, 0x00 };
  sendSPIbuffer(buff, 2, W25Q_SLAVE);
  return buff[1];
}
```

Listato 4.13: Metodi winb_busy e winb_wait_if_busy

```
1 unsigned char winb_busy() {
2   unsigned char s1 = winb_readRegister1();
3   return (s1 & WINB_S1_BUSY_MASK);
4 }
5
6 void winb_wait_if_busy() {
7   while(winb_busy() == 1);
8 }
```

con la prima funzione) o un blocco (come succederebbe nella seconda).

Listato 4.14: Funzioni di cancellazione della memoria

```
1  void winb_sectorErase(unsigned int address) {
2    winb_wait_if_busy();
3    winb_writeEnable();
4    unsigned char sector = (unsigned char) ((address & 0xF000)>>12); // settore
5    unsigned char block = (unsigned char) ((address & 0xF0000)>>16); // blocco
6    unsigned char buf[] = { WINB_SECER, block, (sector<<4), 0x00 };
7    sendSPIbuffer(buf, 4, W25Q_SLAVE);
8  }
9
10 // permette di cancellare un determinato blocco (i quattro bit meno
       significativi dell'indirizzo sono ignorati)
11 void winb_blockErase(unsigned int address) {
12   winb_wait_if_busy();
13   winb_writeEnable();
14   unsigned char block = (unsigned char) ((address & 0xF0000)>>16); // blocco
15   unsigned char buf[] = { WINB_SECER, block, 0x00, 0x00 };
16   sendSPIbuffer(buf, 4, W25Q_SLAVE);
17 }
18
19 // permette di cancellare l'intera memoria
20 void winb_chipErase() {
21   winb_wait_if_busy();
22   winb_writeEnable();
23   sendSPIdata(WINB_CHER, W25Q_SLAVE);
24 }
```

Concludono il capitolo le due funzioni di scrittura e lettura: `winb_writePage` e `winb_readByte`.

La prima è stata progettata in modo da permettere di programmare esattamente una pagina di memoria. Come si vede nel relativo codice (cfr. 4.15), i valori da salvare vengono passati mediante un puntatore. Appena inizia l'esecuzione, viene chiamata `winb_wait_if_busy` al fine di garantire che tutte le operazioni precedenti siano terminate. Successivamente, si abilita la scrittura con l'istruzione `winb_writeEnable` e si isolano i diversi frammenti dell'indirizzo. Come si vede, se si sta tentando di scrivere sulla prima pagina l'esecuzione termina, in quanto essa è accessibile in sola

lettura (contiene diverse informazioni pre-salvate dal produttore come ad esempio il numero di serie).

Se il controllo viene invece superato, viene creato un buffer di dimensione pari a 260, e cioè 4+256: i primi quattro byte corrispondono rispettivamente all'istruzione di scrittura e ai byte di indirizzo. Gli altri 256 sono invece riempiti copiando l'array `data` passato come parametro. L'intero buffer viene quindi trasmesso al dispositivo.

La seconda funzione si occupa anch'essa di aspettare che il chip sia libero, isolando in seguito le singole componenti dell'indirizzo. A differenza del metodo precedente, però, invia un buffer composto di soli 5 elementi: l'istruzione (8 bit), l'indirizzo (24 bit) e un byte vuoto. Dopo la chiamata a `sendSPIbuffer` il byte vuoto sarà invece stato riempito con il valore da leggere, che può dunque essere ritornato.

Listato 4.15: Metodo `winb_writePage`

```
1  void winb_writePage(unsigned char *data, unsigned int address) {
2    // aspetta che la memoria sia pronta per ricevere istruzioni
3    winb_wait_if_busy();
4    // setta il bit "Write Enable"
5    winb_writeEnable();
6    // isola le singole parti dell'indirizzo
7    unsigned char page = (unsigned char) ((address & 0xF00)>>8); // pagina
8    unsigned char sector = (unsigned char) ((address & 0xF000)>>12); // settore
9    unsigned char block = (unsigned char) ((address & 0xF0000)>>16); // blocco
10   // controlla che non si stia cercando di scrivere sulla prima pagina, e in
         tal caso interrompe la scrittura
11   // nota: se l'indirizzo va oltre il massimo consentito, la scrittura si
         interrompe comunque!
12   if(page==0 && sector==0 && block==0) {
13    winb_writeDisable();
14    return;
15   }
16   unsigned char buf[260];
17   buf[0] = WINB_PP;
18   buf[1] = block;
19   buf[2] = (sector<<4 & page);
20   buf[3] = 0x00;
21   int i=0;
22   for(i=4;i<260;i++)
23     buf[i] = data[i-4];
24     sendSPIbuffer(buf, 260, W25Q_SLAVE);
25 }
```

Listato 4.16: Metodo winb_readByte

```
unsigned char winb_readByte(unsigned int address) {
  winb_wait_if_busy();

  // isola le singole parti dell'indirizzo
  unsigned char byte_to_read = (unsigned char) (address & 0xFF);
  unsigned char page = (unsigned char) ((address & 0xF00)>>8); // pagina
  unsigned char sector = (unsigned char) ((address & 0xF000)>>12); // settore
  unsigned char block = (unsigned char) ((address & 0xF0000)>>16); // blocco

  unsigned char buf[5];
  buf[0] = WINB_READ;
  buf[1] = block;
  buf[2] = (sector<<4 & page);
  buf[3] = byte_to_read;
  buf[4] = 0x00;

  sendSPIbuffer(buf, 5, W25Q_SLAVE);
  return buf[4];
}
```

Capitolo 5

Display LCD e Scheda di Espansione

Durante la scrittura di un codice C, un passo fondamentale cosiste nell'attività di debug: gli errori di sintassi vengono individuati in fase di compilazione, mentre quelli logici vanno trovati mandando in esecuzione il programma, e verificando che ogni istruzione compia effettivamente il compito assegnato. Nelle applicazioni desktop, solitamente si testano le diverse funzioni implementate utilizzando il terminale di comando: si esegue ogni metodo con un set di parametri in ingresso noti, quindi si stampano i risultati delle diverse operazioni sullo schermo, confrontandoli con quelli attesi. Incongruenze tra i valori stampati da una funzione e quelli attesi permettono di individuare un errore nel codice.

A titolo di esempio, si consideri la seguente funzione definita in codice C, che calcola la somma dei primi N numeri naturali:

```
1.    unsigned char sommaInteri(unsigned char N) {
2.      unsigned char s=0;
3.      unsiged char i;
4.      for(i=0; i<=N; i++) {
5.        s+=i;
6.      }
7.      return s;
8.    }
```

Tale funzione non contiene errori di sintassi; al contrario è sicuramente presente un errore logico: alla riga 2 la variabile `s` è dichiarata come `unsigned char`, e questo significa che il massimo valore che può contenere è pari a $2^8 - 1 = 255$ (in quanto normalmente le variabili `unsigned char` sono a 8bit). Questo comporta un problema di troncamento/overflow, che in C solitamente non viene segnalato.

Tenendo conto che la somma dei primi N numeri naturali può essere calcolata mediante:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

si immagini di chiamare la funzione con valore in ingresso pari a 200. Il risultato sarebbe uguale a 20100, che è troppo grande per essere memorizzato all'interno di una variabile di tipo `unsigned char`. La variabile `s` andrebbe dunque in overflow ripetutamente durante i calcoli, e il risultato ottenuto - a meno di fortuite coincidenze - sarebbe diverso da quello previsto.

Per risolvere il problema, basterebbe modificare la funzione dichiarando `s` di tipo `unsigned int` e modificando di conseguenza il valore di uscita della funzione.

Un errore di questo tipo è semplice da individuare tramite il terminale di comando; la Freedom Board non dispone però di un simile dispositivo di output, e rende più ardua l'individuazione di errori logici. Si è quindi pensato di aggiungere un secondo dispositivo esterno oltre alla memoria Flash, e cioè un display LCD (*Liquid Crystal Display*).

Con l'inserimento di componenti esterni, si è però incontrata una problematica non da poco: per il controllo dei motori, del servo e delle telecamere, si è utilizzata una scheda di espansione che viene inserita sopra alla Freedom Board. In questo modo però vengono occupati tutti i pin di uscita del microcontrollore, rendendo impossibile il collegamento di harware aggiuntivo, nonstante la scheda di espansione non utilizzi effettivamente tutti i terminali della Freedom Board.

Per queste ragioni, si è progettata un'ulteriore scheda, da interporre tra il microcontrollore e la piattaforma aggiuntiva, che permette l'aggiunta sia della memoria Flash e del display, che di eventuali dispositivi aggiuntivi.

5.1 Display LCD WH1602B

Il display utilizzato è siglato *WH1602B*, prodotto dalla Winstar, e consiste in una matrice da 32 caratteri, disposti su due righe. Il display viene alimentato a $5\,V$[1], e viene comandato tramite il driver *HD44780U*.

Uno dei principali motivi che ha spinto ad utilizzare questo dispositivo è stato il costo estremamente ridotto (meno di 15€), e inoltre il fatto che esso si basa sul popolare driver prodotto da *Hitachi*, già integrato nel modulo. Questo è infatti alla base di un grandissimo numero di dispositivi LCD, anche con caratteristiche diverse tra loro (come ad esempio numero di righe e colonne), e permette quindi di scrivere un'unica libreria di codice che potrà essere facilmente adattata alle caratteristiche di ogni diverso modello.

[1] Nel manuale tecnico viene riportato che alcuni modelli possono essere anche alimentati a $3\,V$. Si consulti il datasheet [17] per ulteriori informazioni.

Figura 5.1: Display LCD WH1602B

5.1.1 Driver HD44780U

Il driver in oggetto ha come principale vantaggio la semplicità di utilizzo e al contempo la grande flessibilità di funzionamento.

Permette di controllare display in cui ogni singolo carattere può essere una matrice di punti 5x8 oppure 5x10, gestendo in output fino a 80 caratteri (compatibilmente con le capacità del display associato). Internamente è dotato di una ROM in cui sono memorizzati alcuni caratteri pronti all'uso, e dispone di una RAM in cui possono essere salvati ulteriori caratteri definiti dall'utente (detta CGRAM). Infine, un'ulteriore memoria RAM, detta DDRAM, memorizza i caratteri che attualmente devono essere visualizzati sul display.

Le funzioni del driver non si concludono qui, tuttavia all'interno del presente progetto sono state utilizzate solo quelle di base del display: si rimanda a [13] per approfondimenti.

La semplicità del driver consiste anche nel cablaggio ridotto richiesto per comandare il display tramite il driver: sono necessari 3 pin per la selezione del comportamento, più 4 o 8 linee dati (il numero dipende da una scelta dell'utilizzatore).

A questi pin vanno comunque aggiunti quelli di alimentazione (condivisi con il display vero e proprio), oltre che un pin ulteriore necessario per la regolazione del contrasto. L'elenco dei pin (numerati in base al loro posto sul modulo LCD) è visibile in tabella 5.1.

Il driver utilizza tre registri per il salvataggio di diversi valori. I primi due registri prendono il nome di *Instruction Registers*, e hanno come funzione quella di memorizzare alcuni parametri di funzionamento. Il terzo viene chiamato invece *Data Register*, e immagazzina i caratteri da visualizzare sullo schermo. Il segnale di controllo RS (Register Select) permette di scegliere su quale registro scrivere tra i due presenti.

Il segnale $R/\overline{W}$ ha come scopo quello di impostare le linee di comunicazione rispettivamente come input o come output. Nella realizzazione della libreria di interfaccia, si è deciso di non leggere dati dal modulo LCD (ovviamente questo non ne preclude l'uso come dispositivo di output) e pertanto il segnale deve rimanere sempre a 0 (scrittura).

Il pin "E" corrisponde alla funzione "Enable", che indica al driver di leggere i dati

Pin	Num.	Funzione
V_{ss}	1	Messa a terra.
V_{dd}	2	Alimentazione.
V_0	3	Regolazione del contrasto.
RS	4	Segnale "Register Select".
$R/\overline{W}$	5	Segnale "Read/Write".
E	6	Segnale "Enable".
DB0 - DB7	7-14	Linee dati.
A	15	Anodo del LED di retro-illuminazione.
K	16	Catodo del LED di retro-illuminazione.

Tabella 5.1: Pin del display LCD

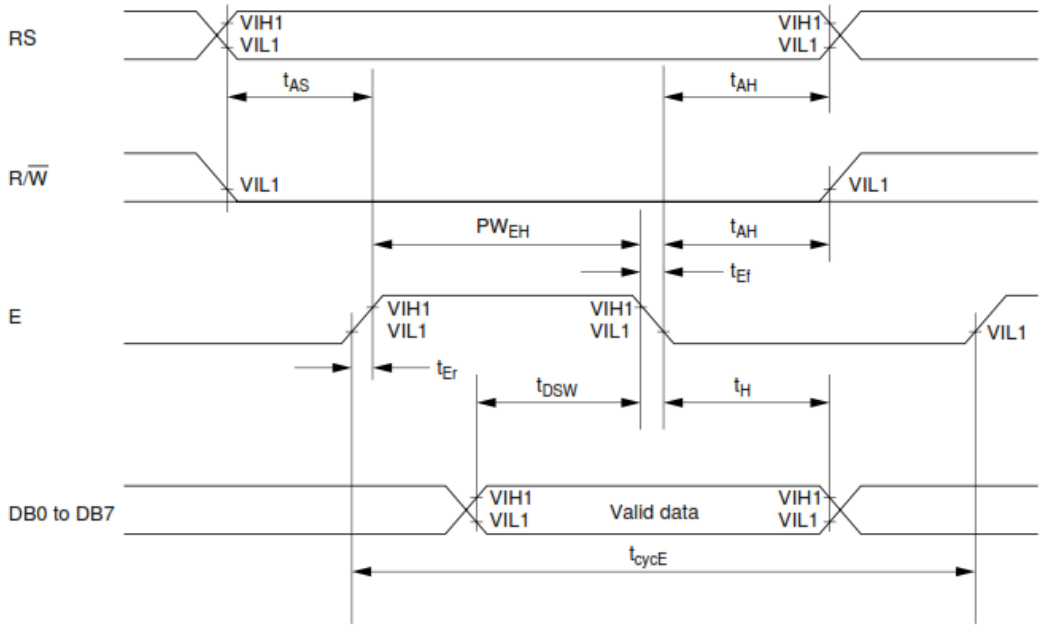
dalle linee di comunicazione.

La comunicazione dei dati avviene tramite istruzioni a 8 bit. Le informazioni possono essere inivate in parallelo sugli otto terminali DB0-DB7, oppure inviate alle sole quattro linee DB4-DB7 (in tal caso i dati vengono inviati in due gruppi da 4 bit).

È fondamentale, per il corretto funzionamento del dispositivo, rispettare ben determinate temporizzazioni nel pilotaggio dei diversi segnali. La figura 5.2 mostra i diversi segnali del modulo LCD e le diverse temporizzazioni da rispettare.

Per completare un ciclo di scrittura dei dati all'interno dei registri, bisogna seguire i seguenti passi:

- Caso a 8 linee:
  1. si porta $R/\overline{W}$ a 0;
  2. si impostano gli otto segnali DB0-DB7 in maniera appropriata inserendo il codice di un carattere o di un'istruzione (cfr. tabella 5.2 e figura 5.3);
  3. si seleziona il registro dati o quello delle impostazioni tramite RS;
  4. si porta in alto il terminale E;
  5. si riporta in basso il terminale E.
- Caso a 4 linee:
  1. si porta $R/\overline{W}$ a 0;
  2. si seleziona il registro dati o quello delle impostazioni tramite RS;
  3. si impostano i segnali DB4-DB7 immettendo i 4 bit più significativi del dato da inviare;
  4. si porta in alto il terminale E;
  5. si riporta in basso il terminale E;



Nome	*Simbolo*	*min* [*ns*]	*max* [*ns*]
enable cycle time	t_{cycE}	1000	—
enable pulse width (high level)	PW_{EH}	450	—
enable rise/fall time	t_{Er},t_{Ef}	—	450
address setup time	t_{AS}	60	—
address hold time	t_{AH}	20	—
data setup time	t_{DSW}	195	—
data hold time	t_H	10	—

Figura 5.2: Temporizzazione dei segnali del display; in legenda si leggono i valori (espressi in ns) dei diversi tempi in gioco.

6. si impostano i segnali DB4-DB7 immettendo i 4 bit meno significativi del dato da inviare;
7. si porta in alto il terminale E;
8. si riporta in basso il terminale E;

Le istruzioni che il driver può riconoscere sono riportate in tabella 5.2, mentre la figura 5.3 mostra i codici dei caratteri che il display può mostrare. Nella pratica, il codice di un carattere corrisponde al relativo codice della codifica ASCII.

Figura 5.3: Codici dei caratteri del display

5.1.2 Libreria per il Controllo del Display

Il codice per il controllo del display si occupa di gestire i diversi segnali entranti nel driver. Il file header `liquid_crystal.h` (listato 5.1) mostra l'interfaccia pubblica contenente i prototipi delle funzioni implementate.

Istruzione	**RS**	**Codice**	**Descrizione**	T_{max}
NOP	0	`00000000`	Nessuna operazione.	$0\,s$
Cancella display	0	`00000001`	Cancella tutti i dati visualizzati e riporta il cursore alla posizione zero.	$1.64\,ms$
Cursore a capo	0	`0000001x`	Riporta il cursore alla posizione zero.	$1.64\,ms$
Modo d'accesso dei caratteri	0	`000001IS`	Imposta la direzione del movimento del cursore (`I=0`: decrementa la posizione del cursore; `I=1`: decrementa la posizione del cursore). Inoltre specifica se a muoversi deve essere il cursore o il flusso di caratteri già stampati (`S=0`: spostamento del display; `S=1`: spostamento del cursore).	$40\,\mu s$
Controllo display	0	`00001DCB`	Permette di accendere/spegnere il display (`D=0`: display spento; `D=1`: display acceso). Controlla inoltre la presenza del cursore (`C=0`: cursore non visibile; `C=1`: cursore visibile) e il suo lampeggio (`B=0`: lampeggio disattivo; `B=1`: lampeggio attivo).	$40\,\mu s$
Scorrimento cursore e display	0	`0001SDxx`	Sposta il cursore (`S=0`) oppure il display (`S=1`) nella direzione indicata (`D=0`: sposta a sinistra; `D=1`: sposta a destra) senza cambiare il contenuto del registro dati.	$40\,\mu s$
Impostazioni funzioni	0	`001LNFxx`	Imposta il numero di linee da utilizzare (`L=0`: 4 linee; `L=1`: 8 linee), il numero di righe del displayda utilizzare (`N=0`: una riga; `N=1`: due righe) e infine la dimensione dei caratteri (`F=0`: caratteri da 5x7 punti; `F=1`: caratteri da 5x10 punti).	$40\,\mu s$
Indirizzo CGRAM	0	`01QQQQQQ`	Indica che si vuole salvare un nuovo carattere personalizzato. Il valore QQQQQQ indica la locazione di memoria in cui si vuole salvare il nuovo carattere.	$40\,\mu s$
Indirizzo DDRAM	0	`1QQQQQQQ`	Seleziona un determinato indirizzo nella DDRAM. Si tenga presente che gli indirizzi della memoria sono 80, ma spesso i caratteri visualizzabili sono molti meno. Tipicamente, in un display a 16x2 caratteri, i caratteri della prima riga corrispondono agli indirizzi esadecimali da `0h00` a `0h0F`, mentre quelli della seconda hanno indirizzi che vanno da `0h40` a `0h4F`.	$40\,\mu s$
Scrittura	1	`QQQQQQQQ`	Scrive un dato nella cella DDRAM o CGRAM selezionata.	$40\,\mu s$

Tabella 5.2: Istruzioni del driver HD44780U; nei codici delle istruzioni il valore "x" indica che il bit corrispondente può assumere qualunque valore.

Listato 5.1: File header `liquid_crystal.h`

```
#ifndef LIQUID_CRYSTAL_H_
#define LIQUID_CRYSTAL_H_

void setupLCD();
void lcd_clear();
void lcd_home();
void lcd_numLines(unsigned char num);
void lcd_enable(unsigned char on_or_off);
void lcd_blink(unsigned char on_or_off);
void lcd_cursor(unsigned char on_or_off);
unsigned char lcd_printLetter(unsigned char letter);
unsigned char lcd_printNumber(unsigned int num, unsigned char digits);
unsigned char lcd_printHexNumber(unsigned int num, unsigned char digits);
unsigned char lcd_printText(unsigned char *letters, unsigned char len);
unsigned char lcd_setText(unsigned char *letters, unsigned char len);
void lcd_newLine();
void lcd_setCursorPosition(unsigned char pos);
void lcd_setCursorPositionRC(unsigned char row, unsigned char col);
```

Dei metodi riportati nel file header, se ne illustreranno solamente alcuni, in quanto gli altri sono stati realizzati in maniera del tutto simile.

All'inizio del corrispondente file sorgente `liquid_crystal.c` vengono dichiarate alcune macro che hanno lo scopo di facilitare la scrittura del codice. Tali maschere sono riportate nel listato 5.2.

Come si vede, è stata dichiarata una variabile `display_status` che permette di salvare le attuali impostazioni scelte per il display, mentre `lcd_counter` permette di memorizzare la posizione corrente (impostata dall'utente) del cursore.

Il metodo `setupLCD` (listato 5.3) permette di inizializzare la comunicazione con il dispositivo con le seguenti modalità: comunicazione a 4 bit, display su due righe, output visibile, cursore visibile, lampeggio attivo.

È importante notare che, dopo aver impostato le funzioni GPIO correttamente per i diversi pin, si avvia una sorta di procedura di sincronizzazione: questa è necessaria per il funzionamento del modulo, e viene descritta approfonditamente nel datasheet.

La procedura inizia imponendo, dopo l'accesione del driver, un ritardo minimo di 50 *ms*; in realtà questa è solo una precauzione, poiché dopo l'accensione della Freedom Board (e di conseguenza anche del modulo LCD) difficilmente questo setup verrà eseguito subito, in quanto bisogna prima portare a termine i setup di inizializzazione del microcontrollore. Un ritardo di 50 millisecondi è tuttavia contenuto, e, considerato anche che deve essere eseguito una volta sola, non costa nulla agire in sicurezza chiamando la funzione `delay(50)`.

La procedura continua inviando sui quattro terminali il codice esadecimale `0x03`. Inizialmente il modulo si aspetta di comunicare su 8 linee, il che vuol dire che in questo caso l'istruzione viene interpretata in binario come `0b0011nnnn`, e cioè

Listato 5.2: Macro definite nel file `liquid_crystal.c`

