UNIVERSITA' DEGLI STUDI DI ROMA

TOR VERGATA

FACOLTA' DI INGEGNERIA

Tesi di Laurea Specialistica in Ingegneria Informatica

Sviluppo di un driver per un controllore digitale di movimento

Relatore

Prof. Marco Cesati

Laureando

Emiliano Betti

Anno accademico 2004-2005

"Vivi semplicemente,

affinché gli altri possano semplicemente vivere."

Mohandas Karamchand Gandhi

Indice

Indice iv

Elenco delle figure v

Elenco delle tabelle vi

Introduzione 1

Driver in Linux 2

Moduli del kernel 2

Caratteristiche di un driver 3

Buffer 4

Galil DMC 1800 5

1 Schede PCI Galil DMC 1800 6

1.1 Porte e memoria di I/O 6

1.2 Interruzioni 8

1.3 Bus PCI 10

1.3.1 Indirizzamento di una periferica 10

1.3.2 Comunicazione 12

1.3.3 Spazio di configurazione 13

1.4 Galil DMC 1800 16

1.4.1 Buffer FIFO ed interruzioni 18

1.4.2 Interfaccia di comunicazione 21

1.4.3 Spazio di configurazione PCI 22

1.4.4 Le porte di I/O 23

2 Kernel Linux 2.6 26

2.1 Creare un modulo del kernel 26

2.1.1 Struttura di un modulo 27

2.1.2 La compilazione 30

2.1.3 Il caricamento 30

2.2 Modi di esecuzione dei processi 31

2.3 Strumenti del kernel utilizzati nel driver 32

2.3.1 Inizializzazione del bus PCI 32

2.3.2 Gestione della memoria 37

2.3.3 Gestione della concorrenza 39

2.3.4 Processi differiti 44

2.4 Approfondimenti 50

3 Organizzazione del driver 51

3.1 Informazioni generali e licenza 51

3.2 Caratteristiche principali 52

3.3 Comunicazione con lo spazio utente 53

3.3.1 I device file 54

3.3.2 I device file per le schede Galil DMC 1800 56

3.4 Il codice sorgente 58

3.5 Le strutture dati principali 58

3.6 Inizializzazione 64

3.6.1 Opzioni di caricamento del modulo 65

3.6.2 Registrazione PCI 66

3.6.3 Inizializzazione dell'hardware 70

3.6.4 Registrazione dei device file a caratteri 70

3.6.5 *sysfs* filesystem 73

3.6.6 Completamento dell'inizializzazione durante l'apertura del device file 74

4 Comunicazione con l'hardware 77
4.1 Strutture dati coinvolte 77
4.1.1 Utilizzo dello spinlock 79
4.2 Operazioni di controllo 80
4.2.1 Interruzioni 81
4.2.2 Reset 82
4.3 Accesso alle FIFO primarie 83
4.3.1 Lettura 84
4.3.2 Scrittura 87
4.4 Lettura della FIFO secondaria 90

5 Strategie di memorizzazione dei comandi 94
5.1 Introduzione al device file card 94
5.1.1 Opzioni 95
5.2 Apertura e chiusura del device file 98
5.3 Comandi della chiamata di sistema `ioctl` 103
5.4 Lettura 105
5.5 Scrittura non bufferizzata 107
5.6 Il buffer dei comandi 109
5.6.1 Progettazione 112
5.6.2 Descrizione 114
5.6.3 Implementazione 117
5.7 Scrittura bufferizzata 128

6 Eventi speciali ed interruzioni 131
6.1 Introduzione al device file event 131
6.1.1 Prestazioni 134
6.1.2 Opzioni 134
6.2 Apertura e chiusura del device file 135
6.3 Il buffer degli eventi 136
6.3.1 Progettazione 136
6.3.2 Implementazione 139

6.4 Lettura degli eventi 149
6.4.1 File operation `poll` 152
6.5 Comandi della chiamata di sistema `ioctl` 153
6.6 Gestione delle interruzioni 154
6.6.1 Inizializzazione 155
6.6.2 Stack delle interruzioni 156
6.6.3 Top half handler 158
6.6.4 Bottom half handler 160

7 Lettura della FIFO secondaria 162
7.1 Introduzione al device file info 162
7.1.1 Opzioni 163
7.2 Apertura e chiusura del device file 164
7.3 Lettura della FIFO secondaria 165