```
// definizioni per accendere/spegnere le porte connesse a RS, RW, EN
#define RS_HIGH GPIOA_PSOR |= 1<<12
#define RS_LOW GPIOA_PCOR |= 1<<12
#define RW_HIGH GPIOA_PSOR |= 1<<4
#define RW_LOW GPIOA_PCOR |= 1<<4
#define EN_HIGH GPIOA_PSOR |= 1<<5
#define EN_LOW GPIOA_PCOR |= 1<<5
// definizioni dei comandi accettati dal driver HD44780U e relativi
// tempi di esecuzione (espressi in microsecondi). I tempi sono
// approssimati per eccesso, con un margine di sicurezza.
#define LCD_CMD_CLEAR 0x01
#define LCD_CMD_CLEAR_DELAY 2000
#define LCD_CMD_HOME 0x02
#define LCD_CMD_HOME_DELAY 2000
#define LCD_CMD_CHARACCESS_BASE 0x04
#define LCD_CMD_CHARACCESS_INCREMENT 0x02
#define LCD_CMD_CHARACCESS_SHIFT 0x01
#define LCD_CMD_CHARACCESS_DELAY 50
#define LCD_CMD_CONTR_BASE 0x08
#define LCD_CMD_CONTR_SHOWDISP 0x04
#define LCD_CMD_CONTR_CURSOR 0x02
#define LCD_CMD_CONTR_BLINK 0x01
#define LCD_CMD_CONTR_DELAY 50
#define LCD_CMD_SHIFT_BASE 0x10
#define LCD_CMD_SHIFT_DISP 0x08
#define LCD_CMD_SHIFT_RIGHT 0x04
#define LCD_CMD_SHIFT_DELAY 50
#define LCD_CMD_FUNC_BASE 0x20
#define LCD_CMD_FUNC_DATAMODE 0x10
#define LCD_CMD_FUNC_LINES 0x08
#define LCD_CMD_FUNC_FONT 0x04
#define LCD_CMD_FUNC_DELAY 50
#define LCD_CMD_DDRAM_BASE 0x80
#define LCD_CMD_DDRAM_DELAY 50
// definizioni per la memorizzazione delle impostazioni attuali del
// display. Il bit meno significativo memorizza se attiva
// l'impostazione "display attivo"; il secondo LSB l'impostazione
// "blink"; il terzo la visibilit del cursore; il quarto la
// visualizzazione su una o due righe
#define DISP_ST_EN_MASK 0x01
#define DISP_ST_BL_MASK 0x02
#define DISP_ST_CUR_MASK 0x04
#define DISP_ST_2L_MASK 0x08
unsigned char display_status = 0x00;
unsigned char lcd_counter = 0x00;
```

Listato 5.3: Metodo setupLCD

```
1  void setupLCD() {
2    // clock alle porte A e C
3    SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK | SIM_SCGC5_PORTC_MASK;
4    // PTA4: RW PTA5: EN PTA12: RS
5    PORTA_PCR4 = (PORTA_PCR4 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
6    PORTA_PCR5 = (PORTA_PCR5 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
7    PORTA_PCR12 = (PORTA_PCR12 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
8    // PTC8: D4 PTC9: D5 PTC10: D6 PTC11: D7
9    PORTC_PCR8 = (PORTC_PCR8 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
10   PORTC_PCR9 = (PORTC_PCR9 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
11   PORTC_PCR10 = (PORTC_PCR10 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
12   PORTC_PCR11 = (PORTC_PCR11 & (~(PORT_PCR_MUX_MASK))) | PORT_PCR_MUX(1);
13   GPIOA_PCOR |= 1<<4|1<<5|1<<12; // imposta a LOW RS, RW ed EN
14   GPIOA_PDDR |= 1<<4|1<<5|1<<12; // imposta in output RS, RW ed EN
15   GPIOC_PDDR &= ~(0x0F<<8); // imposta in input i pin D4-D7
16
17   // aspetta 50ms, come richiesto dal datasheet,
18   // affinch il dispositivo possa "svegliarsi"
19   delay(50);
20   /* Appena sveglio, bisogna mandare tre volte il comando 0x03 sui 4
21      "pin pi significativi" del display (DB4-DB7) in modo che questo
22      possa sincronizzarsi. I bit meno significativi sono irrilevanti in
23      questo caso. Tra un comando e l'altro vi sono delle pause, come
24      indicate dal costruttore nel datasheet. */
25   send4bits(0x03);
26   delay(5);
27   send4bits(0x03);
28   delay(5);
29   send4bits(0x03);
30   delay(2);
31   send4bits(0x02); // imposta la modalit di scrittura a 4 bit
32   delayMicroseconds(50); // attende 50us, come indicato nel datasheet
33
34   lcd_enable(1); // abilita il display
35   lcd_blink(1); // abilita il lampeggio
36   lcd_cursor(1); // abilita la visualizzazione del cursore
37   lcd_clear(); // pulisce lo schermo
38   lcd_numLines(2); // abilita la visualizzazione su due righe
39 }
```

l'istruzione "impostazioni funzioni" (cfr. tabella 5.2), in cui i primi tre bit sono rispettivamente 0, 0 e 1; il quarto bit è pari a 1, indicando che la comunicazione deve essere a 8 linee. I successivi quattro bit sono indeterminati, in quanto le linee DB0-DB3 non sono collegate a nulla, tuttavia le impostazioni ad esso associate possono essere aggiustate in una fase successiva. A seguito di tale invio, si aspetta un tempo pari a 5 *ms* (tale tempo è indicato nel datasheet).

Questa fase si ripete per altre due volte (l'ultima volta si possono aspettare solo 2 *ms*). Alla fine, il costruttore garantisce che il dispositivo sia pronto a ricevere le istruzioni in maniera corretta.

Il motivo per cui bisogna indicare per tre volte che la comunicazione avverrà su 8 linee è semplice: il costruttore riporta sul manuale che, appena acceso il driver, la prima istruzione, e talvolta anche la seconda, potrebbero essere "perse". Se come prima istruzione venisse mandata la richiesta di comunicare a 4 bit, non si sarebbe sicuri su cosa inviare alla fase successiva: se l'istruzione fosse stata persa, dovrebe essere mandata nuovamente la richiesta di comunicare a 4 linee, altrimenti bisognerebbe inviare le istruzioni desiderate, facendo attenzione a mandarle in formato corretto (cioè in due blocchi da 4 bit). Al contrario, seguendo la procedura esposta, si è sicuri che entro la terza istruzione il dispositivo sia certamente sincronizzato e possa accettare istruzioni a 8 linee.

È dunque possibile passare alla comunicazione a 4 linee, inviando questa volta il codice `0x02` sulle linee DB4-DB7, e aspettando il tempo necessario a completare l'istruzione (40 μs) chiamando la funzione `delayMicroseconds(50)` definita nel codice di gestione del SysTick (si aspettano 50 μs per ragioni di sicurezza).

Infine, mediante le funzioni definite nel file header (e implementate successivamente nel file sorgente), si impostano i parametri di funzionamento desiderati.

All'interno della funzione di setup viene utilizzata una funzione ausiliaria non riportata nel file header: `send4bits`. Questa è una funzione di basso livello, implementata per poter utilizzare in maniera più semplice il dirver Hitachi. Il codice viene riportato nel listato 5.4.

Listato 5.4: Funzione ausiliaria `send4bits`

```
1 // procedura che permette di inviare, in modalit 4 bit,
2 // mezzo byte al driver
3 void send4bits(unsigned char data) {
4   GPIOC_PDDR |= 0x0F<<8; // setta le porte D4-D7 in OUTPUT
5   GPIOC_PCOR |= 0x0F<<8; // pulisce le porte
6   GPIOC_PSOR |= (data&0x0F)<<8; // imposta le porte per l'output
7   pulseEnable(); // trasferisce i dati
8 }
```

Tale funzione si aspetta di ricevere in ingresso 4 bit, che manda in uscita sui terminali DB4-DB7 tramite funzione GPIO. Dopo aver preparato i dati, invia al driver un impulso sul terminale Enable tramite la funzione `pulseEnable()` (il relativo codice è visibile in 5.5).

Altre due funzioni ausiliarie necessarie per la trasmissione dei dati, sono le funzioni

Listato 5.5: Metodo pulseEnable

```
// abilita la ricezione dei dati da parte del driver (cfr datasheet)
void pulseEnable() {
  EN_LOW; // porta in basso EN
  delayMicroseconds(1); // aspetta 1us
  EN_HIGH; // porta in alto EN
  delayMicroseconds(1); // aspetta 1us
  EN_LOW; // riporta in basso EN
  delayMicroseconds(100); // aspetta 100 us
}
```

sendCommand e sendLetter (listati 5.6 e 5.7) che permettono rispettivamente di inviare un comando (quindi ponendo RS a 0 e inviando il codice relativo tramite due chiamate successive a send4bits) oppure di stampare un carattere (si pone RS a 1, e si invia il codice del carattere da stampare tramite due chiamate a send4bits).

Listato 5.6: Funzione sendCommand

```
// procedura che invia un comando al driver
void sendCommand(unsigned char cmd, unsigned int microseconds) {
  // abbassa i segnali RS e RW, quindi aspetta 1us (cfr datasheet)
  RS_LOW;
  RW_LOW;
  delayMicroseconds(1);
  send4bits(cmd>>4); // invia i 4 bit pi significativi del comando
  send4bits(cmd&0x0F); // invia i 4 bit meno significativi del comando
  delayMicroseconds(microseconds); // aspetta che l'istruzione termini
}
```

Listato 5.7: Funzione sendLetter

```
// procedura che invia una carattere al driver
void sendLetter(unsigned char L) {
  // alza RS e abbassa RW, quindi aspetta 1us (cfr datasheet)
  RS_HIGH;
  RW_LOW;
  delayMicroseconds(1);
  send4bits(L>>4); // invia i 4 bit pi significativi del comando
  send4bits(L&0x0F); // invia i 4 bit meno significativi del comando
  delayMicroseconds(50); // aspetta che l'istruzione termini
}
```

Il resto delle funzioni implementate consiste in pratica nell'invio di un'istruzione tramite una chiamata a sendCommand, oppure, nel caso di scrittura sul display, a una o più chiamate del metodo sendLetter. A titolo di esempio, sono riportati i codici delle due funzioni lcd_clear (5.8) e lcd_printLetter (5.9). Inoltre, si mostra l'implementazione della funzione lcd_printText (5.10) che permette di mandare a schermo una sequenza di caratteri.

Listato 5.8: Metodo lcd_clear

```
// pulisce lo schermo del display
void lcd_clear() {
  // invia il comando "CLEAR"
  sendCommand(LCD_CMD_CLEAR, LCD_CMD_CLEAR_DELAY);
  // resetta il contatore del display
  lcd_counter = 0;
}
```

Listato 5.9: Funzione lcd_printLetter

```
// manda a schermo una lettera
unsigned char lcd_printLetter(unsigned char letter) {
  // verifica che vi sia posto per scrivere, altrimenti esce ritornando 1
  if(((display_status&DISP_ST_2L_MASK)!=0 && (lcd_counter>=32)) ||
      ((display_status&DISP_ST_2L_MASK)==0 && (lcd_counter>=16)))
    return 1;

  // porta il cursore nella posizione corretta (per ogni evenienza)
  lcd_setCursorPosition(lcd_counter+1);
  sendLetter(letter); // invia la lettera da scrivere
  lcd_counter++; // aumenta il contatore
  // forza il cursore ad andare a capo se si raggiunge
  // la fine della prima linea in modalit doppia riga
  if((display_status&DISP_ST_2L_MASK)!=0 && lcd_counter==16)
    lcd_setCursorPositionRC(2,1);
  return 0; // dato scritto correttamente
}
```

Listato 5.10: Funzione lcd_printText

```
// permette di mandare a schermo una sequenza di caratteri
unsigned char lcd_printText(unsigned char *letters, unsigned char len) {
  unsigned char i; // contatore
  unsigned char r=0; // numero di caratteri non scritti
  for(i=0; i<len; i++)
    // tenta di mandare a schermo tutti i caratteri;
    //per ogni tentativo fallito, aumenta r
    if(lcd_printLetter(letters[i]))
      r++;
  // restituisce il numero di caratteri che
  //non sono stati scritti a schermo
  return r;
}
```

5.2 Scheda Componenti

In fase di progettazione della scheda componenti, si sono tenute in considerazione le seguenti scelte costruttive:

- non occupare i pin utilizzati dalla scheda di controllo motori;
- alloggiare e cablare la memoria Flash;
- posizionare un'interfaccia di collegamento per eventuali dispositivi comunicanti mediante protocollo SPI;
- posizionare e collegare il display LCD (con apposita circuiteria);
- creare un'interfaccia di collegamento ai tutti i terminali della Freedom Board che non vengono utilizzati dai dispositivi sopra elencati.

Leggendo sul manuale del display, si è notato che, con alimentazione a $5\,V$, la tensione di input corrispondente a un livello logico alto deve essere almeno pari a $4.7\,V$, non erogabili dai pin della Freedom Board. Si è pertanto inserito l'integrato 74244, che consiste in quattro buffer unitari, con funzione di disabilitazione (che, una volta attivata, permette di mettere le uscite dei buffer in alta impedenza). Questi buffer hanno internamente un blocco rigenerativo, che permette di portare l'uscita a circa $5\,V$ (tensione di alimentazione) se la tensione di input raggiunge almeno i $2\,V$, tensione adatta alle funzioni GPIO della Freedom Board.

È stato poi inserito un potenziometro, funzionante come partitore di tensione, da collegare al terminale V_0 del display in modo da poterne regolare il contrasto.

Una ulteriore scelta progettuale è stata la seguente: molti terminali dei diversi dispositivi sono spesso utilizzati collegandoli direttamente all'alimentazione o a massa (ad esempio, il terminale Write Protect della memoria Flash viene sempre tenuto alto). Si è quindi collegato ciascuno di questi pin a un selettore jumper. Questo permette, in una posizione, di cortocircuitare il terminale direttamente con l'alimentazione (o la massa del dispositivo), oppure, spostando il selettore, di connettere il pin del dispositivo a uno dei terminali della Freedom Board, in modo da poter pilotare manualmente il segnale (riprendendo l'esempio precedente, il terminale WP può essere collegato direttamente ai $3\,V$ oppure alla porta C16 del microcontrollore).

Lo schema di collegamento risultante dalle precedenti scelte viene riportato in figura 5.4.

Come si vede, nello schema si distinguono le diverse parti della scheda:

- in alto a sinistra vi sono le tre interfacce per il collegamento di dispositivi SPI;
- in alto, al centro, vi sono i pin di output, che non vengono utilizzati né dalla scheda di controllo dei motori, né dai dispositivi presenti (memoria Flash, display, integrato 74244);

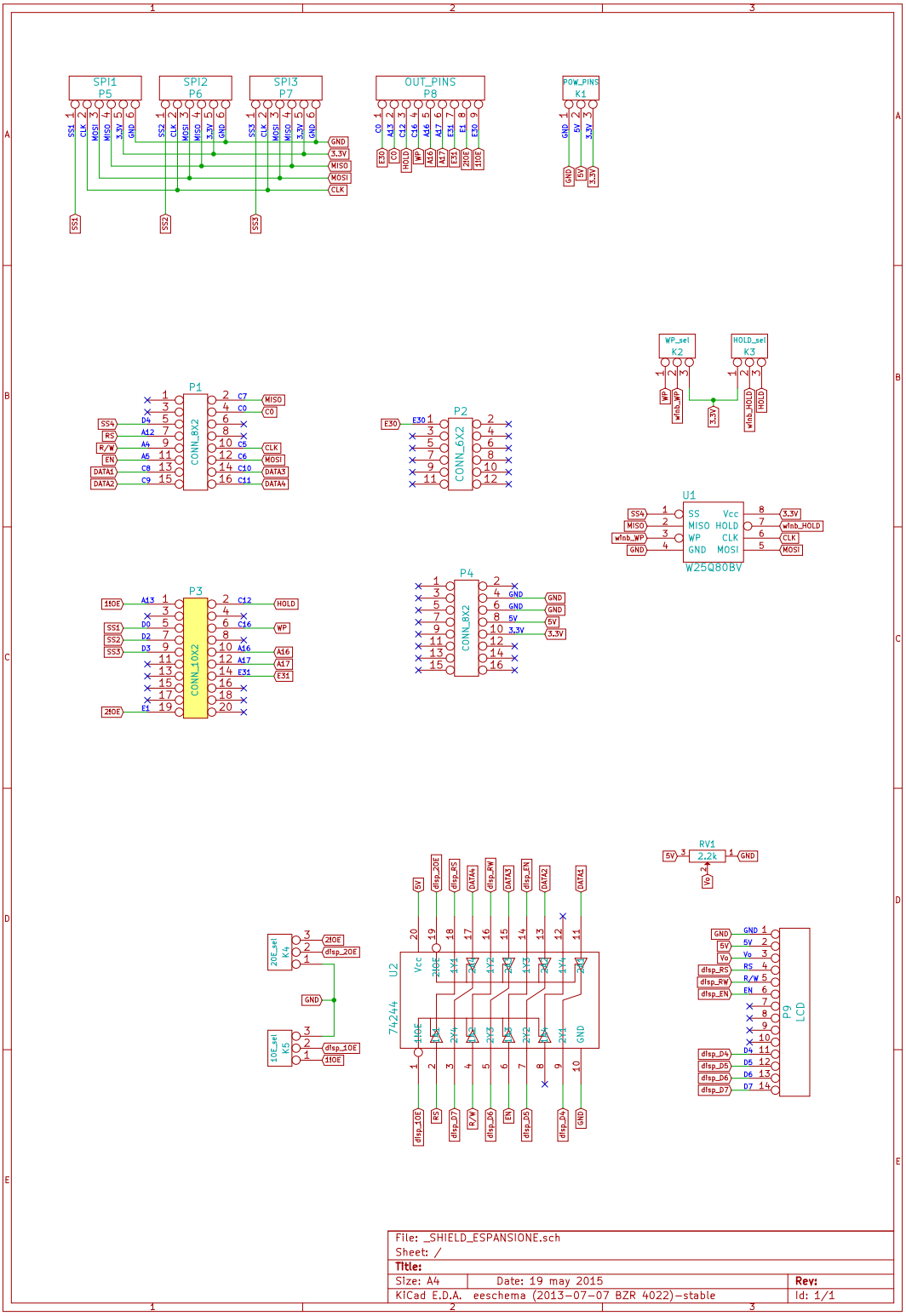
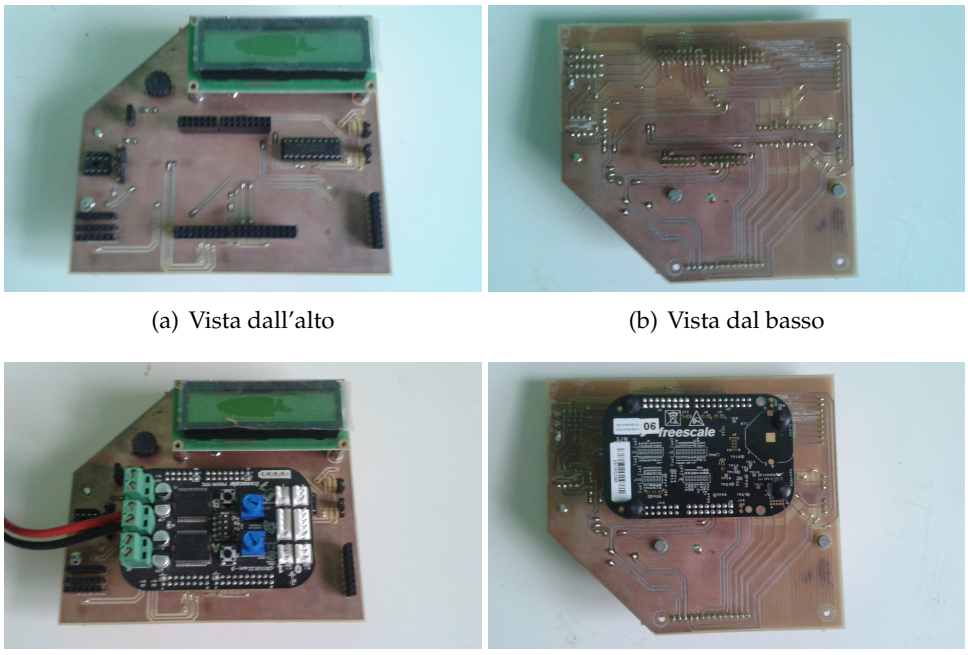


Figura 5.4: Schema della scheda componenti

- in alto a destra vi sono i terminali di uscita di alimentazione (terra, 3.3 V, 5 V);
- al centro a sinistra si trovano i quattro blocchi di pin (P1-P4) corrispondenti ai terminali della Freedom Board
- in centro a destra si trovano il chip di memoria e i due selettori jumper per impostare la sorgente per i segnali WP e HOLD;
- l'integrato 74244 si trova, insieme ai relativi selettori jumper, in basso a sinistra;
- il potenziometro per la regolazione del contrasto e il display LCD si trovano infine in basso a destra nello schema.

Per concludere il capitolo, si mostrano in figura 5.5 la scheda realizzata, mentre i layer superiore e inferiore da utilizzare per stampare il circuito sono visibili in figura 5.6).



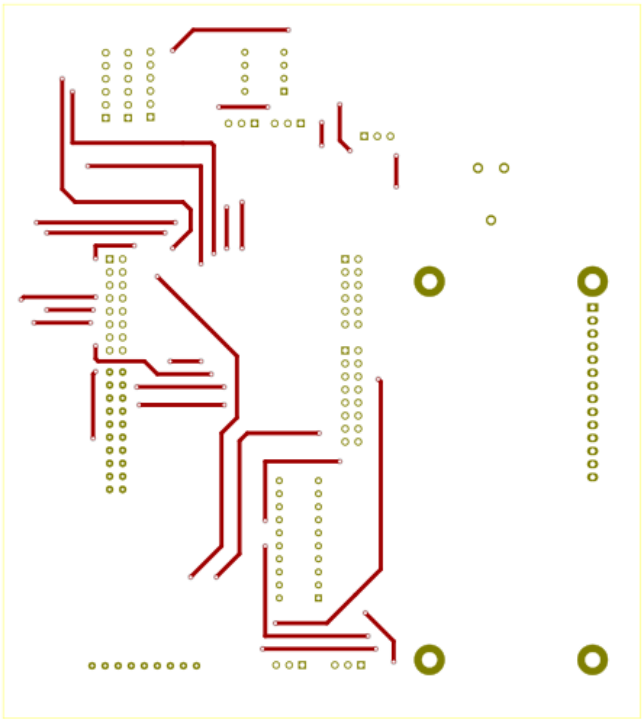
(a) Vista dall'alto

(b) Vista dal basso

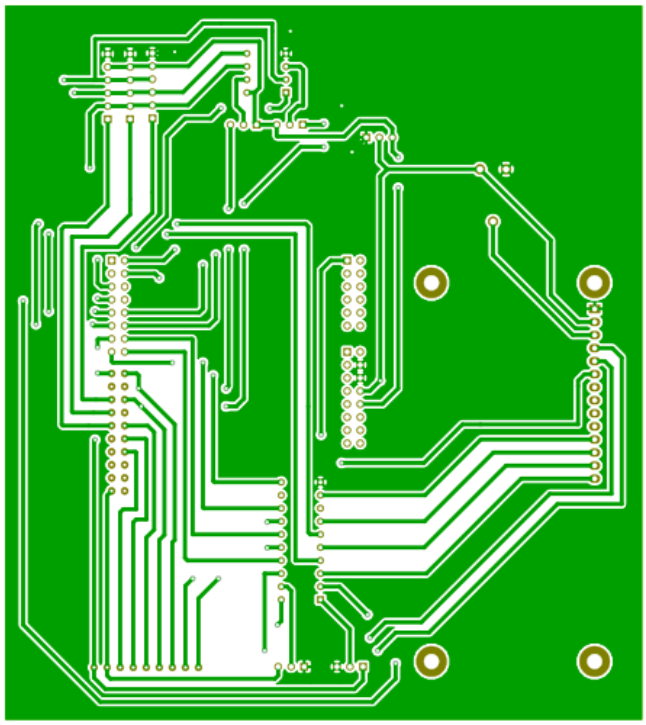
(c) Vista dall'alto con scheda di controllo motori inserita

(d) Vista dal basso con Freedom Board inserita

Figura 5.5: Scheda componenti



(a) Lato superiore



(b) Lato inferiore

Figura 5.6: Layers del circuito stampato

Capitolo 6

Trasferimento e Visualizzazione dei Dati

In questo ultimo capitolo, si illustrerà la soluzione adottata per trasferire i dati dalla memoria Flash a una scheda SD. In seguito, verrà illustrato il software progettato per la visualizzazione e l'elaborazione delle immagini acquisite tramite le telecamere della Freedom Board, mostrando (solo in parte per brevità) il codice che contiene le diverse funzioni implementate per l'analisi e l'interpretazione dei dati.

In conclusione, verrà presentato un algoritmo di simulazione che ha permesso di scrivere una procedura di auto-taratura del tempo di integrazione delle fotocamere.

Gran parte del codice illustrato è scritto in java. Per il lettore che non avesse familiarità con le basi di questo linguaggio si consiglia di consultare [4]. Per la parte legata alla creazione e gestione delle finestre e dei componenti swing si consiglia invece [6].

6.1 Trasferimento delle Immagini

Si è mostrato, nel capitolo 4, il metodo che permette di salvare dati dal microcontrollore alla memoria Flash. Le funzioni descritte sono state dunque utilizzate per salvare su tale dispositivo gli array che rappresentano le immagini visualizzate dalle telecamere.

Come descritto nel capitolo 3, le immagini della telecamera possono essere acquisite sotto forma di vettori da 128 elementi ciascuno, con dati di dimensione pari a un byte.

Considerato che le pagine di memoria del chip W25Q80BV hanno capacità pari a $256\,B$, si è scelto di salvare le immagini a coppie, memorizzando in ogni pagina prima i 128 valori della telecamera sinistra, e quindi riempendo le rimanenti 128 celle con i dati della telecamera destra.

Si è proceduto dunque acquisendo diverse immagini con diversi tempi di integrazione e differenti condizioni di luminosità ambientale. Questo ha permesso di creare

un'ampia collezione di immagini, disponibili senza dover ogni volta alimentare la Freedom Board.

Per il trasferimento dei dati dalla memoria al computer, si è scelto di utilizzare il microcontrollore Arduino, un celebre progetto open-source di costo ridotto e per cui sono disponibili in rete molte librerie pronte all'uso. Si rimanda al sito web del progetto (`https://www.arduino.cc/en`) per approfondimenti in merito all'hardware.

Dal punto di vista operativo, ogni ciclo di acquisizione inizia caricando le informazioni nella memoria Flash e quindi spegnendo la Freedom Board. Si scollega quindi l'integrato dalla scheda componenti, e lo si posiziona su un circuito dedicato collegato ad Arduino. Si avvia il programma di trasferimento che rende disponibili i dati sulla scheda SD, e infine si inserisce quest'ultima in un PC e i dati vengono copiati.

Per il cablaggio, si è utilizzato lo schema di collegamento riportato in tabella 6.1. La scheda SD viene collegata mediante una semplice interfaccia acquistabile in molti negozi online, permettendo la comunicazione con la scheda tramite protocollo SPI.

W25Q80BV

Pin IC Flash	Pin Arduino
/CS	PIN 7
DO	PIN 12
/WP	3V3
GND	GND
Vcc	3V3
/HOLD	3V3
CLK	PIN 13
DI	PIN 11

Scheda SD

Pin SD	Pin Arduino
GND	GND
3V3	3V3
5V	non connesso
CS	PIN 4
MOSI	PIN 11
CLK	PIN 13
MISO	PIN 12

Tabella 6.1: Collegamenti tra Arduino e le periferiche Flash e SD

La scheda elettronica Arduino si può programmare in linguaggio C o C++, similmente a come si fa con la Freedom Board (scrivendo quindi nei registri), anche se per progetti molto semplici si consiglia di ricorrere all'IDE dedicata (scaricabile dal sito). Questa permette agli utenti meno esperti di programmare il microcontrollore in maniera semplice e veloce, utilizzando delle funzioni pre-esistenti che programmano opportunamente i registri del microcontrollore. In particolare, ogni programma (chiamato in gergo *sketch*) scritto in tale ambiente dichiara due funzioni fondamentali: `void setup()` e `void loop()`. La prima contiene il codice che viene eseguito non appena Arduino viene alimentato, mentre la seconda è una funzione che viene chiamata continuamente una volta terminato il metodo `setup`.

Tra le librerie ufficiali disponibli per Arduino, ve n'è una che permette di interfacciare la scheda a una periferica tramite il protocollo SPI, mentre un'altra (basata sulla precedente) consente di leggere e scrivere dati da e su una scheda SD. Avendo

Listato 6.1: Skecth minimo con Arduino

```
// inclusione dei file necessari

void setup() {
  // il codice viene eseguito una sola volta, all'accensione della scheda
}

void loop() {
  // il codice inizia l'esecuzione appena termina la funzione 'setup'
  // una volta concluso, questo metodo viene immediatamente richiamato
}
```

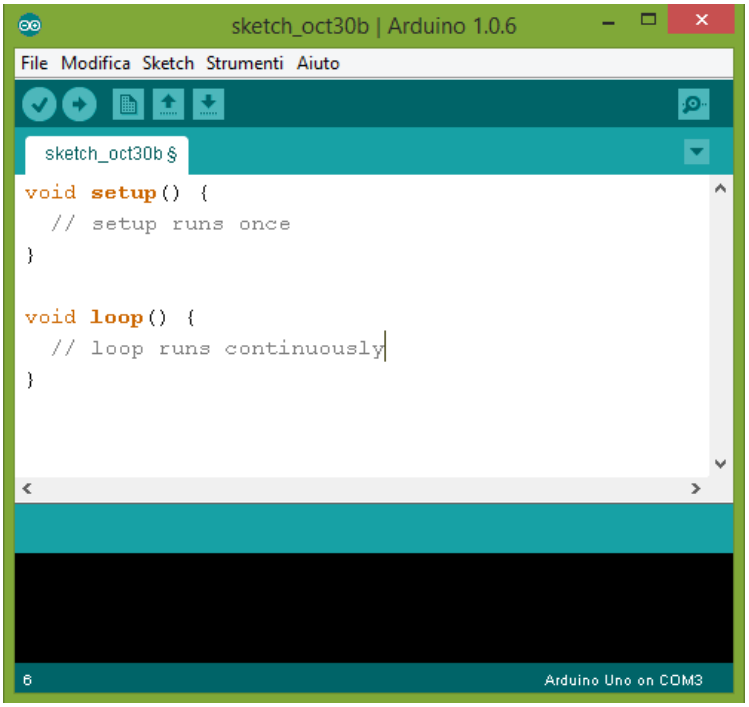


Figura 6.1: IDE per la programmazione di Arduino

già a disposizione tali librerie, si è dunque dovuto solo adattare il codice di gestione della memoria Flash modificando il file sorgente `W25Q80BV.c`.

Una volta pronto il codice di gestione delle periferiche, si è infine scritto lo sketch che permette ad Arduino di trasferire i dati da una memoria all'altra.

In questo caso il programma deve essere eseguito una sola volta, e pertanto tutto il codice è stato posizionato all'interno della funzione `setup`. Tale codice viene riportato in due listati differenti: 6.2 e 6.3 (è stato spezzato in quanto troppo lungo per essere contenuto in una sola pagina).

Listato 6.2: Codice di trasferimento dati - parte 1

```
1  unsigned int i=0;
2
3  Serial.begin(9600);
4  Serial.println("Programma avviato. Inserire tre caratteri (per il nome del
       file, sara' test###.txt");
5  while(!Serial.available());
6  char fileName[] = "test000.txt";
7  String input = Serial.readString();
8  if(input.length()>=3) {
9    for(i=0; i<3; i++)
10     fileName[4+i] = input[i];
11 }
12 else {
13   for(i=0; i<input.length(); i++)
14     fileName[6-i] = input[input.length()-1-i];
15 }
16
17 Serial.print("Inizializzo la flash eeprom... ");
18 delay(700);
19 setupW25Q();
20 delay(5000);
21 Serial.println("Fatto.");
22 delay(700);
23
24 Serial.print("Inizializzo la scheda SD... ");
25 pinMode(10, OUTPUT);
26 if(!SD.begin(4)) {
27   Serial.println("ERRORE SD!");
28   return;
29 }
30 Serial.println("Fatto.");
31 delay(700);
```

Prima di illustrare nel dettaglio le diverse istruzioni riportate, è necessaria una breve premessa. I dati delle telecamere sono rappresentati da array di valori interi, a lunghezza fissa: per memorizzarli correttamente, basterebbe scrivere i dati tenendo a mente che le immagini consistono complessivamente di 256 pixel (128 l'una), e quindi in fase di lettura basterebbe leggere i valori "a blocchi" di 256. Si è tuttavia pensato che con pochi sforzi si potesse rendere più elastica la struttura del file,

Listato 6.3: Codice di trasferimento dati - parte 2

```
boolean file_gia_esistente = SD.exists(fileName);
File f = SD.open(fileName, FILE_WRITE);
if(f) {
  if(file_gia_esistente) {
    f.println(" ");
    Serial.print(fileName);
    Serial.println(" aperto in scrittura");
  }
  else {
    f.println("FILE_ARRAY");
    Serial.print(fileName);
    Serial.println(" creato e aperto in scrittura");
  }
}
else {
  Serial.println("Errore in apertura file.");
  return;
}
delay(700);

Serial.println("Leggo i dati:");
unsigned char d[256] = {0};
unsigned int address = 0x000100;
for(i=0; i<256; i++) {
  d[i] = winb_readByte((unsigned int) (address+i));
  delay(5);
  Serial.print(d[i], DEC);
  Serial.print(" ");
  delay(5);
}
Serial.println("\n");

f.print("length 256 data ");
for(i=0; i<256; i++) {
  f.print(d[i],DEC);
  f.print(" ");
}

Serial.println("Salvataggio completato.");
delay(700);
f.close();
Serial.println(fileName);
Serial.println(" chiuso.");
```

permettendo di memorizzare un numero indefinito di array interi la cui lunghezza sia variabile. Si è deciso che ciascun file contenente i dati deve avere una struttura siffata:

- ogni file dati inizia con la parola chiave `FILE_ARRAY`
- quando si scrive un array, bisogna prima stampare la parola `length` seguita dalla lunghezza (numero positivo) dell'array
- l'inizio di un array viene denotato dalla parola chiave `data`, seguita dai valori interi dell'array

Passando ora al contenuto dei due listati, si può vedere che la prima parte contiene il codice di setup vero e proprio: si comincia inizializzando la comunicazione seriale (tramite cavo USB) con il terminale di input/output integrato nell'IDE: questo permette di interfacciare l'utente e la scheda tramite informazioni di tipo testuale.

Viene quindi richiesto all'utente, tramite il terminale, di indicare un numero a tre cifre che identificherà il nome del file su cui scrivere. Il file dati avrà come nome `test###.txt`, dove al posto di `###` vi saranno le tre cifre indicate.

Successivamente il programma inizializza la memoria Flash, tramite la funzione `setup25Q`, del tutto analoga a quella scritta per la Freedom Board.

Infine, si tenta di inizializzare la memoria SD tramite la funzione `begin`, che prende come input il numero del pin a cui è collegato il terminale Slave Select della scheda.

Concluso questo setup, si controlla se il file indicato (`test###.txt`) esiste già. Qualunque sia il risultato, si procede a chiamare la funzione `open` passando il nome del file e la modalità di accesso (scrittura); la funzione si occuperà di creare il file nel caso in cui questo non fosse già presente in memoria.

Se l'apertura avviene correttamente, si può iniziare a editare il file. Nel caso questo sia stato appena creato, si procede scrivendo all'interno dello stesso la parola chiave `FILE_ARRAY` e notificando l'utente dell'avvenuta creazione.

Se per quache motivo la creazione del file non può avvenire, il programma avvisa l'utente che la procedura non può essere completata, e termina.

Nel listato riportato, il codice legge i dati contenuti nella sola pagina di memoria avente indirizzo `0x000100`: ovviamente la sequenza di istruzioni può essere agevolmente modificata in maniera da compiere diversi cicli di trasferimento caricando le immagini da indirizzi differenti.

In sintesi, il programma provvede a leggere tutti i dati della pagina, mostrandoli sul terminale, e li scrive dunque sul file contenuto nella scheda SD (avendo cura di scrivere prima la sequenza di parole chiave `lenght 256 data`).

Una volta concluso il trasferimento, si chiude il file tramite la funzione `close`, e il metodo `setup` finisce. A questo punto, fino a che Arduino continua ad essere alimentato, si continua a eseguire la funzione (vuota) `loop`.

6.2 Caricamento dei Dati

Per l'elaborazione delle informazioni acquisite, si è valutata la possibilità di creare un'applicativo in diversi linguaggi di programmazione. Subito si è pensato a Matlab, un ambiente di programmazione avanzato che mette a disposizione un enorme numero di pacchetti specifici per l'elaborazione e l'analisi di dati. Tale software è però concesso a pagamento, e pertanto non è di facile accesso per chiunque.

Valutate altre possibilità, si è infine deciso di creare un applicativo basato sul linguaggio java: questo linguaggio, oltre ad essere molto diffuso e multipiattaforma, permette di creare con facilità finestre grafiche anche abbastanza complesse. Le librerie standard sono molto complete, e per queste è disponibile una documentazione online (che può essere scaricata e consultata anche offline) che descrive approfonditamente il funzionamento delle diverse librerie utilizzabili. La documentazione online è consultabile all'indirizzo `docs.oracle.com/javase/7/docs/api/`.

Java, in quanto linguaggio orientato agli oggetti, fa uso, per la creazione dei programmi, delle cosiddette "classi". Pertanto, al fine di caricare i dati, si è creata una classe denominata `FileArray`: di questa si riporta lo "scheletro" nel listato 6.4 (i metodi sono solo dichiarati, senza però che venga mostrato il codice in essi contenuto).

Come si vede, si utilizza un tipo particolare di eccezione, non presente nelle librerie standard: `FileArrayException`. Questa classe è stata definita estendendo direttamente `IOException`, e ha lo scopo di indicare se il file che si vuole leggere contiene errori di formato secondo lo standard definito nel paragrafo precedente (si è deciso di creare una sottoclasse di `IOException` per ragioni di chiarezza, in quanto permette di capire subito il tipo di problema incontrato). Grazie all'ereditarietà delle classi, essendo `FileArrayException` un discendente di `Exception` ma non di `RuntimeException`, l'eccezione definita rientra nella categoria delle cosiddette "eccezioni controllate". Questo vuol dire che, qualora si utilizzi un metodo che potrebbe generare un'eccezione di tale tipo, è necessario racchiudere le istruzioni passibili di errore all'interno di un blocco "`try-catch`". Il codice, estremamente semplice, della classe `FileArrayException` è visibile nel listato 6.5.

Si riporta nella tabella 6.2 una breve descrizione delle funzioni implementate. In generale, quello che permette di fare la classe è aprire un file specificato, e leggerne il contenuto cercando di estrarne i dati sotto forma di array interi utilizzando la classe `Scanner`. In aggiunta, un file invece che essere letto può essere editato, inserendo al suo interno array interi scritti nel formato corretto.

Per brevità, non si riporta il codice di tutte le funzioni, ma solamente di `openFile(boolean mode)` (listati 6.6 e 6.7), `readArray()` (listato 6.10) e `hasNextArray()` (listati 6.8 e 6.9).

Tabella 6.2: Funzioni della classe `FileArray`

Nome	Descrizione
`FileArray()`	

continua nella pagina successiva

continua dalla pagina precedente

Costruttore della classe. Inizializza le variabili di istanza a un valore di default (`null` per gli oggetti, `true` per la variabile `reading`).

`void openFile(String file_path, boolean mode)`
: Apre un file avente il percorso specificato da "`file_path`". Il metodo non fa nulla qualora sia già aperto un altro file. La modalità (`mode`) indica se il file deve essere aperto in scrittura (`false`) o in lettura (`true`). Si consiglia di utilizzare le costanti `LETTURA` e `SCRITTURA`. Il metodo genera eccezioni di tipo `FileNotFoundException` nel caso in cui non si riesca ad aprire il file. Se il file viene aperto in lettura, ma il formato non è corretto (in altre parole, se non inizia con la parola chiave `FILE_ARRAY`) viene lanciata un'eccezione di tipo `FileArrayException`.

`void openFile(boolean mode)`
: Ha lo stesso comportamento del metodo precedente, con la differenza che la selezione del file avviene mediante una finestra di selezione grafica. Nel caso l'utente chiuda la finestra senza selezionare un file, il metodo termina automaticamente.

`void closeFile()`
: Permette di chiudere il file precedentemente aperto con uno dei metodi `openFile`. Se non era stato aperto nessun file, il metodo si conclude senza effettuare alcuna operazione.

`boolean opened()`
: Il metodo restituisce `true` nel caso sia già stato aperto un file tramite uno dei metodi `openFile`, `false` altrimenti.

`boolean reading()`
: Restituisce `true` se un file è correntemente aperto in lettura, `false` se aperto in scrittura. Nel caso in cui nessun file sia correntemente aperto, il valore di uscita andrebbe ignorato in quanto non significativo.

`boolean reading()`
: Restituisce l'equivalente di `!(reading())`. Anche in questo caso, se nessun file è correntemente aperto, il valore restituito andrebbe ignorato.

`int [] readArray()`

continua nella pagina successiva

continua dalla pagina precedente

Legge il prossimo array contenuto nel file di lettura e lo restituisce. Restituisce null nei casi seguenti: non è ancora stato aperto un file, il file aperto è in modalit scrittura, il file non contiene dati utili (si è raggiunta la fine del file o non è presente la parola chiave `length`). Il metodo solleva un'eccezione di tipo `FileArrayException` in questi casi (riconducibili tutti a un formato errato): dopo la parola chiave "length" non è presente un numero che identifica la lunghezza dell'array; dopo la lunghezza dell'array non è presente la parola chiave `data`; i dati che vengono letti non sono sufficienti a completare l'array. Si tiene a precisare che questo metodo permette di "riconoscere" solo i suddetti tipi di errore. Esistono casi in cui un file scritto in maniera scorretta può comunque essere letto: si consiglia pertanto di affidare la scrittura dei file al metodo `writeArray()`, che garantisce la scrittura dei dati nel formato corretto.

`boolean hasNextArray()`
: Comunica all'utente se è presente un nuovo array da leggere nel file. Se non è ancora stato aperto nessun file, oppure se è stato aperto un file in scrittura, viene restituito false. Può sollevare un'eccezione di tipo `FileArrayException` qualora il file in lettura contenga uno dei seguenti errori di formato: dopo la parola chiave `length` non è presente un numero che identifica la lunghezza dell'array; dopo la lunghezza dell'array non è presente la parola chiave `data`; i dati che vengono letti non sono sufficienti a completare l'array. Vale in generale quanto detto per il metodo `readArray()`: esistono errori di formato che non generano eccezioni, e per evitare simili eventualità si consiglia di utilizzare il metodo `writeArray()` per la scrittura dei file nel formato corretto.

`void writeArray(int [] a)`
: Scrive un array di interi nel formato corretto. Se non è ancora stato aperto un file, oppure se il file attualmente aperto è in modalit lettura, il metodo termina senza eseguire nessuna istruzione. Se viene passato un riferimento `null` il metodo termina senza eseguire nulla.

Il metodo `openFile` mostrato permette di aprire un file specificato dall'utente. La selezione del file in questo caso avviene tramite una finestra grafica, utilizzando la classe `JFileChooser` del pacchetto `javax.swing`. Dalla finestra che si apre, l'utente si può spostare nelle varie cartelle presenti sul PC; una volta selezionato il file, il pannello si chiude, restituiendo una valore intero che rappresenta l'azione selezionata dall'utente. Se è stato selezionato un file (e cioè il pannello ha restituito il valore `JFileChooser.APPROVE_OPTION`) si salva il path corrispondente all'elemento scelto in una stringa, che viene quindi utilizzata per creare un oggetto di tipo `File`. A partire da quest'ultimo, si cerca di creare un nuovo oggetto di tipo `Scanner`, che permette di leggere o scrivere dati sul file. Nel caso in cui il file indicato non sia apribile per qualche ragione, verrà lanciata un'eccezione di tipo `FileNotFoundException`.

Prima di concludere il metodo, vengono eseguite alcune operazioni legate al forma-

Listato 6.4: Struttura della classe `FileArray`

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4  import java.util.Formatter;
5  import javax.swing.JFileChooser;
6  import javax.swing.JFrame;
7
8  public class FileArray {
9    public static final String CODICE_FORMATO = "FILE_ARRAY";
10   public static final boolean LETTURA = true;
11   public static final boolean SCRITTURA = false;
12
13   private File file;
14   private String last_path;
15   private Scanner input;
16   private Formatter output;
17   private boolean reading;
18   private int[] buff;
19
20   public FileArray() { ... }
21
22   public void openFile(String file_path, boolean mode) throws
         FileArrayException, FileNotFoundException { ... }
23
24   public void openFile(boolean mode) throws FileArrayException,
         FileNotFoundException { ... }
25
26   public void closeFile() { ... }
27
28   public boolean opened() { ... }
29
30   public boolean reading() { ... }
31
32   public boolean writing() { ... }
33
34   public int[] readArray() throws FileArrayException { ... }
35
36   public boolean hasNextArray() throws FileArrayException { ... }
37
38   public void writeArray(int[] a) { ... }
39
40 }
```

Listato 6.5: Classe `FileArrayException`

```
1 import java.io.IOException;
2
3 public class FileArrayException extends IOException {
4   public FileArrayException(String message) {
5     super(message);
6   }
7 }
```

to definito per la trasmissione dei dati. Nel caso in cui il file sia stato aperto in lettura, si controlla che questo inizi con la parola chiave `FILE_ARRAY`. Se così non fosse, il file potrebbe non essere interpretabile da un oggetto `FileArray`, e pertanto si interrompe l'esecuzione del metodo lanciando un'eccezione di tipo `FileArrayException`. Se il file viene invece aperto in scrittura, la prima operazione che si porta a termine è quella di scrivere, all'inizio del testo, `FILE_ARRAY`. In questo modo una volta chiuso, tale file sarà leggibile dagli oggetti `FileArray`.