8 Test 167

Conclusioni 170
Sviluppi futuri 172

Riferimenti bibliografici 173

Ringraziamenti 176

Elenco delle figure

1.1 Diagramma a blocchi di un sistema con bus PCI 11
1.2 Bit dell'indirizzo di una periferica PCI 11
1.3 Intestazione dello spazio di configurazione di un dispositivo PCI 13
1.4 Scheda Galil DMC 1840 17
1.5 Schema semplificato dell'interfaccia verso il bus PCI delle schede Galil DMC 1800 18
1.6 Porte e memoria di I/O delle schede Galil DMC 1800 21

3.1 Interazione tra i registri delle schede Galil DMC 1800 ed i device file creati dal driver 57

5.1 Schema rappresentativo del buffer dei comandi 117

6.1 Esempio di operazione di lettura nel buffer degli eventi 139

Elenco delle tabelle

1.1 Tipi di intestazioni per lo spazio di configurazione PCI 14
1.2 Eventi che le schede Galil DMC 1800 sono in grado di segnalare attraverso interruzioni 20
1.3 Campi per l'indentificazione della funzione nello spazio di configurazione delle schede Galil DMC 1800 22
1.4 Significato dei bit del registro di controllo per le schede Galil DMC 1800 24
1.5 Valori possibili per il registro di reset per i DMC 1800 25

2.1 Corrispondenza fra i campi della struttura *pci_device_id* e lo spazio di configurazione PCI 34

3.1 Elenco dei file sorgenti 58
3.2 Elenco delle strutture dati definite nel driver 59

4.1 Inizializzazione degli indirizzi per l'accesso alle porte di I/O . 78

5.1 Opzioni del device file card 96
5.2 Lista dei comandi per la funzione `ioctl` del device file card. . 103

6.1 Possibili tipi di eventi riportati dal device file event 141

Introduzione

Il lavoro che sarà descritto in questo testo si potrebbe definire brevemente come la realizzazione di un'interfaccia fra una scheda dedicata al controllo di movimento robot ed i programmi che utilizzeranno tale sistema: questo tipo di interfaccia (cioè fra hardware e sistema operativo), a causa della sua funzione di controllore, è comunemente chiamata *driver*. A tal fine è stata quindi realizzata un'estensione del kernel che aggiunge a quest'ultimo le funzionalità e le informazioni necessarie a gestire tale scheda. In modo particolare il kernel scelto è quello del sistema operativo *Linux*[1] nella sua versione 2.6.15, mentre la scheda per il controllo di movimento robot è la *Galil DMC 1860*, gentilmente fornita dalla *Galil Motion Control, Inc.* al Dipartimento di Sistemi e Produzione dell'Università di Roma Tor Vergata.

Si vedrà in seguito che il driver sviluppato non si limita solo a fornire le caratteristiche di base per il funzionamento della scheda, ovvero il cosidetto *driver di basso livello*. La parte principale del lavoro è stata invece la costruzione di un modello di funzionamento orientato a risolvere problemi tipici dei controllori digitali di movimento relativi alle prestazioni ed all'affidabilità ; a questo scopo è stato sviluppato un complesso *driver di alto livello* che, oltre ad estendere le funzionalità della scheda, introduce alcuni buffer[2] il cui scopo è proprio quello di migliorare l'affidabilità e le prestazioni.

[1] Linux è un marchio registrato da Linus Torvalds.

[2] Con questo termine si intendono zone di memoria dedicate ad accumulare dati temporanei.

Driver in Linux

Linux nasce nel 1991 per opera di Linus Torvalds: è ispirato a *Unix* ed è un sistema operativo *opensource*, cioè che mette a disposizione di tutti i propri file sorgenti[3]: l'ultima versione del suo kernel è costituito da circa 6 milioni di righe di codice scritte in linguaggio C ed in linguaggio Assembler. Il capitolo 2 ne discuterà alcune parti che riguardano strettamente lo sviluppo di un driver; di seguito, al fine di fornire una visione generale del lavoro svolto, sarà introdotta una sua caratteristica particolarmente importante.

Moduli del kernel

Il kernel di Linux ha il supporto ad oggetti chiamati *moduli*: ciò permette lo sviluppo di intere sue parti che implementano particolari funzioni come blocchi separati (definiti appunto *moduli*), compilati a parte e poi successivamente al caricamento in memoria del kernel (quindi a sistema funzionante) collegati a quest'ultimo. I vantaggi di questo tipo di approccio sono tali che ormai praticamente tutti i driver sono sviluppati come moduli. In particolare:

- Il kernel di Linux supporta la maggior parte dell'hardware in commercio, quindi compilare tutti i relativi driver ed inserirli staticamente nel kernel produrrebbe un'immagine di quest'ultimo con dimensioni inutilmente grandi. Quello che si fa quindi è compilare semplicemente i driver necessari al sistema in uso.