Il metodo `hasNextArray` permette di controllare, nel caso in cui un file sia aperto in lettura, se nel file è presente un array leggibile. Per farlo l'idea è quella di leggere i dati contenuti nel file e cercare di capire se questi corrispondono a un array scritto nel formato prestabilito. Bisogna considerare tuttavia che gli oggetti `Scanner`, quando leggono dati dallo stream di input, utilizzano un indice che permette di identificare la posizione attuale nel file. Quando viene eseguita un'operazione di lettura, i dati letti vengono "consumati", nel senso che l'indice di posizione viene progressivamente incrementato, e non è più possibile leggere i dati precedenti. Pertanto, quando si leggono i dati per controllare se è presente un array leggibile, è anche opportuno salvare fin da subito i valori acquisiti.

Dopo aver tentato di leggere le parole chiave `length` e `data` (lanciando una `FileArrayException` nel caso in cui queste non siano presenti), il metodo provvede quindi a salvare l'eventuale array nell'oggetto `buff`. Se durante la lettura non si riscontrano errori di formato, tale buffer viene riempito correttamente, e il metodo restituisce il valore `true`.

Un accorgimento è stato quello di avviare la procedura di lettura del dato solo nel caso in cui `buf` sia vuoto (il suo contenuto sarebbe in questo caso `null`). Così facendo, a una prima chiamata del metodo, il buffer viene riempito correttamente. Se l'utente richiamasse `hasNextArray` senza aver prima acquisito il dato, il metodo ritornerebbe immediatamente il valore `true`, poiché l'array memorizzato in `buff` è sicuramente disponibile.

Infine, il metodo `readArray` permette di ottenere un dato letto tramite `hasNextArray`. Per prima cosa, ci si assincera che il dato sia effettivamente disponibile, chiamando il precedente metodo. Come illustrato, tramite questo si è certi di riempire correttamente il buffer solamente con il primo array disponibile, senza andare a sovrascriverlo nel caso in cui questo sia già pieno. Se il valore restituito da `hasNextArray` corrisponde a `false`, vuole dire che nessun ulteriore dato è presente nel file, e si resituisce pertanto `null`. Al contrario, si provvede a copiare il buffer in una variabile

Listato 6.6: Metodo openFile(boolean mode) - parte 1

```
1  public void openFile(boolean mode) throws FileArrayException,
       FileNotFoundException{
2    // si pu operare sul file solamente se esso non  aperto, e cioe' se e' null.
3    // Prima di aprire un file, e' consigliato assincerarsi che non vi sia un
         altro file gia' aperto (in tal caso lo si pu chiudere mediante il metodo
         "closeFile()" )
4    if(file == null) {
5      JFileChooser file_selection_panel;
6      if(last_path == null)
7        file_selection_panel = new JFileChooser();
8      else
9        file_selection_panel = new JFileChooser(last_path);
10     file_selection_panel.setDialogTitle("Selezionare un file in formato .txt");
11     file_selection_panel.setFileSelectionMode(JFileChooser.FILES_ONLY);
12     int n = file_selection_panel.showOpenDialog(new JFrame());
13     if(n != JFileChooser.APPROVE_OPTION)
14       return; // nessun file selezionato
15     file = file_selection_panel.getSelectedFile(); // si crea il file a partire
           dal percorso specificato
16     last_path = file.getParent();
17     if(file.exists() && file.isFile()) { // se il percorso esiste ed e' un file
           si eseguono le operazioni da apertura
18       this.reading = mode; // si imposta la modalita' (true=lettura,
             false=scrittura). Si consiglia di usare le costanti LETTURA e
             SCRITTURA
19     if(reading) { // file da aprire in lettura
20         try { // prova ad associare il file a un oggetto Scanner
21           input = new Scanner(file);
22         }
23         catch(FileNotFoundException e) { // se il tentativo fallisce, viene
               sollevata un'eccezione di tipo "FileNotFoundException". "file"
               viene reimpostato a null.
24           file = null;
25           throw new FileNotFoundException("Errore in apertura del file di
                 lettura");
26         }
27         // prima di finire il metodo, si controlla che il file sia scritto nel
               formato corretto: esso deve contenere come prima parola il codice
28         // "FILE_ARRAY". Tale codice e' contenuto nella costante CODICE_FORMATO.
29         // Nel caso di file malformato si solleva un'eccezione di tipo
               "FileArrayException".
```

Listato 6.7: Metodo openFile(boolean mode) - parte 2

```
1         if(!input.hasNext() || !input.next().equalsIgnoreCase(CODICE_FORMATO)) {
2           input.close();
3           file = null;
4           throw new FileArrayException("Il file non inizia con il formato
                corretto ("+CODICE_FORMATO+")");
5         }
6       }
7       else { // file da aprire in scrittura
8         try { // prova ad associare il file ad un oggetto Formatter
9             output = new Formatter(file);
10        }
11        catch(FileNotFoundException e) { // se il tentativo fallisce, viene
              sollevata un'eccezione di tipo "FileNotFoundException". "file"
              viene reimpostato a null.
12          file = null;
13          throw new FileNotFoundException("Errore in apertura del file di
                scrittura");
14        }
15        output.format("%s %n",CODICE_FORMATO); // appena il file viene aperto
              correttamente, viene scritto in esso il codice "FILE_ARRAY" che
              permette di identificare questo tipo di file dati.
16      }
17    }
18    else { // nel caso in cui il file inserito non sia valido, si esce
          reimpostando "file" al valore null e lanciando un'eccezione.
19      file = null;
20      throw new FileNotFoundException("Il percorso inserito non risulta
            valido");
21    }
22  }
23 }
```

Listato 6.8: Metodo hasNextArray() - parte 1

```
1  public boolean hasNextArray() throws FileArrayException {
2    if(file==null || (file!=null && !reading)) // controllo che l'oggetto sia
         predisposto per la scrittura, in caso contrario restituisco false
3      return false;
4    // se il buffer  stato gi riempito in precedenza, sicuramente un array
         "disponibile"
5    if(buff!=null)
6      return true;
7    // in questo caso il buffer  vuoto. Come prima cosa si legge il file per
         trovare un nuovo array.
8    // quando un array viene letto viene cercata la parola chiave "length"
         seguita dalla lunghezza dell'array
9    // quindi si cerca la parola chiave "data" e gli elementi dell'array
10   String str = ""; // conterr i dati letti di volta in volta dal file di
         lettura
11   // ricerca della parola chiave length
12   while(input.hasNext() && !str.equalsIgnoreCase("length"))
13     str = input.next();
14   // se  stato esaurito il file, sicuramente non c' un array
15   // NOTA: in questo modo, si restituisce null in due casi: se il file non
         contiene dati (in altre parole, se non vi  scritto "length") o se si
16   //  giunti alla fine del file stesso. In tutti gli altri casi (presenza
         della parola "length" seguita da un formato errato di dati) viene
17   // lanciata un'eccezione di tipo RuntimeException.
18   if(!input.hasNext())
19     return false;
20   //acquisizione della lunghezza dell'array
21   str = input.next();
22   int len = 0;
23   try { // si tenta di convertire il prossimo dato in un intero
24     len = Integer.parseInt(str);
25   }
26   catch(NumberFormatException e) { // se ci non  possibile, vuol dire che il
         file  stato malscritto
27     closeFile();
28     throw new FileArrayException("Il file di lettura contiene errori di
           formato");
29   }
```

Listato 6.9: Metodo `hasNextArray()` - parte 2

```
1   // si controlla che dopo la lunghezza dell'array sia presente la parola
        chiave "data". In caso contrario viene lanaciata un'eccezione.
2   if(input.hasNext())
3     str = input.next();
4   if(!str.equalsIgnoreCase("data")) {
5     closeFile();
6     throw new FileArrayException("Il file di lettura contiene errori di
          formato");
7   }
8
9   // viene creato l'array, e tramite un ciclo lo si riempe con i
        corrispondenti dati
10  int[] r = new int[len];
11  int i=0;
12  while(i<len && input.hasNext()) { //ciclo che continua finch non si completa
        l'array (termina comunque in assenza di dati)
13    // procedura per acquisire il prossimo dato e convertirlo in intero.
14    str = input.next();
15    try {
16      r[i] = Integer.parseInt(str);
17    }
18    catch(NumberFormatException e) {
19      closeFile();
20      throw new FileArrayException("Il file di lettura contiene errori di
            formato");
21    }
22    i++;
23  }
24  if(i!=len) { // se i!=len vuol dire che l'array non  stato riempito
        completamente, e pertanto il file presenta un errore.
25    closeFile();
26    throw new RuntimeException("Il file di lettura contiene errori di formato");
27  }
28  else {
29    buff = r;
30    return true; // viene resituito l'array
31  }
32 }
```

Listato 6.10: Metodo `readArray()`

```
1  public int[] readArray() throws FileArrayException {
2    if(file==null || (file!=null && !reading)) // controllo che l'oggetto sia
         predisposto per la scrittura, in caso contrario restituisco null
3      return null;
4    try {
5      if(!hasNextArray())
6        return null;
7      else {
8        int[] r = buff;
9        buff = null;
10       return r;
11     }
12   }
13   catch(FileArrayException e) {
14     closeFile();
15     throw new FileArrayException("Impossibile leggere un nuovo array: il file
           di lettura contiene errori di formato");
16   }
17 }
```

temporanea `r`. Per ultimo, `buff` viene resettato a `null` (indicando quindi che deve essere riempito), e l'array `r` viene restituito, rendendo disponibili i dati.

6.3 Processing

Per la creazione dell'applicazione grafica, si è utilizzato l'ambiente di programmazione *Processing*. Questo è un progetto open-source basato sul linguaggio java, che permette di sviluppare semplici applicazioni grafiche.

Da java eredita interamente la sintassi, per cui tutto il codice sviluppato in tale linguaggio può essere riutilizzato facilmente quando si crea un'applicazione con Processing.

Di fatto, l'ambiente definisce una serie di primitive grafiche che permettono di creare in automatico una finestra pronta per essere riempita con oggetti grafici come linee, circonferenze, rettangoli, immagini e altro ancora.

Le potenzialità di Processing sono enormi, e non verrano esposte nel dettaglio. Ci si limiterà invece a presentare alcuni dei comandi chiave utilizzati all'interno del presente progetto, invitando i lettori interessati ad approfondire autonomamente l'argomento (come prima fonte di informazioni, si può consultare la documentazione del progetto, riportata in [1]).

Un'applicazione Processing consiste di due funzioni fondamentali: `void setup()` e `void draw()`[1].

[1]Si noti la struttura molto simile a quella di un sketch scritto tramite l'IDE di Arduino (come visto nel paragrafo 6.1). Questa somiglianza non è affatto casuale: l'IDE di Arduino è stata creata modificando il codice sorgente dell'IDE di Processing, adattandolo alle esigenze del microcontrollore.

Figura 6.2: IDE per la programmazione di Processing

La funzione `setup` viene chiamata una volta all'inizio dell'esecuzione del programma, e in genere contiene il codice necessario per inizializzare i diversi oggetti e le variabili dichiarate. Il metodo `draw` viene invece chiamato continuamente, e contiene il codice che solitamente serve a disegnare le diverse figure nella finestra grafica.

Durante l'esecuzione, i diversi elementi grafici da visualizzare vengono inseriti in un buffer. Al termine dell'esecuzione dei metodi `setup` e `draw`, il buffer viene svuotato e gli elementi vengono disegnati "in blocco" all'interno dell'area grafica.

In tabella 6.3 si riportano alcune delle funzioni di uso comune, con relativa spiegazione.

In aggiunta ai nuovi comandi, anche alcuni nuovi tipi di dato vengono definiti. Uno di essi è il tipo `color`, che rappresenta un determinato colore. Quando si vuole passare un colore a una funzione, solitamente si può passare direttamente un valore di tipo `color`. In alternativa, i metodi hanno quasi sempre definizioni multiple: alcune accettano un valore a virgola mobile che rappresenta una determinata tonalità di grigio, altre prendono in ingresso una terna di valori che rappresenta un colore nello spazio RGB (Red-Green-Blue) o HSB (Hue-Saturation-Brightness). Nuovamente, si invita il lettore a consultare la documentazione di Processing per saperne di più.

Tabella 6.3: Funzioni di uso comune in Processing

Nome	Descrizione
`void size(int w, int h)`	Funzione solitamente chiamata all'inizio del metodo `setup` che permette di impostare la dimensione della finestra di disegno. I parametri `w` e `h` indicano rispettivamente la larghezza e l'altezza (in pixel) della finestra.
`void background(float gray)`	Colora l'intera finestra della tonalità di grigio specificata tramite il parametro `gray`. Un valore pari a 0 corrisponde al nero, mentre 255 corrisponde al bianco.
`void backgorund(float r, float g, float b)`	Ha lo stesso comportamento del metodo precedente, con la differenza che il colore non viene specificato in scala di grigi, ma tramite le componenti RGB.
`void point(float x, float y)`	Disegna un punto in corrispondenza della coppia di coordinate (`x`,`y`).
`void line(float x1, float y1, float x2, float y2)`	Disegna una linea che congiunge i due punti (`x1`,`y1`) e (`x2`,`y2`).
`void rect(float tlcx, float tlcy, float w, float h)`	

continua nella pagina successiva

continua dalla pagina precedente

Disegna un rettangolo di base `w` e altezza `h` il cui vertice in alto a sinistra (Top-Left-Corner) si trova in posizione (`tlcx`,`tlcy`).

`void stroke(float gray)`
Imposta il colore dei punti, delle linee e dei contorni delle figure al valore in scala di grigio specificato da `gray`.

`void stroke(float r, float g, float b)`
Imposta il colore dei punti, delle linee e dei contorni delle figure al valore in scala di grigio specificato dalle componenti `r`, `g` e `b`.

`void fill(float gray)`
Imposta il colore in scala di grigio per il riempimento delle figure create tramite le primitive 2D (ad esempio `rect`).

`void fill(float r, float g, float b)`
Imposta il colore (in valori RGB) per il riempimento delle figure create tramite le primitive 2D (ad esempio `rect`).

`void strokeWeight(float weight)`
Imposta lo spessore di punti linee e contorni (in pixel) specificato tramite il parametro `weight`.

6.4 Pannello di Visualizzazione

Per la visualizzazione dei dati si è deciso di creare la classe `DataPanel`, che definisce dei pannelli che permettono di disegnare un array come se fosse la traccia di un oscilloscopio.

Il prototipo "pubblico" della classe viene riportato nel listato 6.11. Essendo la definizione della classe molto lunga, non si intende mostrarla tutta nel dettaglio: se ne illustrerà solo il funzionamento generale, escludendo tutti i metodi ausiliari privati e senza mostrare ulteriore codice.

Si è deciso di confinare i dati all'interno di un rettangolo, che rappresenta il pannello. Il rettangolo viene dunque caratterizzato dalla posizione dell'angolo in alto a sinistra e dalle dimensioni (larghezza e altezza) dello stesso. Tramite i metodi `setPosition`, `getXTLC`, `getYTLC`, `setDimension`, `getWidth` e `getHeight` è possibile modificare e ottenere tali valori.

Le tracce da visualizzare vengono salvate all'interno di una variabile `float [][] data`, e cioè, dal punto di vista logico, un array che contiene array di tipo `float`. Ciascun array "elementare" può avere una lunghezza qualunque (anche se i risultati migliori, a livello di visualizzazione, si ottengono quando tutte le tracce hanno uguale lunghezza). L'array "esterno" – quello, che contiene gli array di tipo `float` – viene trattato come array parzialmente riempito, e una procedura apposita per-

Listato 6.11: Prototipo della classe `DataPanel`

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4  import javax.swing.event.*;
5
6  public class DataPanel {
7    public DataPanel(float Xtlc, float Ytlc, float w, float h) throws
         IllegalArgumentException {...}
8    public DataPanel(float Xtlc, float Ytlc, float w, float h, boolean
         auto_scaling) throws IllegalArgumentException {...}
9
10   public void setPosition(float newXTLC, float newYTLC) {...}
11   public float getXTLC() {...}
12   public float getYTLC() {...}
13   public void setDimension(float newWidth, float newHeight) {...}
14   public float getWidth() {...}
15   public float getHeight() {...}
16
17   public void addData(float[] new_data) {...}
18   public void addData(int[] new_data) {...}
19   public void addData(double[] new_data) {...}
20   public float[] getData(int index) {...}
21   public int getNumData() {...}
22   public int getSelected() {...}
23   public void selectData(int index) {...}
24   public void selectNextData() {...}
25   public void resetData() {...}
26
27   public void setOffset(float new_x_offset, float new_y_offset) {...}
28   public float getXoffset() {...}
29   public float getYoffset() {...}
30   public void setMinMaxY(float min_y, float max_y) {...}
31   public void setAxisValues(int x_val, int y_val) {...}
32   public void setAutoScaling(boolean autoScaling) {...}
33   public boolean getAutoScaling() {...}
34   public void setTitle(String new_title) {...}
35   public String getTitle() {...}
36   public void setDataColor(color c) {...}
37   public color getDataColor() {...}
38   public void setStyle(int new_style) {...}
39   public int getStyle() {...}
40
41   public void display() {...}
42
43   public void showSettingsFrame() {...}
44   public void hideSettingsFrame() {...}
45 }
```

mette di ridimensionarlo qualora venga saturato di informazioni, permettendo di aggiungere un numero qualunque di tracce (o meglio, entro i limiti consentiti dalla memoria messa a disposizione del programma).

Al fine di rendere più flessibile l'uso di questa classe, si sono implementati tre differenti metodi, chiamati tutti `addData`, che permettono l'aggiunta di una traccia. Il primo accetta un array di tipo `float` in ingresso, ed esegue le operazioni necessarie a posizionarlo correttamente in `data`. Gli altri due accettando rispettivamente un array `int` e uno di tipo `double`: ciascun metodo provvede a convertire tali dati in array a valori `float` e a chiamare quindi la prima delle tre funzioni passando l'array convertito.

Altre funzioni permettono di accedere a informazioni relative ai dati inseriti: `getNumData` restituisce il numero di tracce inserite, mentre `getData` permette di ottenere uno degli array memorizzati.

Per scelta progettuale, le tracce vengono visualizzate una alla volta: questo vuol dire che bisogna esplicitamente indicare, di volta in volta, quale traccia si intende disegnare all'interno del pannello. Per farlo, si possono utilizzare i due metodi `selectData` e `selectNextData`. Il primo permette di selezionare l'n-esima traccia inserita, mentre il secondo procede con lo scegliere la traccia seguente a quella attualmente visualizzata. La funzione `getSelected` restituisce l'indice dell'array correntemente selezionato.

Infine, il metodo `resetData` permette di svuotare l'array `data`.

All'interno della classe sono presenti diversi metodi per "personalizzare" le impostazioni di visualizzazione.

`setOffset` è un metodo che permette di impostare i margini interni del pannello: si immagini che esista un ulteriore rettangolo contenuto all'interno del pannello, al cui interno verranno visualizzate le tracce. Questo permette di ottenere un risultato migliore, dal punto di vista estetico, per quanto riguarda la visualizzazione delle tracce. I margini corrispondono quindi alla distanza tra i lati del pannello esterno e del rettangolo interno (per ottenere un risultato gradevole alla vista, si potrebbe fissare un margine di 30 pixel su un pannello di larghezza pari a 500 pixel e altezza di circa 300 pixel). Per ottenere i correnti valori di offset si possono usare `getXoffset` e `getYoffset`.

All'interno del pannello vengono visualizzate delle tacchette sul lato sinistro e sul lato inferiore, ciascuna accompagnata da un valore numerico. I numeri sull'asse sinistro corrispondono al valore della traccia a un dato indice, mentre quelli sulla parte inferiore corrispondono agli indici dei diversi elementi contenuti negli array. La gestione dell'asse delle ascisse è automatico, nel senso che si adattano i valori in modo da riempire in larghezza il riquadro. I valori delle ordinate invece si possono impostare manualmente, tramite il metodo `setMinMaxY`, che imposta il valore minimo e massimo per tale asse. Tutti i valori che si troverebbero al di sopra del massimo o al di sotto del minimo vengono "tagliati". Il metodo `setAxisValues` permette di indicare il numero di tacchette che si devono visualizzare sui due assi.

Si è implementata una funzione di visualizzazione particolare, chiamata "autoscaling", che permette di adattare in automatico i valori massimo e minimo delle

ordinate in modo da non tagliare alcun valore della traccia. Per abilitare o disabilitare tale funzionalità, si può utilizzare il metodo `setAutoScaling`, passando come valore del parametro `autoScaling` rispettivamente `true` o `false` a seconda che si voglia abilitare o disabilitare l'adattamento automatico.

È possibilie poi stampare una scritta all'interno del pannello, in modo da poter distinguere diversi pannelli tra loro. Per farlo basta chiamare la funzione `setTitle` inserendo tramite una stringa il titolo da visualizzare.

Ancora, è possibile cambiare il colore della traccia (di default, questa è nera) tramite il metodo `setDataColor`.

Infine, per quanto riguarda la "personalizzazione" del pannello, è possibile impostare uno stile per la traccia tramite il metodo `setStyle`. A questo, si può passare una delle costanti pubbliche `STYLE_LINE`, `STYLE_POINT` o `STYLE_COMBO`. La prima indica di visualizzare le tracce sotto forma di linea spezzata; passando la seconda, i dati verranno disegnati invece sotto forma di punti (uno per ogni valore dell'array); la terza opzione porta a visualizzare entrambi gli stili (si vedrà quindi una serie di punti congiunti da delle linee).

Tutti i metodi sopra elencati permettono solamente di indicare le proprietà del pannello e di caricare i dati, ma non disegnano nulla sulla schermata creata da Processing. Per visualizzare i dati, è infatti necessario richiedere esplicitamente al programma di mandare a schermo le informazioni tramite la funzione denominata `display`.

Un esempio di visualizzazione viene riportato in figura 6.3. Qui si possono osservare in alto una traccia acquisita dalla telecamera, e in basso, in un secondo pannello, la derivata della precedente. Si sono definite dimensioni uguali per entrambi i pannelli. Per l'impostazione degli assi del primo, si sono impostate 25 tacchette orizzontali e 20 verticali, con valori delle ascisse che vanno da 0 a 255. La traccia è blu, e come stile si utilizza quello a linea spezzata. Per la derivata il discorso è simile, con la differenza che gli assi vanno da -100 a +100 e lo stile grafico è misto (punti e linee insieme).

Guardando il listato 6.11 si vede che vi sono ancora due metodi a concludere la classe: `showSettingsFrame` e `hideSettingsFrame`. Si è pensato che fosse molto più comodo avere a disposizione una finestra grafica con cui impostare "in tempo reale" (non, quindi, a livello di codice sorgente) i diversi parametri del pannello. Pertanto, in fase di inizializzazione di un oggetto, viene creato un Frame che non viene però mostrato. Per renderlo visibile basta utilizzare la funzione `showSettingsFrame`, mentre per nasconderlo vi sono due alternative: chiamare `hideSettingsFrame` oppure chiudere il Frame con il pulsante dedicato[2].

La figura 6.4 mostra il Frame delle impostazioni di un pannello.

Come si vede, la finestra di selezione si compone di più parti. In alto si trovano due pulsanti di tipo `JRadioButton`, e cioè dei bottoni di selezione mutuamente esclusivi. Tramite questi, è possibile selezionare se abilitare o meno la funzione di adattamento

[2] A differenza di quanto si potrebbe inizialmente pensare, il Frame non viene "distrutto" premendo il pulsante di chiusura, viene solo reso invisibile. Per distruggerlo bisogna chiudere l'applicazione, in modo che le risorse richieste vengano rilasciate.

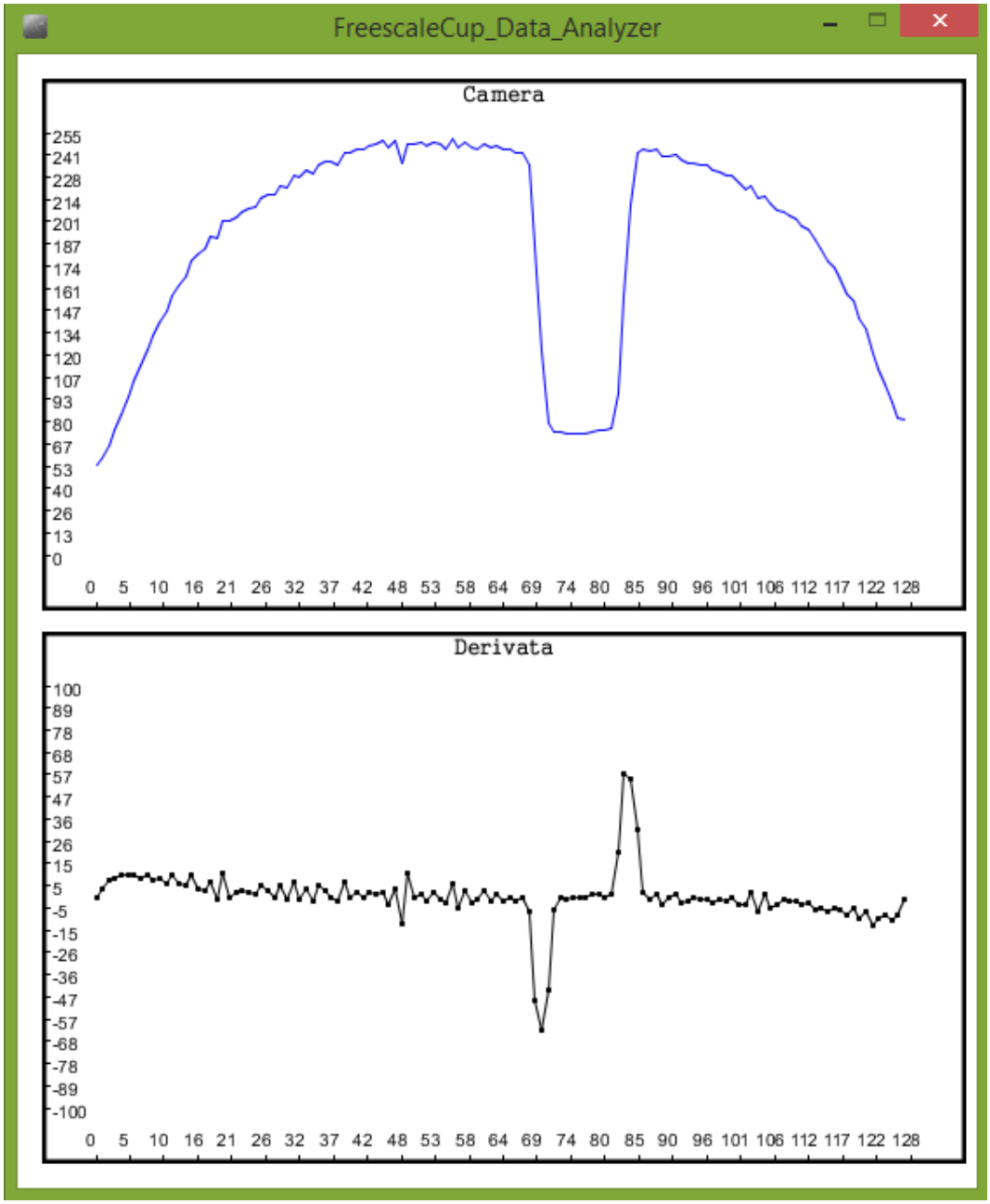


Figura 6.3: Visualizzazzione di tracce tramite `DataPanel`

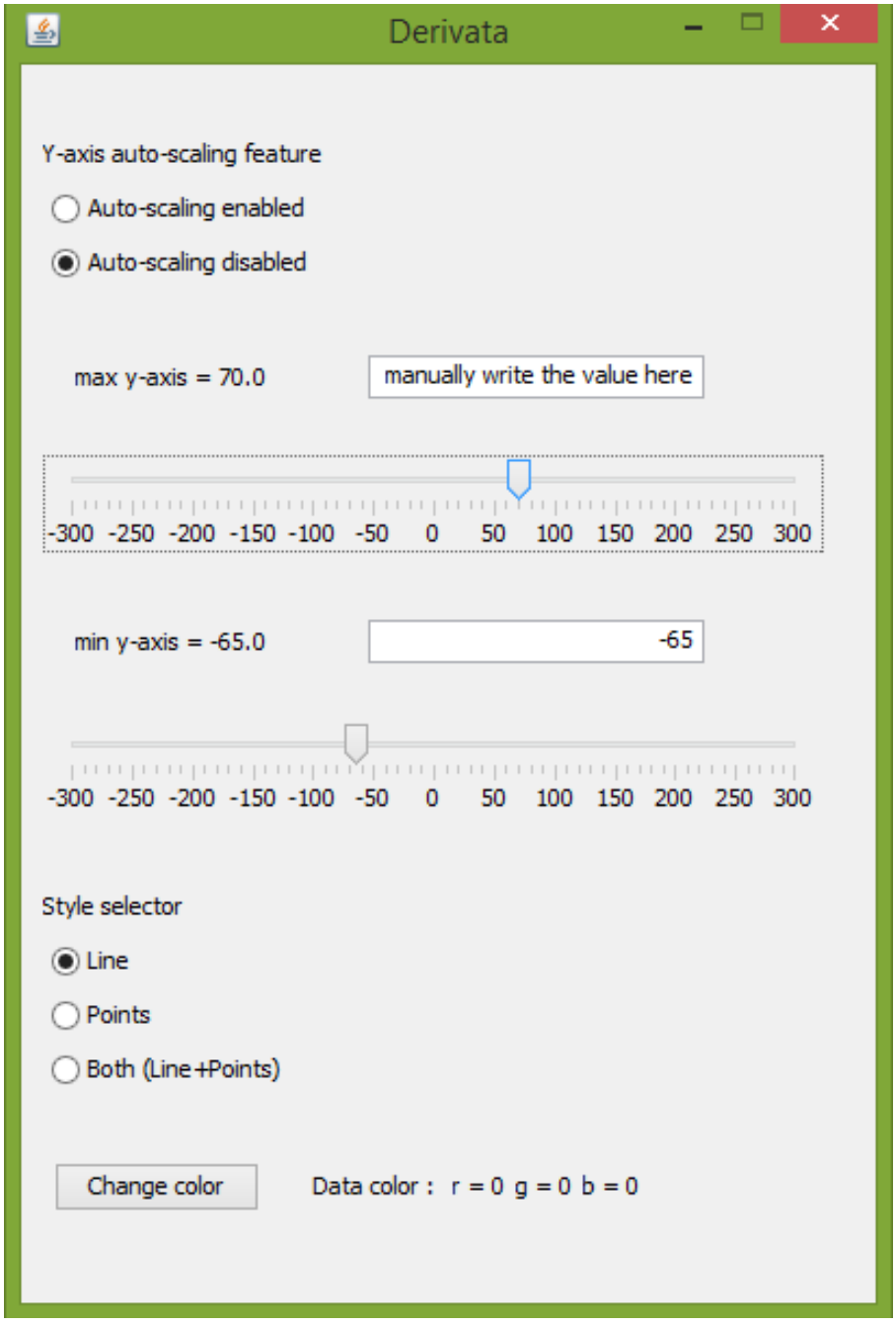


Figura 6.4: Frame delle impostazioni di un Pannello Dati

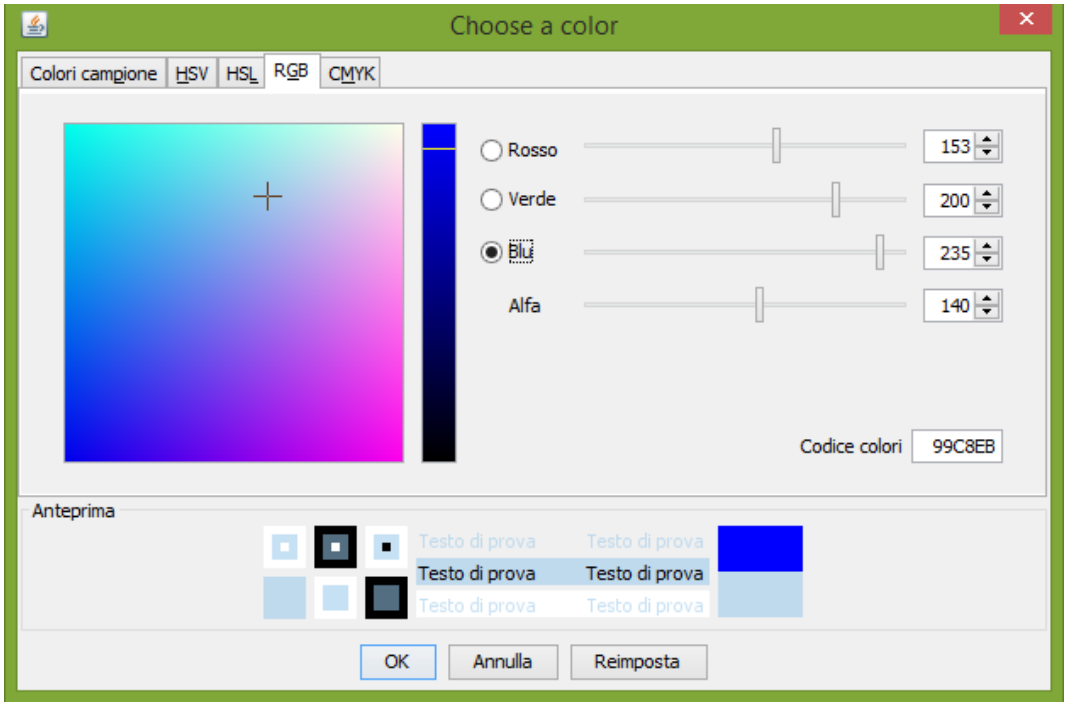


Figura 6.5: Finestra di selezione dei colori con `JColorChooser`

automatico.

Seguono poi una serie di oggetti grafici utili per selezionare i valori massimo e minimo dell'asse verticale (ovviamente il loro effetto è apprezzabile solamente nel caso incui sia attivo il dimensionamento manuale). In tali sezioni, tra loro identiche, si tova innanzitutto un'etichetta che indica l'attuale valore selezionato. A destra di essa, vi è poi un campo di testo in cui è possibile immettere manualmente il valore massimo o minimo. Nel caso in cui non si immettesse un numero ma, ad esempio, una parola, il programma rifiuta l'input e chiede all'utente di inserire un valore numerico al posto di una stringa alfanumerica.

Infine è disponibile una barra a scorrimento (creata con oggetti di tipo `JSlider`) che permette di impostare il valore massimo/minimo a quello puntato dal cursore. Per questioni pratiche ed estetiche, le barre a scorrimento permettono di impostare valori solo nel range $[-300; +300]$. Nel caso in cui si volessero immettere valori più ampi, è necessario far uso dei campi di inserimento manuale.

Per evidenti ragioni grafiche e logiche, i valori massimo e minimo devono essere l'uno maggiore dell'altro. Se l'utente tentasse di violare questa condizione, l'azione verrebbe bloccata, e un messaggio di errore sarebbe visualizzato in una finestra di notifica.

Tramite un gruppo di tre `JRadioButton` si può cambiare lo stile della traccia, selezionando una delle tre alternative.

Infine, si trova un pulsante con la scritta "Change color". Premendolo, si apre una finestra di selezione dei colori (figura 6.5). Questa è ottenibile creando, all'interno

del codice, un oggetto di tipo `JColorChooser`. Una volta selezionato il colore prescelto, la finestra si chiude, e le componenti RGB del colore scelto vengono riportate alla destra del pulsante.

6.5 Funzioni di Elaborazione

L'applicazione creata è stata pensata come software sia di visualizzazione che di elaborazione. Per la seconda funzione, si è quindi creata una libreria di utilità denominata `SignalFunctions` contenente numerosi metodi statici per l'elaborazione delle immagini.

La classe elabora sequenze di valori `float` interpretandoli, in un certo senso, come funzioni a tempo dicreto di durata finita.

Si presentano quindi in tabella 6.4 le funzioni implementate, con una breve descrizione.

Tabella 6.4: Funzioni della classe `SignalFunctions`

Nome	Descrizione
`float[] convert_to_float(int[] a)`	Restituisce la copia di un array a valori interi convertendolo in uno a valori float.
`float[] convert_to_float(double[] a)`	Restituisce la copia di un array a valori decimali a doppia precisione convertendolo in uno a valori float.
`float[] sum(float[] a, float[] b)`	Restituisce una sequenza `r` i cui elementi sono la somma degli elementi di `a` e `b`. Analiticamente, la funzione associata è: $r(i) = a(i) + b(i)$. L'operazione si interrompe qualora le sequenze non abbiano uguale lunghezza.
`float[] subtract(float[] a, float[] b)`	Restituisce una sequenza `r` i cui elementi sono la differenza tra gli elementi di `a` e quelli di `b`. Analiticamente, la funzione associata è: $r(i) = a(i) - b(i)$. L'operazione si interrompe qualora le sequenze non abbiano uguale lunghezza.
`float[] derivata(float[] a)`	Restituisce una sequenza `r` i cui elementi rappresentano la derivata discreta di `a`. Analiticamente, la funzione associata è: $r(i) = a(i) - a(i-1)$. Per convenzione, il primo valore di `r` viene posto uguale a zero.
`float subtract(float[] a, int x)`	Calcola l'integrale discreto di `a` da 0 a `x`, inteso analiticamente come $integrale(a, x) = \sum_{i=0}^{x} a(i)$.

continua nella pagina successiva

continua dalla pagina precedente

`float[] integrale(float[] a)`
: Restituisce una sequenza `r` che rappresenta la funzione integrale associata ad `a`. Utilizzando la precedente funzione, si ha che $r(i) = integrale(a, i)$.

`float mediaIntegrale(float[] a)`
: Calcola il valore medio, inteso in media integrale, di `a`. Detto N il numero di elementi di `a`, si ha: $mediaIntegrale(a) = integrale(a, N)/N$.

`float rms(float[] a)`
: Restituisce il valore medio, inteso come radice della media quadratica, di `a`. Analiticamente, definita la sequenza $f(i) = [a(i)]^2$, si ha: $rms(a) = \sqrt{mediaIntegrale(f)}$.

`float min(float[] a)`
: Restituisce il valore minimo di `a`.

`float max(float[] a)`
: Restituisce il valore massimo di `a`.

`float[] addNoise(float[] a, float max_noise)`
: Restituisce una copia di `a` a cui viene sovrapposto del rumore bianco nel range $[-max\_noise; +max\_noise)$.

`float[] medianFilter(float[] a, int n)`
: Permette di filtrare il segnale `a` tramite un'operazione di media mobile a `n` elementi. Analiticamente, detta `r` la sequenza di uscita, si ha $r(i) = n^{-1} \sum_{k=-n/2}^{n/2} a(i+k)$.

`float[] gaussianFilter(float[] a, int n, float v)`
: Permette di filtrare il segnale `a` tramite un'operazione di media mobile pesata a `n` elementi. I pesi sono calcolati tramite una funzione gaussiana a media nulla e con varianza `v`. Analiticamente, dette `r` la sequenza di uscita e `g` la funzione gaussiana per la pesatura, si ha $r(i) = n^{-1} \sum_{k=-n/2}^{n/2} g(k)a(i+k)$.

`float[] hysteresisFilter(float[] a, float ampl)`
: Questa funzione elabora tramite un'operazione non lineare la sequenza `a`. Il concetto di base è quello di eliminare le piccole oscillazioni della sequenza di input. Analiticamente si può descrivere l'operazione come:
$$r(i) = \begin{cases} a(0) & se\ i = 0 \\ a(i) & se\ |a(i) - r(i-1)| \geq ampl \\ r(i-1) & se\ |a(i) - r(i-1)| < ampl \end{cases}.$$

6.5.1 Considerazioni sul Filtraggio

Le operazioni di filtraggio sono molto importanti al fine di elaborare le immagini acquisite dalle telecamere. Le funzioni `medianFilter` e `gaussianFilter` implementate permettono ad esempio di attenuare il rumore presente sui segnali delle telecamere dovuti alle irregolarità della pista. Ovviamente, un'operazione di filtraggio porta però anche una serie di possibili svantaggi, dovuti al fatto che dopo l'eleaborazione si potrebbero perdere alcune informazioni chiave sul segnale originale.

Come verrà illustrato in seguito, un parametro fondamentale utilizzato per l'analisi dei dati è il valore medio di una traccia, inteso come media integrale, ottenibile dalla funzione `mediaIntegrale`.

Si è dunque affrontato il problema di determinare cosa succede, in seguito a un'operazione di filtraggio, al valore medio di un segnale.

Per il momento, si consideri un generico segnale a tempo continuo $x(t)$. Si supponga che per esso sia applicabile la trasformata di Fourier, per cui $X(f) = \mathcal{F}[x(t)](f)$ e che inoltre valga:

$$I = \int_{-\infty}^{+\infty} x(t)\, dt\,, \ |I| < +\infty \tag{6.1}$$

Si denoti con $H(f)$ la risposta in frequenza di una generica funzione di filtraggio $h(t)$.

Nel dominio temporale, la risposta al filtraggio si ottiene tramite l'integrale di convoluzione tra la funzione filtrante e la funzione originale $x(t)$:

$$y(t) = [x * h](t) \tag{6.2}$$

Nel dominio delle trasformate, l'integrale di convoluzione si traduce invece in un prodotto, per cui vale:

$$G(f) = X(f)\, H(f) \tag{6.3}$$

Si consideri di porre $f = 0$. In tal caso, dalla definizione di trasformata di Fourier, si ha:

$$X(0) = \int_{-\infty}^{+\infty} x(t)\, e^{-j2\pi f 0}\, dt = \int_{-\infty}^{+\infty} x(t)\, dt = I \tag{6.4}$$

Da cui consegue immediatamente che:

$$Y(0) = I\, H(0) \tag{6.5}$$

Si consideri adesso un caso meno generale, in cui si abbiano due funzioni limitate nel tempo, e tali cioè per cui:

$$x(t) = 0 \ \forall t \notin [a_x, b_x]$$
$$h(t) = 0 \ \forall t \notin [a_h, b_h]$$

Con simili ipotesi, è facile comprendere che anche $y(t)$ sarà limitata nel tempo, e, in particolare, gli estremi risultano essere $a_y = a_x + a_h$ e $b_y = b_x + b_h$.

Per un segnale generico $a(t)$ limitato nel tempo vale sicuramente l'uguaglianza:

$$\int_{-\infty}^{+\infty} a(t)\,dt = \int_{t_1}^{t_2} a(t)\,dt \tag{6.6}$$

Pertanto imponendo nella 6.5 che sia $H(0) = 1$ e considerando la 6.6 per entrambi i segnali $x(t)$ e $y(t)$ si giunge al seguente risultato importante:

$$\int_{a_y}^{b_y} y(t)\,dt = \int_{a_x}^{b_x} x(t)\,dt \tag{6.7}$$

Questo mostra come il valore dell'integrale I (definito in 6.1) del segnale $x(t)$ si conservi a filtraggio avvenuto.

Tali considerazioni si possono riformulare nel caso di segnali a tempo discreto. Si consideri la sequenza x_k; nell'ipotesi che questa ammetta trasformata di Fourier a tempo discreto, vale:

$$X(\omega) = \sum_{k=-\infty}^{+\infty} x_k\, e^{-j\omega k} \tag{6.8}$$

Si supponga che valga $\left|\sum_{k=-\infty}^{+\infty} x_k\right| < +\infty$. Si consideri dunque una funzione di filtraggio h_k. La funzione filtrata si ottiene mediante convoluzione discreta delle due funzioni x_k e h_k. Nel Dominio delle trasformate vale invece:

$$Y(\omega) = H(\omega)\,X(\omega) \tag{6.9}$$

Quindi se $H(0) = 1$ si può scrivere:

$$\sum_{k=-\infty}^{+\infty} y_k = \sum_{k=-\infty}^{+\infty} x_k \tag{6.10}$$

Analogamente al caso continuo, se le sequenze x_k e h_k sono a durata limitata, anche la sequenza y_k lo è. In tal caso si può scrivere la seguente uguaglianza:

$$\sum_{k=a_y}^{b_y} y_k = \sum_{k=a_x}^{b_x} x_k \tag{6.11}$$

La 6.11 esprime, in tempo dicreto, lo stesso concetto della 6.7.

Si consideri adesso un caso ancora diverso: sia $x(t)$ un segnale periodico di periodo T. Immaginando di filtrare tale segnale mediante una data funzione $h(t)$, si può facilmente mostrare che il segnale di uscita $y(t)$ è anch'esso un segnale periodico di periodo T.

Si supponga che sia $H(0) = 1$, e si calcoli il valore medio (inteso come media integrale) del segnale $y(t)$ in un suo periodo. Vale la seguente catena di uguaglianze:

$$\begin{aligned}\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} y(t)\,dt &= \lim_{\tau\to+\infty}\frac{1}{\tau}\int_{-\frac{\tau}{2}}^{\frac{\tau}{2}} y(t)\,dt = \lim_{\tau\to+\infty}\frac{1}{\tau}\int_{-\infty}^{+\infty} y(t)\,dt \\ &= \lim_{\tau\to+\infty}\frac{1}{\tau}Y(0) = \lim_{\tau\to+\infty}\frac{1}{\tau}H(0)\,X(0) \\ &= \lim_{\tau\to+\infty}\frac{1}{\tau}X(0) = \lim_{\tau\to+\infty}\frac{1}{\tau}\int_{-\infty}^{+\infty} x(t)\,dt \\ &= \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} x(t)\,dt\end{aligned} \tag{6.12}$$

Questa mostra che, se $H(0) = 1$, il valore medio all'interno di un periodo di $x(t)$ si conserva anche dopo il filtraggio. Una differenza rilevante con i casi visti sopra, è che in questo caso il valore medio si conserva su qualunque periodo delle funzioni di ingresso e di uscita, e pertanto è possibile definire un intervallo "comune" ad entrambe su cui si tale valore viene conservato (mentre, osservando la **??**, si nota che gli integrali, per quanto uguali, vengono calcolati su due intervalli $[a_x, b_x]$ e $\left[a_y, b_y\right]$ differenti).

Similmente a quanto fatto prima, è possibile mostrare che la 6.12 si può tradurre anche in tempo discreto. Il procedimento è del tutto simile a quello appena visto, e si conclude con il risultato notevole:

$$\frac{1}{N}\sum_{k=0}^{N-1} x_k = \frac{1}{N}\sum_{k=0}^{N-1} y_k \tag{6.13}$$

Avendo assunto che x_k e y_k rappresentino gli N campioni di un singolo periodo.

Come si spiegherà più avanti, il valore medio del segnale della telecamera gioca un ruolo fondamentale per la riuscita dell'acquisizione. Per una corretta riuscita delle strategie proposte, la conservazione del valore medio è molto importante, e pertanto si è fatta un'ipotesi molto forte: si è deciso di trattare tutte le immagini acquisite come se fossero un singolo periodo di una sequenza discreta periodica.

Questa scelta è coerente con la teoria delle Trasformate Discrete di Fourier (che verranno affrontate nel prossimo paragrafo), che trattano in generale una sequenza di valori finiti come se fossero appunto il singolo periodo di un segnale periodico a tempo discreto.

Si pone però un potenziale problema: durante un filtraggio, il valore di un pixel della telecamera viene in qualche modo influenzato dai pixel adiacenti[3]. Questo potrebbe creare problemi agli estremi delle immagini, in quanto pixel non correlati tra loro vanno a influenzarsi a vicenda. Ad esempio, si immagini di acquisire un'immagine per metà nera e per metà bianca: tramite il filtraggio, è possibile che i bordi neri vadano a modificarsi di molto a causa dei bordi bianchi all'estremità opposta.

Il problema in realtà, per quanto non del tutto assente, può essere agilmente evitato, considerando la presenza di quel "effetto bordo" descritto in precedenza, secondo cui i pixel laterali dell'immagine hanno sempre valori ridotti, a prescindere dalla luminosità presente. In tal caso, i valori ai bordi saranno abbastanza simili, e quindi si infuenzeranno poco; oltretutto, poichè quei pixel non portano informazioni utili, comunque andrebbero scartati, e pertanto poco importa se si alterano vicendevolmente. Questo porta a concludere che la scelta operata, a patto che si operino alcuni accorgimenti, non andrà a inlfuire eccessivamente sulle acquisizioni.

Stabilite perciò le condizioni sotto cui è possibile conservare il valore medio di una sequenza discreta, si vuole infine mostrare che un filtraggio a media mobile e uno gaussiano rispettano le ipotesi formulate.

Si consideri un'operazione di filtraggio mediante media mobile a N elementi. Assumendo per semplicità che N sia un numero dispari è possibile determinare in maniera univoca il numero intero $M = \frac{N-1}{2}$. Il filtraggio è pertanto descritto dall'equazione:

$$y_k = \frac{1}{2M+1} \sum_{n=-M}^{+M} x_n \tag{6.14}$$

Considerando le proprietà delle trasformate a tempo discreto, è possibile scrivere:

$$Y(\omega) = X(\omega) \frac{1}{2M+1} \sum_{n=-M}^{+M} e^{j\omega n} \tag{6.15}$$

Quindi si può ottenere la risposta in frequenza del filtro come:

$$H(\omega) = \frac{Y(\omega)}{X(\omega)} = \frac{1}{2M+1} \sum_{n=-M}^{+M} e^{j\omega n} \tag{6.16}$$

È facile rendersi conto che, nella 6.16, $H(0) = 1$: si può perciò affermare che la media rimane inalterata anche dopo il filtraggio.

Si consideri ora una qualunque sequenza finita h_k, con $k \in [-M; +M]$. In tal caso, si può definire un'operazione di media pesata nella forma:

[3] Si pensi sempre che il filtraggio si può immaginiare come la risposta di un sistema discreto, descritto da una certa equazione alle differenze, sollecitato da un determinato ingresso.

$$y_k = \sum_{n=-M}^{+M} h_n \, x_{k+n} \tag{6.17}$$

Non è difficile riconoscere in tale formula un prodotto di convoluzione discreto, in cui le due funzioni convolute corrispondono a x_k e h_{-k}. In tal caso, si ha che:

$$H(0) = \sum_{n=-M}^{+M} h_n \tag{6.18}$$

In generale, tale somma non sarà nulla, ma corrisponderà a un certo valore H_0.

Facilmente si conclude che, per ogni sequenza h_k tale per cui $H_0 \neq 1$, è possibile definire la sequenza $\overline{h}_k = \frac{h_k}{H_0}$. Per quest'ultima è facile, grazie alla linearità delle trasformate, mostrare che $\overline{H}_0 = 1$.

Questo risultato ha permesso di definire il metodo `gaussianFilter` in maniera che conservi la media della sequenza di ingresso.

Si noti, prima di passare oltre, che entrambi i filtraggi devono essere operati assumendo che la sequenza in ingresso sia periodica (verrebbe altrimenti a mancare una delle ipotesi fondamentali formulata). Questo richiede di sviluppare i due metodi con un piccolo accorgimento, e cioè che i valori situati all'inizio della sequenza vengano filtrati utilizzando sia i pixel immediatamente successivi, che quelli presenti alla fine dell'array.

Per fare un esempio, si consideri il filtraggio del primo elemento di una sequenza, chiamando la funzione `medianFilter` con `n=5`. In tal caso l'espressione analitica da realizzare sarebbe:

$$y_0 = \frac{x_{N-2} + x_{N-1} + x_0 + x_1 + x_2}{5}$$

6.6 Numeri Complessi

Nella teoria dei segnali un ruolo fondamentale è giocato dalle funzioni a valori complessi e dalle cosiddette Trasformate di Fourier. Tramite queste è possibile studiare il comportamento dei segnali nel dominio delle frequenze invece che da quello temporale. Il vantaggio principale, poi, è che la risposta a un determinato filtro, rappresentata nel dominio temporale da un integrale di convoluzione, si riduce a una semplice moltiplicazione nel dominio delle trasformate, per cui è anche più facile rendersi conto dell'effetto che ha l'applicazione di un particolare filtro.

La Trasformata Discreta di Fourier riveste poi un ruolo fondamentale nell'ambito informatico, in quanto è stato studiato un algoritmo particolarmente performante che permette di calcolarla in tempi molto ridotti.

Siccome tra le librerie standard di java non ve ne è nessuna che permetta di operare con i numeri complessi, si è pensato di creare una classe apposita: `Complex` (lo

scheletro della classe viene mostrato nel listato 6.12). Questa si divide in due parti fondamentali: la prima dichiara i costruttori e i metodi di istanza che permettono di compiere le operazioni fondamentali, mentre la seconda consiste in una collezione di metodi statici utili per l'uso e la gestione di numeri e array a valori complessi.

Nella tabella 6.5 si spiegano le funzioni associate a ciascun metodo. Per chiarezza, ci si riferirà ai numeri complessi usando la notazione $s = \sigma + j\omega$, in cui σ e ω rappresentano rispettivamente la parte reale e immaginarai del numero complesso s.

Tabella 6.5: Funzioni della classe `Complex`

Nome	Descrizione
`Complex(double real, double imaginary)`	Costruttore della classe, permette di costruire un numero complesso avente parte reale e immaginaria specificate dai parametri espliciti.
`Complex()`	Costruttore della classe, permette di costruire il numero complesso nullo: $o = 0 + j0$.
`double re()`	Restituisce la parte reale (σ) del numero complesso.
`double im()`	Restituisce la parte immaginaria (ω) del numero complesso.
`double abs()`	Restituisce il modulo del numero complesso, calcolato come $\|s\| = \sqrt{\sigma^2 + \omega^2}$.
`double arg()`	Restituisce l'argomento – o fase – del numero complesso, calcolato come $\angle s = atan2\ (\omega/\sigma)$.
`Complex conjugate()`	Restituisce il complesso coniugato del numero complesso, calcolato come $\overline{s} = \sigma - j\omega$.
`String toString()`	Restituisce una stringa da stampare a schermo che rappresenta il numero complesso (sovrascrive il metodo `toString` della classe `Object`).
`Object clone()`	Restituisce un oggetto che è copia del numero complesso (sovrascrive il metodo `clone` della classe `Object`).
`boolean equals(Complex c, int digits)`	

continua nella pagina successiva

continua dalla pagina precedente

Confronta il parametro implicito con il numero complesso `c`, restituendo `true` se le parti reali e immaginiarie sono uguali a coppie per almeno `digits` cifre significative dopo il punto decimale (questo in quanto si vuole fronteggiare il problema della perdita di precisione nell'uso dei numeri a virgola mobile).

`Complex fromPolar(double r, double t)`
Restituisce un nuovo oggetto `Complex` caratterizzato da modulo e fase specificati. Per implementare questo metodo, si è utilizzata la relazione di Eulero, secondo cui: $\rho\, e^{j\theta} = \rho\,[\cos\theta + j\,\sin\theta]$.

`Complex sum(Complex a, Complex b)`
Restituisce la somma dei due numeri complessi `a` e `b`.

`Complex subtract(Complex a, Complex b)`
Restituisce la differenza tra i due numeri complessi `a` e `b`.

`Complex product(Complex a, Complex b)`
Restituisce il prodotto dei due numeri complessi `a` e `b`.

`Complex pow(Complex a, double n)`
Restituisce il numero complesso $s = a^n$.

`Complex exp(Complex n)`
Restituisce il numero complesso $s = e^n = e^{\sigma + j\omega}$.

`Complex[] toComplex(int[] f)`
Restituisce un array di numeri complessi i cui valori sono una copia in campo complesso dei valori di `f` (ogni valore avrà parte immaginaria nulla).

`Complex[] toComplex(float[] f)`
Come per il metodo precedente, ma in ingresso riceve un array di tipo `int`.

`Complex[] toComplex(int[] f)`
Come per i due metodi precedenti, ma riceve in ingresso un array di tipo `double`.

`Complex[] resize(Complex[] f, int M)`
Restituisce un array complesso di dimensione `M`, riempendolo con valori presi da `f`. Se `M` è minore della dimensione di `f`, l'array restituito costiuisce una sotto-sequenza estratta da `f`. Al contrario, se `f` contiene meno di `M` elementi, l'array di uscita viene riempito con dei valori nulli. Se `f` ha dimensione `M`, l'array restituito è una copia di `f`.

`Complex[] copy(Complex[] f)`

continua nella pagina successiva

continua dalla pagina precedente

Restituisce una copia di `f`.

`String toString(Complex[] f)`
Restituisce una stringa che rappresenta l'array complesso `f`.

`double[] re(Complex[] c)`
Restituisce un array a valori reali i cui elementi sono le parti reali dei corrispondenti valori in `c`.

`double[] im(Complex[] c)`
Restituisce un array a valori reali i cui elementi sono le parti immaginarie dei corrispondenti valori in `c`.

`double[] abs(Complex[] c)`
Restituisce un array a valori reali i cui elementi sono i moduli dei corrispondenti valori in `c`.

`double[] arg(Complex[] c)`
Restituisce un array a valori reali i cui elementi sono le fasi dei corrispondenti valori in `c`.

`Complex[] conjugate(Complex[] c)`
Restituisce un array complesso i cui elementi sono i complessi coniugati dei corrispondenti valori in `c`.

`Complex[] sum(Complex[] a, Complex[] b)`
Restituisce un array complesso i cui elementi sono la somma dei corrispondenti valori in `a` e `b`.

`Complex[] subtract(Complex[] a, Complex[] b)`
Restituisce un array complesso i cui elementi sono la differenza tra i corrispondenti valori in `a` e `b`.

`Complex[] product(double a, Complex[] f)`
Restituisce un array complesso i cui elementi corrispondono agli elementi di `f` moltiplicati per il numero reale `a`.

`Complex[] product(Complex[] a, Complex[] b)`
Restituisce un array complesso i cui elementi sono prodotto dei corrispondenti valori in `a` e `b`.

`Complex[] DFT(Complex[] f)`
Restituisce la DFT della sequenza `f`.

`Complex[] IDFT(Complex[] F)`

continua nella pagina successiva

continua dalla pagina precedente

Restituisce la IDFT della sequenza `F`.

`Complex[] FFT(Complex[] f)`
: Restituisce la DFT della sequenza `f` calcolata usando un algoritmo FFT. Il metodo funziona solamente con sequenze la cui lunghezza è una potenza di 2.

`Complex[] IFFT(Complex[] F)`
: Restituisce la IDFT della sequenza `F` calcolata usando un algoritmo FFT. Il metodo funziona solamente con sequenze la cui lunghezza è una potenza di 2.

`Complex[] circShift(Complex[] f, int m)`
: Opera uno scorrimento circolare di `m` passi sulla sequenza di ingresso `f`.

6.6.1 Fast Fourier Transform

Come si è detto, uno strumento molto potente per l'analisi dei segnali sono le Trasformate di Fourier, e, in particolare, la Trasformata Discreta di Fourier (DFT).

Il motivo di tale importanza risiede nel fatto che questa utilizza sequenze numeriche aventi un numero finito di elementi, e può dunque essere utilizzata all'interno di un programma informatico. In aggiunta, esiste per tale Trasformata un algoritmo noto in letteratura con il nome di *Fast Fourier Transform*, o brevemente *FFT*. Questo è stato presentato da James Cooley e John Tukey in un articolo intitolato *An Algorithm for the Machine Calculation of Complex Fourier Series* [5], e permette di calcolare la DFT di una sequenza a valori complessi in un tempo proporzionale a $N \log N$ (ove N rappresenta il numero di elementi della sequenza), e non pari a N^2, come avverrebbe applicando la definizione di Trasformata Discreta di Fourier.

A titolo di esempio, si sono registrati i tempi di calcolo, in millisecondi, delle trasformate di sequenze via via più lunghe, utilizzando sia l'approccio "diretto" (calcolo della DFT tramite la definizione) che "ottimizzato" (tramite FFT). I dati sono riportati in tabella 6.6; si noti che, siccome il programma di calcolo è stato eseguito su un PC su cui possono essere attivi altri processi, i tempi non seguono perfettamente le leggi previste.

Al fine di mostrare il funzionamento dell'algoritmo FFT, è necessario partire dalla definizione di Trasformata Discreta di Fourier. Data una sequenza complessa $f_k = [f_0, f_1, \ldots, f_{N-1}]$ di N elementi, anche detta *N-sequenza*, si definisce Trasformata Discreta di Fourier di f_k la N-sequenza F_k definita come:

$$F_k \doteq \sum_{m=0}^{N-1} f_m \, e^{-j\frac{2\pi}{N}mk} \tag{6.19}$$

Per comodità si definisce:

Listato 6.12: Costruttori e metodi di istanza della classe `Complex`