- Non è detto poi che tutti i driver compilati debbano necessariamente essere sempre caricati in memoria: infatti, se il driver è stato compilato come modulo, è possibile collegarlo al kernel a *runtime*, ovvero a sistema avviato. Questo, nei moderni sistemi dotati di molta RAM, potrebbe sembrare un vantaggio di poco conto, ma in realtà non è così: ogni qualvolta viene caricato un modulo tipicamente la periferica

[3]Tali file sono rilasciati sotto la *GNU General Public Licence (GPL) [21]*.

viene riavviata e questo permette di installare un driver senza riavviare la macchina (come è necessario invece in alcuni sistemi operativi). Inoltre la possibilità di rimuovere dalla memoria moduli precedentemente caricati e poi eventualmente ricaricarli permette facilmente di riavviare soltanto il driver (e quindi anche la periferica), in caso di malfunzionamenti.

- Creare driver sotto forma di moduli semplifica il lavoro dello sviluppatore: infatti un modulo ha una struttura ben definita che gli permette di interfacciarsi con il kernel: questa struttura serve anche come linea guida per il programmatore.

- Un modulo può non appartenere direttamente ai sorgenti ufficiali del kernel, ma può essere sviluppato e compilato separatamente, permettendo quindi a tutti di poter realizzare il proprio driver senza modificare altri file del kernel.

A questo punto, in considerazione di quanto è stato detto finora, si può dare una più precisa definizione del lavoro svolto in questa tesi: la progettazione e lo sviluppo di un modulo per il kernel di Linux 2.6.15 che svolge la funzione di driver per una particolare serie di schede dedicate al controllo digitale di movimento.

Caratteristiche di un driver

Linux fornisce un'astrazione di gran parte parte dell'hardware sotto forma di speciali file: i *device file*. Ad esempio, per comunicare con una qualsiasi periferica un programmatore può utilizzare praticamente le stesse chiamate di sistema che utilizzerebbe su un file di testo. Vediamo alcuni esempi di chiamate di sistema applicate ad un device file:

- `open()`: apertura del file ed inizializzazione del dispositivo

- `close()`: chiusura del file e rilascio delle risorse

- `read()`: lettura di dati dal dispositivo
- `write()`: scrittura di dati sul dispositivo

Usare la funzione `read()` su un device file equivale quindi a leggere dati dal dispositivo, mentre, come è noto, la stessa funzione applicata ad un file di testo ne restituisce il contenuto letto attraverso il file system. Sebbene le due operazioni siano apparentemente molto diverse, Linux nasconde le differenze alle applicazioni utente. Ciò è realizzato suddividendo tutte le chiamate di sistema che operano sui file in due parti principali: una comune a tutti ed una seconda specifica per ogni file. Per i file regolari questa seconda parte dipende dal particolare file system al quale appartengono (ad esempio *ext2* o *fat*), mentre per i device file deve essere implementata dal driver che controlla la periferica.

Quello appena descritto è il principale compito di un driver. In ogni caso non è sempre necessario implementare la parte di basso livello di tutte le chiamate di sistema operanti su file: la maggior parte dei driver infatti ne implementa solo un sottogruppo che dipende dalle funzioni della particolare periferica che si deve controllare.

Ovviamente il driver si occupa anche di inizializzare il dispositivo, richiedere al kernel le risorse necessarie, gestire le interruzioni ed altre eventuali caratteristiche proprie del dispositivo che si intende controllare.

Buffer

È molto comune che un driver, per migliorare le prestazione dell'hardware da controllare, introduca dei buffer software, ovvero delle zone di memoria del kernel dedicate ad accumulare dati in transito da e verso il dispositivo controllato. Il driver sviluppato in questa tesi introduce due importanti buffer che modificano in maniera evidente il funzionamento della scheda: un buffer di scrittura (per i comandi) ed un buffer degli eventi speciali (interruzioni ed altro). Questi buffer saranno descritti rispettivamente nei capitoli 5 e 6.

Galil DMC 1800

Il software sviluppato in questa tesi controlla una serie di schede per bus PCI[4], in particolare le *Galil Digital Motion Controller serie Optima 1800*: le schede sono dei controllori di movimento progettati per applicazioni da 1 ad 8 assi (in base al modello) [24]. Il driver funziona con ogni scheda della serie 1800, ma in particolare quella usata per lo sviluppo ed i test è la Galil DMC 1860, che può controllare fino a 6 assi.