```
1  public class Complex {
2    private double re, im;
3
4    public Complex(double real, double imaginary) {...}
5    public Complex() {...}
6
7    public double re() {...}
8    public double im() {...}
9    public double abs() {...}
10   public double arg() {...}
11   public Complex conjugate() {...}
12
13   public String toString() {...}
14   public Object clone() {...}
15   public boolean equals(Complex c, int digits) {...}
16
17   /* Metodi statici di utilita' */
18   public static Complex fromPolar(double r, double t) {...}
19   public static Complex sum(Complex a, Complex b) {...}
20   public static Complex subtract(Complex a, Complex b) {...}
21   public static Complex product(Complex a, Complex b) {...}
22   public static Complex pow(Complex a, double n) {...}
23   public static Complex exp(Complex n) {...}
24
25   public static Complex[] toComplex(int[] f) {...}
26   public static Complex[] toComplex(double[] f) {...}
27   public static Complex[] toComplex(float[] f) {...}
28   public static Complex[] resize(Complex[] f, int M) {...}
29   public static Complex[] copy(Complex[] f) {...}
30   public static String toString(Complex[] f) {...}
31
32   public static double[] re(Complex[] c) {...}
33   public static double[] im(Complex[] c) {...}
34   public static double[] abs(Complex[] c) {...}
35   public static double[] arg(Complex[] c) {...}
36   public static Complex[] conjugate(Complex[] c) {...}
37   public static Complex[] sum(Complex[] a, Complex[] b) {...}
38   public static Complex[] subtract(Complex[] a, Complex[] b) {...}
39   public static Complex[] product(double a, Complex[] f) {...}
40   public static Complex[] product(Complex[] a, Complex[] b) {...}
41
42   public static Complex[] DFT(Complex[] f) {...}
43   public static Complex[] IDFT(Complex[] f) {...}
44   public static Complex[] FFT(Complex[] f) {...}
45   public static Complex[] IFFT(Complex[] F) {...}
46   public static Complex[] circShift(Complex[] f, int m) {...}
47 }
```

N	$T_{DFT}\,[ms]$	$T_{FFT}\,[ms]$
16	0	0
32	0	0
64	0	0
128	1	1
256	4	1
512	14	1
1024	21	1
2048	29	1
4096	59	2
8192	161	4
16384	599	13
32768	3129	33
65536	8913	94
131072	38195	156
262144	233244	289

Tabella 6.6: Tempi di calcolo della DFT. In tabella N rappresenta il numero di campioni della sequenza, T_{DFT} il tempo impiegato dalla funzione `DFT` e T_{FFT} quello impiegato da `FFT`. I valori nulli sono stati ottenuti in quanto il sistema utilizzato per il rilevamento dei tempi non era in grado di registrare tempi inferiori al millisecondo.

$$w_N \doteq e^{-j\frac{2\pi}{N}} \tag{6.20}$$

Combinando la 6.19 con la 6.20 si ottiene quindi:

$$F_k = \sum_{m=0}^{N-1} f_m \, w_N^{mk} \tag{6.21}$$

Si ipotizzi che N sia una potenza di due[4], e si consideri il numero w_N: esso rappresenta la radice N-esima principale dell'unità. Una proprietà importante delle radici principali dell'unità, ai fini del calcolo della DFT tramite l'algoritmo FFT, è che elevando al quadrato una radice N-esima si ottiene la radice principale (N/2)-esima:

$$(w_N)^2 = \left(e^{-j\frac{2\pi}{N}}\right)^2 = e^{-j\frac{2\pi}{N}2} = e^{-j\frac{2\pi}{N/2}} = w_{N/2} \tag{6.22}$$

Il discorso è valido a patto che $N/2 \in \mathbb{N}$: tale condizione è sicuramente verificata avendo assunto che N sia una potenza di 2. In aggiunta, poiché N è pari, la sommatoria in 6.21 può essere spezzata in due termini, contenenti rispettivamente la somma dei termini con m pari e la somma dei termini con m dispari. Tenendo conto della 6.22, si può scrivere:

[4]Questa ipotesi può essere modificata, creando algoritmi FFT che agiscono su sequenze la cui lunghezza sia una potenza di 3, di 5 o, in generale, di un numero primo qualunque. Per farlo, basta operare in maniera analoga a quanto viene mostrato qui.

$$
\begin{aligned}
F_k &= \sum_{m\ pari}^{N-1} f_m\, w_N^{mk} + \sum_{m\ dispari}^{N-1} f_m\, w_N^{mk} \\
&= \sum_{m=0}^{N/2-1} f_{2m}\, w_N^{(2m)k} + \sum_{m=0}^{N/2-1} f_{2m+1}\, w_N^{(2m+1)k} \\
&= \sum_{m=0}^{N/2-1} f_{2m}\, \left(w_N^2\right)^{mk} + \sum_{m=0}^{N/2-1} f_{2m+1}\, \left(w_N^2\right)^{mk}\, w_N^k \\
&= \sum_{m=0}^{N/2-1} f_{2m}\, w_{N/2}^{mk} + w_N^k \sum_{m=0}^{N/2-1} f_{2m+1}\, w_{N/2}^{mk}
\end{aligned}
\tag{6.23}
$$

Sia ora $L \doteq N/2$. Si possono dunque definire le due L-sequenze:

$$f^{(p)} \doteq [f_0, f_2, f_4, \ldots, f_{N-2}] \tag{6.24}$$

$$f^{(d)} \doteq [f_1, f_3, f_5, \ldots, f_{N-1}] \tag{6.25}$$

Ove $f^{(p)}$ corrisponde alla sotto-sequenza ottenuta considerando i soli termini pari di f_k, mentre $f^{(d)}$ si ottiene considerando i soli elementi dispari. L'espressione finale in 6.23 può essere dunque scritta come:

$$F_k = \sum_{m=0}^{L-1} f_m^{(p)}\, w_L^{mk} + w_N^k \sum_{m=0}^{L-1} f_m^{(d)}\, w_L^{mk} \tag{6.26}$$

Si considerino i due termini in sommatoria riportati nella 6.26. Si può facilmente riconoscere che questi possono essere interpretati come le DFT delle due L-sequenze $f^{(p)}$ e $f^{(d)}$.

Adesso è necessario notare che finora la DFT di una N-sequenza è stata definita solo per $k = 0, 1, \ldots, N-1$. Tuttavia, si comprende facilmente che la definizione di DFT può essere estesa al caso generale $k \in \mathbb{N}$. Così facendo, risulta inoltre che la DFT è una sequenza periodica, di periodo N.

Alla luce di tali considerazioni, risulta perfettamente lecito riscrivere la 6.26 come:

$$F_k = F_k^{(p)} + w_N^k\, F_k^{(d)} \tag{6.27}$$

Ove le sequenze $F_k^{(p)}$ e $F_k^{(d)}$ rappresentano le DFT di $f^{(p)}$ e $f^{(d)}$ rispettivamente.

Nel caso in cui $k \in \{0, 1, \ldots, L-1\}$ l'espressione rimane invariata.

Se invece $k \in \{L, L+1, \ldots, N-1\}$ l'espressione può essere leggermente modificata. Si consideri quindi $k = N/2 + i$ ($i = 0, 1, \ldots$). Tenendo conto che N è una potenza di 2, la 6.27 si riscrive come segue:

$$
\begin{aligned}
F_k &= F_i^{(p)} + w_N^{N/2+i} F_i^{(d)} \\
&= F_i^{(p)} + w_N^{N/2} w_N^i F_i^{(d)} \\
&= F_i^{(p)} + \left(e^{-j\frac{2\pi}{N}}\right)^{N/2} w_N^i F_i^{(d)} \\
&= F_i^{(p)} + \left(e^{-j\frac{2\pi}{N}}\right)^{N/2} w_N^i F_i^{(d)} \\
&= F_i^{(p)} + e^{-j\frac{2\pi}{N}\frac{N}{2}} w_N^i F_i^{(d)} \\
&= F_i^{(p)} + e^{-j\pi} w_N^i F_i^{(d)}
\end{aligned}
\tag{6.28}
$$

E considerato che $e^{-j\pi} = -1$:

$$
F_k = F_i^{(p)} - w_N^i F_i^{(d)} \tag{6.29}
$$

Riscrivendo in maniera più compatta i due casi, si ha infine la relazione base del'algoritmo FFT:

$$
F_k = \begin{cases} F_k^{(p)} + w_N^k F_k^{(d)} & \text{se } k = 0, 1, \ldots, \frac{N}{2} - 1 \\ F_i^{(p)} - w_N^i F_i^{(d)} & \text{se } k = \frac{N}{2}, \frac{N}{2} + 1, \ldots, N \quad \left(i = k - \frac{N}{2}\right) \end{cases} \tag{6.30}
$$

In altre parole, assumendo di avere una funzione chiamata *DFT* che prende in ingresso una sequenza f, il calcolo della DFT può essere effettuato ricorsivamente calcolando le Trasformate delle sotto-sequenze $f^{(p)}$ e $f^{(d)}$, che descrive esattamente il funzionamento dell'algoritmo FFT.

Il listato 6.13 mostra il codice java che permette di ottenere la DFT di una sequenza mediante Trasformata Veloce di Fourier. Si noti come, nel caso in cui $N = 1$, sia $F_k = f_k$: in tal caso non è necessario continuare a calcolare iterativamente la DFT.

Un discorso molto simile riguarda la *Inverse Discrete Fourier Transform*, o semplicemente *IDFT*. Essa corrisponde alla trasformazione inversa alla DFT, ed è definita come:

$$
f_k = \frac{1}{N} \sum_{m=0}^{N-1} F_m \, e^{j\frac{2\pi}{N}mk} \tag{6.31}
$$

Si capisce dalla definizione che è possibile creare un algoritmo *IFFT* che calcoli la IDFT in maniera efficiente analogamente a come si è fatto per la DFT, tuttavia si possono sfruttare le proprietà dei numeri complessi per ridurre il calcolo della Trasformata Inversa a quello di una Trasformata "diretta", per cui è già implementato un algoritmo veloce.

Le proprietà coinvolte sono tutte legate all'operatore "complesso coniugato", e in particolare:

Listato 6.13: Algoritmo FFT in java

```
1  public static Complex[] FFT(Complex[] f) {
2    int N = f.length;
3    // se N=1 la DFT di f  uguale a f stessa
4    if(N==1)
5      return new Complex[] {f[0]};
6    // se la lunghezza di f non  una potenza di 2 l'algoritmo
7    // fallirebbbe, perci si solleva prima un'eccezione
8    if(N%2!=0)
9      throw new RuntimeException("La lunghezza della sequenza non  una potenza di
           due");
10
11   // FFT dei termini pari e dispari della sequenza in ingresso
12   Complex[] pari = new Complex[N/2];
13   Complex[] dispari = new Complex[N/2];
14   for(int i=0; i<N/2; i++) {
15     pari[i] = f[2*i];
16     dispari[i] = f[2*i+1];
17   }
18   pari = FFT(pari);
19   dispari = FFT(dispari);
20
21   // adesso deve sommare i termini
22   Complex[] F = new Complex[N];
23   for(int i=0; i<N/2; i++) {
24     Complex w = Complex.fromPolar(1.0, -2*Math.PI*i/N);
25     F[i] = Complex.sum(pari[i],Complex.product(w,dispari[i]));
26     F[i+N/2] = Complex.subtract(pari[i],Complex.product(w,dispari[i]));
27   }
28   return F;
29 }
```

$$\overline{(a+b)} = \overline{a} + \overline{b} \tag{6.32}$$

$$\overline{(a\,b)} = \overline{a}\,\overline{b} \tag{6.33}$$

$$\overline{\overline{a}} = a \tag{6.34}$$

$$\overline{e^{j\theta}} = e^{-j\theta} \tag{6.35}$$

Si immagini dunque di applicare la 6.34 alla 6.31:

$$f_k = \overline{\overline{f_k}} = \overline{\overline{\frac{1}{N}\sum_{m=0}^{N-1} F_m\, e^{j\frac{2\pi}{N}mk}}} \tag{6.36}$$

Considerata ora la proprietà di linearità (6.32), si ha:

$$f_k = \frac{1}{N}\overline{\sum_{m=0}^{N-1} \overline{F_m\, e^{j\frac{2\pi}{N}mk}}} \tag{6.37}$$

Ancora, si applichi la proprietà descritta in 6.33:

$$f_k = \frac{1}{N}\overline{\sum_{m=0}^{N-1} \overline{F_m}\,\overline{e^{j\frac{2\pi}{N}mk}}} \tag{6.38}$$

Infine, si applica la 6.35:

$$f_k = \frac{1}{N}\overline{\sum_{m=0}^{N-1} \overline{F_m}\, e^{-j\frac{2\pi}{N}mk}} \tag{6.39}$$

La 6.39 mostra che per calcolare la IDFT di una sequenza F_k si può seguire il seguente procedimento:

1. si calcola la sequenza $\overline{F_k}$, avente come elementi i complessi coniugati di F_k
2. si procede calcolando la DFT di $\overline{F_k}$ utilizzando l'algoritmo FFT
3. si coniuga la sequenza appena ottenuta
4. si divide ogni elemento per N

Il listato 6.14 mostra come è stato implementato l'algoritmo in java.

Listato 6.14: Algortimo IFFT in java

```
1 public static Complex[] IFFT(Complex[] F) {
2   if(F.length==1)
3     return new Complex[] {F[0]};
4   Complex[] f = conjugate(F);
5   f = FFT(f);
6   return product((1.0/F.length), conjugate(f));
7 }
```

6.7 Elaborazione dei Dati della Telecamera

Avendo implementato le classi `FileArray`, `DataPanel`, `SignalFunctions` e `Complex`, è tutto pronto per analizzare ed elaborare i dati acquisiti dalle telecamere della Freedom Board.

Tramite Processing, si è creata una finestra di lavoro contenente diversi oggetti `DataPanel`. Le immagini venivano dunque caricate tramite un oggetto `FileArray`, ed elaborate tramite le diverse funzioni definite. I dati visualizzati venivano quindi interpretati, e potevano poi essere salvati per essere ricaricati in un secondo momento.

Per completare l'opera, mancavano tre passi fondamentali: stabilire un procedimento per interpretare i dati, definire quali operazioni di elaborazione fossero necessarie per migliorare la qualità delle immagini e infine implementare e testare alcuni algoritmi che si sarebbero poi trascritti nel codice sorgente del file `camera.c`

6.7.1 Individuazione delle Linee Nere

Il passo fondamentale per analizzare le immagini consiste nel capire i parametri che le caratterizzano. Si ricorda innanzitutto che le telecamere sono in grado di rilevare l'energia luminosa riflessa da una superficie opaca, e che in particolare le superfici scure sono caratterizzate da un ridotto indice di riflessione, al contrario di quelle chiare che assorbono una minima quantità di radiazione incidente.

La pista è caratterizzata da una pista bianca delimitata ai lati da due strisce nere. Ipotizzando di aver impostato il tempo di integrazione in modo tale che le letture siano chiare (il bianco e il nero si distinguno chiaramente), avendo deciso di posizionare due telecamere, una che punta sul lato sinistro della pista e l'altra che si concentra invece sul lato destro, e assumendo di posizionare il veicolo al centro di una porzione rettilinea del tracciato, ci si aspetta che le immagini siano fatte in tale maniera:

- entrambe le immagini presentano nei margini interni una degradazione delle informazioni (a causa di quello che, nel capitolo 3, è stato definito "effetto bordo")
- la telecamera sinistra vede un segnale alto nella parte destra (una volta esclusi i pixel di bordo), in quanto punta alla parte bianca della pista; lo stesso si può

dire dell'immagine destra, in cui i pixel bianchi si trovano però nella parte sinistra dell'array di uscita

- dopo questa parte di pixel bianchi, ci si aspetta di trovare un gradino abbastanza marcato con pixel di valore basso in entrambe le immagini: ciò corrisponde alla presenza delle linee nere laterali
- all'esterno del tracciato, si trova il piano di appoggio della pista (tipicamente, un tavolo o il pavimento) di cui non si hanno a priori informazioni utili; questo vuol dire che i pixel possono assumere qualunque intensità
- le immagini si conludono nuovamente con una serie di pixel distorti dall'effetto bordo

Alla luce di queste considerazioni, si pone il problema di individuare all'interno delle immagini la posizione di ciascuna linea nera. Sono state valutate due alternative: analizzare le derivate delle immagini (intese come differenza tra i valori di due pixel adiacenti) oppure valutare quando queste oltrepassano una determinata soglia.

Il primo approccio si basa sul fatto che l'effetto di bordo si traduce in una salita con pendenza circa costante e tutto sommato ridotta, mentre la transizione da centro pista (bianco) a linea (nero) avviene in maniera brusca, con un valore della derivata molto alto. Pertanto, si potrebbe individuare la posizione della linea nera valutando la posizione del massimo della derivata in valore assoluto.

Questo approccio risulterebbe efficace se si avesse una linea nera singola a centro pista, in quanto si conoscerebbero a priori, e con buona precisione, le caratteristiche dell'immagine rilevata. Tuttavia, la presenza del piano di appoggio del tracciato rende imprevedibile la struttura dell'immagine nelle porzioni esterne, e non si può escludere che i massimi vengano rilevati proprio in tali zone.

Una soluzione al problema potrebbe essere quella di valutare i valori della derivata a partire dal centro pista, e procedendo verso l'esterno: il primo picco derivativo indicherebbe la presenza della linea. Il problema in questo caso è che il cambio di pendenza lungo le zone laterali delle immagini, affette da degradazioni sistematiche delle informazioni, si traduce nella presenza di diversi massimi relativi. Sarebbe dunque opportuno definire una soglia, con cui confrontare i valori di picco: se questi fossero inferiori alla soglia, andrebbero interpretati come semplici cambi di pendenza, mentre se fossero superiori ad essa, potrebbero essere classificati come il punto di transizione da bianco a nero.

La seconda alternativa si basa invece sull'imposizione di una soglia opportuna direttamente all'immagine di partenza. Anche se all'inizio i pixel sono distorti, i loro valori crescono lentamente per raggiungere l'intensità tipica del valore associato a una superficie bianca. In questa porzione, vi sarà quindi una transizione "positiva" dal nero/grigio (media intensità) al bianco (alta intensità). Tale transizione deve essere detetta per indicare che l'immagine si è "stabilizzata". In seguito, si procede analizzando i successivi pixel, fino a quando non si verifica una transizione "negativa" da bianco a nero. In tal caso, si può concludere che la linea sia posizionata lì.

Valutando analogie e differenze tra i due approcci, si consideri che entrambi richiedono l'imposizione di una soglia "sperimentale". L'approccio derivativo richiede di valutare il valore assoluto della differenza tra due pixel adiacenti, e di confrontarli con una soglia. Un esempio di codice protrebbe essere:

```
int i=1, line_position=-1;
while(line_position==-1 && i<DATA_LENGTH) {
  if((data[i]-data[i-1])<-SOGLIA || (data[i]-data[i-1])>SOGLIA)
    line_position=i;
  i++;
}
```

Con il secondo approccio, è necessario valutare invece il valore di ogni pixel, e confrontarlo con la soglia, controllando prima di aver compiuto la transizione postiva e poi quella negativa. Il relativo codice potrebbe essere:

```
int i=0, line_position=-1;
// trova la transizione positiva
while(line_position==-1 && i<DATA_LENGTH) {
  if(data[i]>SOGLIA)
    line_position=0;
  i++;
}
// trova la transizione negativa
while(line_position==0 && i<DATA_LENGTH) {
  if(data[i]<SOGLIA)
    line_position=i;
  i++;
}
```

Supponendo che la linea nera si trovi in corrispondenza del pixel P_{linea}, e che venga trovata correttamente da entrambi gli algoritmi, le prestazioni per i due algoritmi sono diverse: per ogni valore di `i`, l'algoritmo derivativo deve accedere ai valori di due pixel ed eseguire due confronti, con costo approssimativamente proporzionale a $4\,P_{linea}$. Il secondo algoritmo deve invece compiere un accesso per ogni valore di `i`, con un costo proporzionale, quindi, a P_{linea}. Essendo dunque il costo computazionale dell'approccio derivativo più alto, si è quindi preferito ricorrere alla strategia basata sulla soglia direttamente applicata alle immagini.

6.7.2 Definizione dei Filtri di Elaborazione

Un importante passo necessario all'elaborazione delle immagini, è il filtraggio dei dati acquisiti mediante particolari funzioni. Lo scopo è fondamentalmente quello di ridurre il rumore e "sagomare" le immagini in maniera da diminuire i particolari presenti.

Per poter operare su campioni aventi caratteristiche note, si è deciso di acquisire immagini caratterizzate da uno sfondo bianco uniforme e una singola linea nera centrale.

Si premette che, all'interno della raccolta di immagini acquisite, il rumore aveva sempre un'intensità piuttosto bassa, nel senso che solitamente i pixel avevano un andamento abbastanza regolare. Questo si è supposto che potesse essere generato sia dall'elettronica di controllo delle fotocamere, che dalle irregolarità del tracciato. Si sottolinea inoltre che, nelle immagini di buona qualità (caratterizzate da un tempo di integrazione adeguato) il rumore veniva ulteriormente limitato dalla saturazione dei pixel, nel senso che questi si stabilizzavano al massimo valore possibile (255) e il rumore non influiva quasi per nulla sulla loro intensità. Nelle zone degratate dall'effetto bordo, invece, un minimo di rumore era ancora presente, e si traduceva in saltuari cambi di segno della dervata (perciò la traccia, in tale zona, poteva presentare un massimo relativo). Tale effetto risultava comunqe poco marcato.

Considerato quindi che il rumore non giocava un ruolo significativo nelle immagini, si è proceduto cercando di determinare la funzione di filtraggio più adatta. Per farlo, si è proceduto per tentativi, definendo di volta in volta diverse funzioni di filtraggio e valutandone gli effetti.

Per tutti i filtraggi, si è fatto in modo che il valore medio del segnale venisse conservato, basandosi sui risultati ottenuti nel paragrafo 6.5.1.

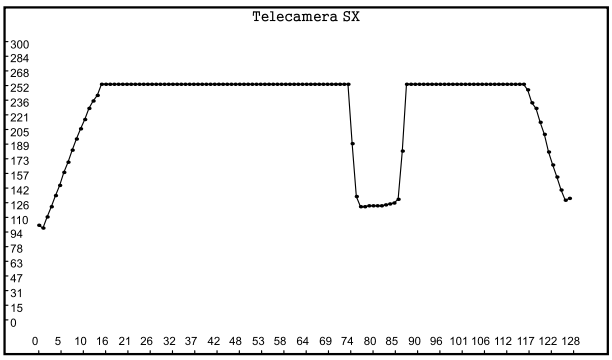
I primi tentativi si sono fatti utilizzando le due funzioni `medianFilter` e `gaussianFilter`. Si sono modificati i parametri caratteristici fino a che non si sono raggiunti dei risultati giudicati "adeguati", filtrando un campione di 20 immagini aventi diversi tempi di integrazione. In figura 6.6 si riportano un esempio di immagine originale (6.6(a)) e i dati ottenuti dopo un filtraggio mediante le due funzioni (6.6(b)). I parametri "ottimali" sono stati stimati come `n=13` per il filtraggio mediante media mobile, `n=11` e `var=2` per il filtraggio Gaussiano.

In secondo luogo, si è tentato di definire dei filtri a valori complessi. Considerato che la risposta all'azione di un filtro avviene, nel dominio delle trasformate, tramite un semplice prodotto, è relativamente semplice individuare un filtro adatto, e tramite le Trasformate Veloci di Fourier il tempo richiesto per portare a termine l'operazione è estremamente ridotto.

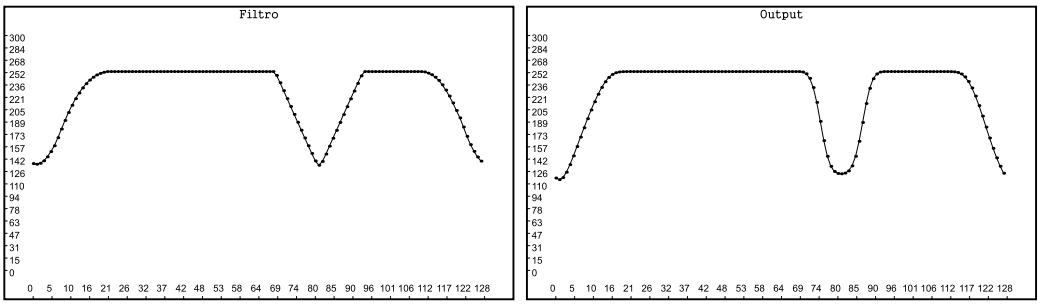
L'obiettivo principale del filtraggio, in questo caso, è quello di sagomare l'immagine. Questo corrisponde, in frequenza, a ridurre la banda del segnale tramite un filtro passa-basso. Il primo filtro preso in considerazione, è il filtro passa-basso ideale. Esso ha, come risposta in frequenza, una finestra rettangolare di ampiezza unitaria e larghezza opportuna.

Fin ora non si è ancora accennato a nessuna relazione esistente tra i diversi tipi di Trasformata. Al fine di giustificare il procedimento con cui i filtri sono stati creati, si vuole solo accennare al fatto che esiste una relazione tra di esse, rimandando a [12] per coloro che volessero avere prove di quanto qui accennato.

Si consideri un segnale a tempo continuo, di durata finita, e si supponga che questo ammetta Trasformata di Fourier. Si immagini adesso di campionare idealmente nel tempo tale segnale con un ben determinato passo di campionamento. In questo



(a) Traccia originale



(b) Traccia filtrata mediante `medianFilter` e `gaussianFilter`

Figura 6.6: Primi tentativi di filtraggio

caso, la Trasformata del nuovo segnale corrisponde a una replicazione nel tempo della sua versione originale moltiplicata per un certo valore legato al passo di campionamento.

D'altra parte, si può definire un segnale a tempo dicreto che riproduce il segnale campionato. Per esso, è definibile la Trasformata di Fourier a Tempo Discreto. Si può dimostrare che in tal caso la Trasformata del segnale campionato idealmente coincide con la DTFT del segnale a tempo discreto.

Ancora, si immagini di replicare la sequenza a tempo discreto in modo da ottener un segnale periodico in cui i valori presi in un singolo periodo corrispondono alla sequenza discreta di partenza. Questo corrisponde, nel dominio delle Trasformate, a campionare la DTFT. Ancora, si può mostrare facilmente che, interpretando i valori del singolo periodo come una N-sequenza, la DTFT campionata coincide con la DFT della N-sequenza appena definita.

Questi risultati permettono di mostrare che, se si vuole implementare un filtro passa-basso ideale tramite una DFT, è necessario costruire un segnale che abbia n valori costanti e pari a 1, quindi prosegua con $N - 2n$ valori nulli, per finire con ancora n valori unitari.

La figura 6.7 mostra a sinistra un esempio di filtro passa basso ideale implementato (nel dominio delle frequenze) avente come valore caratteristico $n = 14$. Nell'immagine destra è invece riportata la risposta del segnale della telecamera (lo stesso mostrato in figura 6.6(a)). La risposta è abbastanza buona sui lati in pendenza, in cui vengono appianate le piccole oscillazioni, mentre le parti piatte e quelle in prossimità dei cambi di pendenza più marcati presentano delle oscillazioni sovraelongate.

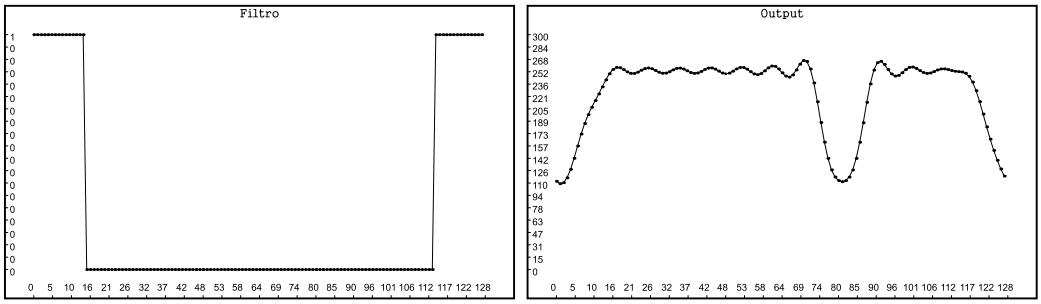


Figura 6.7: Filtro passa basso ideale e relativa risposta al filtraggio

Per quanto riguarda la struttura generale di un filtro, si deve anche tenere presente una proprietà importante delle trasformate, e cioè la cosiddetta "simmetria Hermitiana". Questa stabilisce che se $x(t)$ è un segnale a tempo continuo, la relativa Trasformata di Fourier $X(f)$ gode delle seguenti caratteristiche:

- la parte reale $X_{RE}(f)$ è una funzione pari
- la parte immaginaria $X_{IM}(f)$ è una funzione dispari
- il modulo $\left|X(f)\right|$ è una funzione pari

- l'argomento $\angle X(f)$ è una funzione dispari

Per quanto riguarda la DFT, il discorso è sempre valido, alla luce delle relazioni che la legano alla Trasformata di Fourier. In questo caso, assumendo che essa rappresenti un singolo periodo della Trasformata di Fourier di un segnale campionato e ripetuto, si capisce facilmente che se la sequenza di partenza è a valori reali, la simmetria nella DFT sarà relativa al centro della sequenza.

Nel caso di una sequenza a 128 valori (come nel caso delle telecamere) è quindi opportuno che i valori siano simmetrici rispetto al 65° valore; questo giustifica la simmetria utilizzata nella definzione del passa-basso ideale: così facendo, la IDFT del prodotto tra il filtro e la DFT del segnale della telecamera è ancora una sequenza a valori reali.

Per tentare di attenuare le oscillazioni, si è dunque pensato di utilizzare un filtro che avesse una forma meno "spigolosa", e cioè in cui la transizione da 1 a 0 e viceversa non avvenisse in un unico salto. Per fare ciò, si è provato ad applicare un filtro di tipo gaussiano al passa-basso ideale. Anche in questo caso, si sono dovuti variare i valori di `n` e `var` procedendo per tentativi fino a che non si è ottenuta una risposta soddisfacente. Inoltre, tutti gli elementi del filtro ottenuto sono stato divisi per il valore del primo elemento. In tale maniera, si è mantenuta la forma del filtro invariata (avendo scalato i valori di uno stesso fattore) ma si è anche assicurato che il segnale di uscita conservasse il valore medio, avendo infatti imposto in questa maniera che $F(0) = 1$ (cfr. paragrafo 6.5.1).

A titolo di esempio, in figura 6.8 si riportano due filtri – assieme alla risposta dell'immagine al relativo filtraggio – ottenuti con parametri di diverso valore. Il segnale di uscita di figura 6.8(a) presenta ancora una lieve oscillazione in corrispondenza degli "spigoli" del segnale della telecamera, con un punto di massimo e uno di minimo relativo in ciascuna oscillazione. Per questo filtro si sono usati i valori `n=13` e `var=2`.

Il filtro applicato in figura 6.8(b) è stato ottenuto invece utilizzando `n=21` e `var=5`. Il segnale di uscita in questo caso presenta una sovraelongazione minore rispetto al caso precedente, e inoltre la parte centrale è leggermente più piatta (e quindi più simile all'immagine originale). Dopo tutta una serie di prove, questo è il filtro che si è deciso di utilizzare per l'eleaborazione dei dati della telecamera.

Per concludere questa sezione, si riporta un'ultima immagine (6.9). In questo caso si è voluta testare la capacità del filtro scelto di ridurre il rumore presente su un'immagine. L'immagine a sinistra mostra una sequenza i dati a cui è stato sovrapposto del rumore con la funzione `addNoise`. In questo caso il rumore aggiunto può assumere, con la medesima probabilità, diversi valori nel range $[-10; +10]$. Tali valori sono volutamente molto più elevati del rumore che solitamente si è trovato presente sulle immagini. Questo in quanto si voleva testare il filtro in un caso estremo, che comunque probabilmente non si verificherà mai.

I risultati ottenuti sembrano soddisfacenti, e danno un'ulteriore conferma in merito alla bontà del filtro costruito.

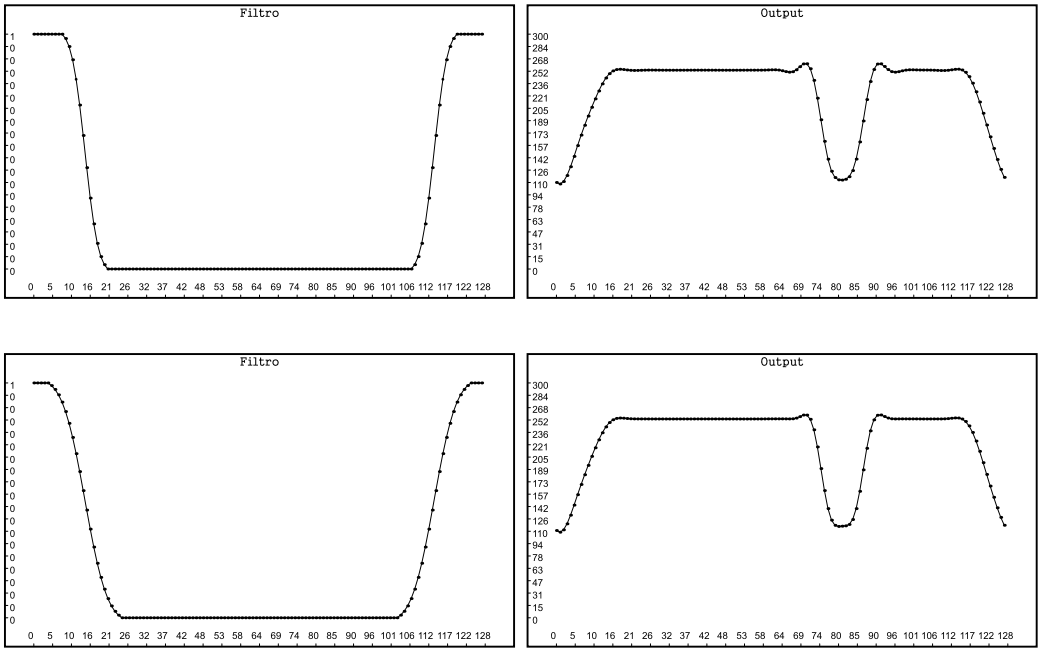


Figura 6.8: Esempi di filtri passa-basso “smussati”

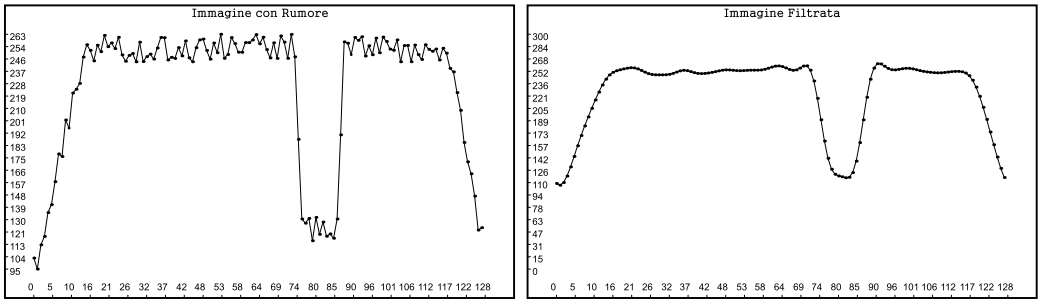


Figura 6.9: Filtraggio di un segnale affetto da rumore

6.7.3 Algortimo di Auto-taratura delle Telecamere

Le considerazioni fatte fin ora hanno permesso, alla fine, di elaborare una procedura di auto-taratura del tempo di integrazione delle telecamere, e che permette, in linea di principio, di individuare anche la soglia da utilizzare per l'algoritmo di riconoscimento delle linee nere sul tracciato.

Tale procedura si può applicare prima che la macchina debba iniziare il percorso, e si basa su letture successive di un'immagine a fondo bianco con una linea nera centrale (in quanto, come già detto, in questo tipo di immagine il tracciato è interamente noto).

Il grosso dell'algoritmo si basa su una funzione di costo, che permette di valutare se un'immagine è di qualità buona o meno. Per poter implementare tale funzione, si sono definiti i parametri fondamentali che caratterizzano una traccia, detta d_k:

- f_k, corrisponde alla traccia filtrata
- D_M, la media integrale del segnale
- P_{SX}, definito come quel valore tale per cui $d_k < D_M \ \forall k < P_{SX}$; questo quantifica il numero di pixel che, a causa dell'effetto bordo, hanno valore inferiore a D_M
- P_{DX}, definito come il valore per cui $d_k > D_M \ \forall k > P_{DX}$ (ha un significato analogo a P_{SX})
- $\widetilde{d_k}$ una versione idealizzata della traccia (verrà spiegato più avanti come ottenerla)
- $e_k = f_k - \widetilde{d_k}$ (differenza tra traccia filtrata e ideale)
- C_d, il "costo" della traccia (anche questo verrà definito dopo)

Il primo passo consiste nell'ottenere f_k filtrando la traccia d_k. Per farlo, si utilizza il filtro descritto nel precedente paragrafo. In seguito si può valutare D_M (che sarà rimasto invariato anche dopo il filtraggio).

Adesso quello che deve essere fatto è l'estrapolazione della sequenza $\widetilde{d_k}$. Se si considera il caso ideale, i pixel che puntano la linea nera dovrebbero riportare come valore 0, mentre quelli che puntano la linea bianca dovrebbero riportare 255 (il valore di saturazione). Per fare ciò si può valutare f_k in ogni punto. Di base, ciò che si fa è definire $\widetilde{d_k}$ come:

$$\widetilde{d_k} = \begin{cases} 0 & \text{se } f_k < D_M \\ 255 & \text{se } f_k \geq D_M \end{cases}$$

In realtà è necessario un accorgimento: potrebbe succedere che durante la salita vi sia un pixel avente valore leggermente superiore al successivo. Se per sfortuna dovessero essere tali per cui $f_k > D_M$ e $f_{k+1} < D_M$, si avrebbe in tale zona un'inversione multipla della traccia (lo stesso accade nelle zone in discesa). Per evitare

questa evenienza, si può applicare a f_k la funzione `hysteresisFilter` della classe `SignalFunctions`. Si è notato che, grazie al filtraggio già operato, basta utilizzare come soglia il valore 2 per evitare che una simile eventualità accada.

Una volta definita $\widetilde{d_k}$, si può procedere con il calcolo della sequenza e_k, e quindi di P_{SX}e P_{DX}.

L'ultimo parte della funzione consiste quindi nel calcolo del costo C_d. Questo passo fondamentale permette di assegnare a ogni immagine acquisita un valore che ha come scopo quello di quantificarne la bontà. L'idea di base, è quella di valutare quanto la traccia reale si discosta da quella ideale. Per farlo, si valuta quindi l'errore quadratico medio, definito come:

$$\sigma_d = \sqrt{\frac{1}{N}\sum_{k=0}^{N-1} e_k^2}$$

Maggiore sarà il valore di σ_d, maggiore sarà lo scostamento della traccia reale da quella ideale. Per avere un valore basso di σ_d, è dunque necessario che i valori alti della traccia siano prossimi a 255, mentre quelli bassi siano prossimi a 0. Per una traccia avente tempo di integrazione basso, i valori saranno tutti molto inferiori a 255, e quindi σ_d sarà alto. Similmente accadrà per una traccia in cui si abbia un tempo di integrazione troppo alto.

Ovviamente una caratteristica fondamentale di una buona immagine è anche quella di avere un numero molto ridotto di pixel alterati dall'effetto bordo. Per quantificare questo effetto, si ricorre dunque a P_{SX} e P_{DX}. Tanto più piccoli saranno questi valori, tanto migliore sarà la qualità dell'immagine.

L'algoritmo di calcolo del costo si conclude quindi valutando C_d come combinazione lineare di σ_d, P_{SX} e P_{DX}. In linea di principio, si può pensare di attribuire un peso unitario al primo termine, e di dare un medesimo peso λ ai rimanenti due. L'espressione del costo si può dunque scrivere come:

$$C_d = \sigma_d + \lambda\ (P_{SX} + P_{DX})$$

Per la valutazione di λ, si sono prese coppie di immagini abbastanza simili, di cui una sembrava leggermente migliore dell'altra. Si è quindi modificato via via il valore del peso fino a quando non è risultato che il costo complessivo dell'immagine migliore fosse minore di quello dell'altra immagine, ottenendo così $\lambda \approx 0.2$. Ovviamente questa procedura potrebbe essere migliorata, ma non si è indagato oltre in quanto i risultati ottenuti con tale valore sembrano già essere soddisfacenti.

L'algoritmo di calcolo del costo ha mostrato che effettivamente, dato un insieme di acquisizioni, quella con il costo più basso costituisce una buona acquisizione.

Si è quindi proceduto implementando le funzioni necessarie anche in codice C. In questo modo, si è potuto sviluppare l'algoritmo di auto-taratura, riportato in codice C nel listato 6.15. Per il funzionamento di tale algoritmo, è necesario indicare a priori un tempo di integrazione minimo di partenza `TEMPO_MINIMO_PARTENZA`, un tempo di integrazione massimo di partenza (`TEMPO_MINIMO_PARTENZA`) e una tolle-

ranza TOLLERANZA. Questa serve a indicare quanto precisamente si vuole trovare il tempo di integrazione.

Listato 6.15: Algoritmo di Auto-taratura della telecamera sinistra

```
unsigned int TEMPO_MINIMO_PARTENZA=130;
unsigned int TEMPO_MASSIMO_PARTENZA=10000;
unsigned int TOLLERANZA=20;
unsigned int t_integrazione=0;

unsigned int t_min=TEMPO_MINIMO_PARTENZA;
unsigned int t_max=TEMPO_MASSIMO_PARTENZA;
unsigned double costi[9] = {0};
unsigned int tempi[11] = {0};

while(t_max-t_min>TOLLERANZA) {
  tempi[0] = t_min;
  tempi[10] = t_max;
  unsigned int i;
  for(int i=0; i<9; i++)
    unsigned int t = t_min+(i+1)*t_max/10;
    tempi[i] = t;
    setTempoIntegrazione(t);
    delay(10);
    unsigned int j,costo=0;
    for(j=0; j<5; j++) {
      unsigned char data[128] = {0};
      while(getBufferSXrequest()!=CAM_BUFFER_REQUEST_READY);
      makeBufferSXRequest();
      while(getBufferSXrequest()!=CAM_BUFFER_REQUEST_REQUEST);
      C += valutaCosto(camBufferSX);
      clearBufferSXrequest();
    }
    costi[i]=C/5.0;
  }
  unsigned int index = indexOfMin(costi);
  t_integrazione = costi[index];
  t_min = index;
  t_max = index+2;
}
setTempoIntegrazione(t_integrazione);
delay(1000);
taraSoglia();
```

Per come è strutturato, l'algoritmo porta a valutare i costi di alcune acquisizioni impostando i tempi di integrazione in un intervallo ampio. Ogni volta, valuta i costi delle tracce a diversi tempi di integrazione, acquisendo per ciascuno più tracce in modo da eliminare possibili effetti rumorosi. Infine, valuta tra quei tempi quale è quello che comporta il costo minore, e restringe l'intervallo in modo da focalizzarsi attorno a quel tempo scelto.

Iterando tale procedura, si riesce a perfezionare di volta in volta la scelta del tempo

di integrazione, fino a quando l'intervallo di valutazione non raggiunge un'ampiezza predefinita. Una volta conclusa la procedura, la variabile `t_integrazione` contiene il tempo di integrazione richiesto. Questo viene impostato tramite `setTempoIntegrazione`, e viene dunque chiamata la funzione `taraSoglia`, che acquisisce un certo numero di immagini, ne valuta la media integrale, e quindi imposta come soglia il valore medio di tali quantità.

Il motivo per cui si è scelto di impostare come soglia il valore della media integrale, trova spiegazione nella semplice considerazione che tale valore permette di individuare in maniera corretta una linea nera al centro della pista senza analizzare per intero l'array. La ricerca parte da un estremo dei dati, in cui si troveranno prima pixel bianchi, e, non appena i pixel diventano neri, si potrà affermare di aver trovato la linea.

Nel caso delle due linee laterali, si avrà che l'immagine acquisita corrisponde alla precedente per la metà che punta alla pista. Pertanto, partendo da quel'estremo, si potrà individuare la linea senza dover mai andare ad analizzare i dati relativi al "fuori pista" esattamente come se la traccia fosse stata acquisita nel caso di fondo interamente bianco e linea nera centrale, legittimando l'uso del valore medio calcolato in fase di setup come soglia adeguata.

Capitolo 7

Conclusioni

Con questo progetto si sono riusciti a risolvere, in buona parte, tutti quei problemi legati allo stadio di acquisizione che rendevano dispendioso, in termini di tempo, il processo di sviluppo e collaudo degli algoritmi di controllo.

La possibilità di monitorare, tramite il display LCD, l'esecuzione dei programmi scritti, permette da un lato di individuare gli errori commessi in maniera veloce e semplice, e dall'altro di tarare parametri interni senza la necessità di riprogrammare ogni volta il microcontrollore.

L'hardware e soprattutto il software progettati costituiscono poi uno strumento potente per lo sviluppo di algoritmi di riconoscimento del tracciato, potendo seguire in maniera agevole l'evoluzione temporale delle immagini acquisite.

Ancora, la presenza di una memoria permette non solo di acquisire i dati ai fini della simulazione degli algoritmi di elaborazione delle immagini, ma permette di creare algoritmi complessi che possono basarsi anche su informazioni precedenti, che non vengono perse quando si spegne il dispositivo.

Tutte queste questioni vengono lasciate aperte, come spunto di lavoro per coloro che si cimenteranno in una competizione simile.

"Penso di fermarmi qui" - A.Wiles

Ringraziamenti

Tre anni di vita sono molti, quando ne hai ventuno. Sono un settimo di ciò che hai vissuto, e quindi sono un settimo di te stesso.

Non posso quindi sentirmi di esser davvero giunto alla fine, se prima non ho modo di ringraziare tutti coloro che hanno giocato una ruolo essenziale in questo periodo:

Carlo, Olga, Mariacarla, Sara e Alessandro: il vostro supporto costante mi ha spinto a non mollare mai.

Maria: mi hai accolto a braccia aperte in una città sconosciuta, mi hai consolato nei momenti difficili ed eri sempre presente in quelli felici. Per quanto possa provarci, non ci sono parole per ringraziarti.

Andrea P. P. Pretto ed Emanuele B. C. Pareglio: avete saputo sopportare la convivenza con me (o forse dovrei dire: avete saputo sopravvivere ad essa), e siete stati la mia famiglia in quest'anno. Non avrei potuto desiderare due coinquilini migliori. Grazie, davvero. E attenti alla calza.

Nairi: non pensavo che, giunto quasi alla fine, avrei potuto trovare una persona come te. Sono almeno 45 minuti che continuo a scrivere frasi e a cancellarle, perché qualunque parola scriva non riesce ad essere all'altezza di ciò che tu rappresenti per me. E alla fine, forse le parole più semplici sono anche le migliori: ti amo.

Mattia e Fabio: compagni di (s)ventura al DTG, mi avete supportato in una lunga lotta contro strani mostri dell'Ingegneria, tra foreste di Trasformate e pericolanti travi a sbalzo.

Gian, Gio e Giugi: non posso non ringraziarvi separatamente. Senza di voi, non ci sarebbero state le Pannocchie. Senza di voi, non avrei avuto un posto dove stare durante una sessione di esami. Senza di voi, Vicenza non sarebbe stata la stessa per me.

I Fuorisede delle Pannocchie: Vicenza non è una città universitaria, si sa, ma, con voi, per me lo è stata sicuramente. Grazie per tutte le feste, le grigliate e gli aperitivi. Insomma, grazie per tutte le esperienze passate insieme. Non le dimenticherò mai.

Ale, il Biondo e Genni: i primi amici universitari incontrati, avete reso spassose le lezioni dei primi tre semestri.

Marco: inutile girarci attorno, senza il tuo aiuto non avrei potuto dare metà degli esami.

Gli scout di Vicenza: benedico quell'incontro casuale con Chelo: questo mi ha

permesso di conoscere lei, Giulia e Marcello, che ringrazio per avermi accolto in questa piccola, grande famiglia vicentina. Ringrazio poi Riccardo, che mi ha portato a cercare sempre domande e mai soluzioni. Ringrazio Agnese, che era sempre presente e pronta a darmi una mano. Ringrazio Diletta, Antonio e Paolo che come mi hanno accompagnato in questo anno per me difficile e pieno di insicurezze. Ringrazio i miei lupetti, che mi hanno dimostrato quanto sia vera la parola maestra *"La forza del Lupo è nel branco"*. Ancora, devo ringraziare alcuni "lupettari" con cui più degli altri ho legato: Anna, Giulia, Stefania, Riccardo e Chiara. Un fratello e amico scout mi ha detto, mentre salutavo gli staff di branco al campo, *"Io forse non sarò un lupettaro, ma per me sei un grande amico"*: grazie, Abe. Infine, un grazie va a tutti gli scout che ho incontrato in questi tre anni. A voi dedico il quarto punto della nostra Legge. È proprio grazie a questa che vi ho consociuto.

Voglio ringraziare infine Roberto Oboe e Paolo Magnone, due professori che mi hanno saputo indicare, forse a loro insaputa, la direzione giusta verso il mio futuro.

Aver vissuto a Vicenza, cosìdiversa dalla mia città natale, per un settimo della mia vita, mi ha cambiato profondamente, sotto una moltitudine di aspetti. E perciò, giunto alla fine di questo lungo tempo, non posso non voltarmi a guardare indietro, ricordando con un misto di gioia e tristezza tutto ciò che ho vissuto.

Ma la cosa importante, è che nel momento in cui mi volto, quello che vedo sono i volti delle persone che ho incontrato. Se chiudo gli occhi, le immagini si susseguono come una serie di scatti congelati. Quando penso a un posto in cui sono stato, non vedo semplicemente una piazza, una strada o un prato. Vedo una piazza gremita di amici per il rito dell'aperitivo. Vedo una strada percorsa da volti noti. Vedo un prato in cui si rotolano i miei lupetti. E, per quanto possa essere un giochetto del mio cervello, giuro che non riesco a vedere una sola scena triste. Vedo le facce sorridenti di voi, che mi avete accompagnato in questo cammino. Vedo le facce sorridenti di voi, che siete un settimo di me.

Io mi dico è stato meglio lasciarci,
che non esserci mai incontrati

Bibliografia

[1] Processing language reference. `https://processing.org/reference`.

[2] TAOS (AMS). *TSL1401CL - 128x1 Linear Sensor Array with Hold*, 2011.

[3] Matteo Bertocco and Alessandro Sona. *Introduzione alle Misure Elettroniche*. Lulu, 2nd edition, 2013.

[4] Cay Horstmann. *Concetti di Informatica e Fondamenti di Java*. Apogeo, 5th edition, 2010.

[5] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[6] Dietel and Dietel. *Programmazione Java - Tecniche Avanzate*. Pearson, 7th edition, 2008.

[7] Freescale Semiconductors. FRDM-KL25Z pin usage and pinout chart.

[8] Freescale Semiconductors. *KL25 Sub-Family Reference Manual (rev.3)*, September 2012. Document number: KL25P80M48SF0RM.

[9] Freescale Semiconductors. FRDM-KL25Z schematics (rev.E), January 2013. Document number: SCH-27556.

[10] Freescale Semiconductors. *FRDM-KL25Z User's Manual (rev.2)*, October 2013.

[11] Freescale Semiconductors. *Kinetis KL25 Sub-Family Datasheet (rev.5)*, August 2014. Document number: KL25P80M48SF0.

[12] Giuseppe Ricci and Maria Elena Valcher. *Segnali e Sistemi*. Libreria Progetto, 4th edition, 2010.

[13] Hitachi. *HD44780U - Dot Matrix Liquid Crystal Display Controller/Driver*, 1998.

[14] Giordano Lilli. *Hardware e Software di Controllo del Veicolo Freescale Cup*. Dipartimento di Tecnica e Gestione dei Sistemi Industriali (DTG) - Padova, 2013.

[15] Francesco Dal Santo. *Trasmissione dati Bluetooth tra PC e microcontrollore per un veicolo autonomo*. Dipartimento di Tecnica e Gestione dei Sistemi Industriali (DTG) - Padova, 2014.

[16] Winbond. *Winbond W25Q80BV Serial Flash Memory (rev.1)*, October 2013.

[17] Winstar. *WH1602B Winstar Professional LCD Module Manufacture*.

Elenco delle figure

Elenco delle tabelle

Elenco dei listati