Queste schede trovano applicazione nella robotica industriale, ad esempio per il controllo di una macchina per fresare il metallo. Incorporano un microcomputer Motorola a 32 bit con memoria locale [23] e forniscono la possibilità di eseguire parallelamente fino ad 8 processi, programmati attraverso un semplice linguaggio appositamente sviluppato dalla Galil [25]. Il capitolo 1 ne descriverà i dettagli.

[4]Peripheral Component Interconnect [26]

Capitolo 1

Schede PCI Galil DMC 1800

Questo capitolo farà un'introduzione ad alcuni degli aspetti hardware trattati in questo lavoro di tesi: l'obiettivo finale è descrivere le caratteristiche delle schede Galil DMC 1800 e quindi la loro interfaccia verso il driver. A tale scopo saranno introdotti concetti come l'I/O di dispositivo, le interruzioni ed il bus PCI. Le descrizioni non saranno particolarmente dettagliate, ma vogliono fornire solo le informazioni necessarie a comprendere come un dispositivo hardware dialoga con il resto del sistema e con il software che lo controlla.

1.1 Porte e memoria di I/O

Tipicamente ogni periferica possiede un'*interfaccia di I/O* e dei registri di lettura e scrittura mediante i quali è possibile controllarla. L'interfaccia solitamente espone i registri in maniera tale che possano essere acceduti attraverso indirizzi contigui che appartengono o allo spazio di indirizzi della memoria oppure ad uno specifico spazio di indirizzi di I/O. Nel primo caso la memoria verrà considerata *memoria di I/O*, mentre nel secondo gli indirizzi dello spazio di I/O sono in genere chiamati *porte di I/O*.

Se i registri sono accessibili nello spazio di indirizzi della memoria allora per accedervi sono sufficienti normali istruzioni comuni a tutti i processori (`mov`, `and`, `or`, ...), mentre per accedere alle porte di I/O sono necessarie

speciali istruzioni in grado di tradurre l'indirizzo di I/O e scambiare i dati direttamente con la periferica. L'utilizzo della memoria di I/O è tipicamente preferito a quello delle porte (specialmente dai dispositivi PCI[1]); vediamo alcuni motivi:

- l'uso di specifiche istruzioni di CPU limita le architetture utilizzabili oppure obbliga ad operazioni di adattamento che possono produrre un degrado delle prestazioni,
- l'accesso alla memoria è in genere più efficiente ed in alcuni casi può essere usato il DMA (Direct Memory Access),
- il compilatore riesce ad ottimizzare meglio il codice.

La famiglia dei processori x86[2] fa parte di quei processori che riservano uno specifico spazio degli indirizzi (che in questo caso è a 16 bit) dedicato all'accesso alle periferiche e fornisce quindi speciali istruzioni per accedere alle porte di I/O. Ogni porta indirizza 8 bit e due o quattro porte consecutive possono essere accedute come un unico registro rispettivamente di 16 o 32 bit. Ci sono a disposizione 4 istruzioni fondamentali:

- `in` e `ins`: usate per la lettura; la versione "`s`" permette di leggere sequenze di byte consecutivi;
- `out` e `outs`: usate per la scrittura; la versione "`s`" permette di scrivere sequenze di byte consecutivi.

In genere i dispositivi tendono a fornire un'interfaccia di I/O comune dove sono presenti tipicamente 4 registri, assegnati a 4 porte di I/O, mediante le quali si riescono ad effettuare tutte le operazioni necessarie. Troviamo:

---

[1] Anche se il dispositivo oggetto di questa tesi in realtà utilizza le porte di I/O.

[2] Con il termine *x86* si fa riferimento alla famiglia dei processori compatibili con l'Intel 8086, un microprocessore progettato da Intel nel 1978. Alcuni esempi sono i processori Intel 386, 486 e tutta la serie Pentium, ma vi sono anche processori non Intel, come l'AMD Athlon e molti suoi predecessori.

- **un registro di stato**: qui è possibile leggere le informazioni sullo stato interno del dispositivo;
- **un registro di controllo**: qui è possibile modificare parametri di un dispositivo oppure inviargli comandi;
- **un registro di ingresso**: per leggere dati dal dispositivo;
- **un registro di uscita**: per scrivere dati nel dispositivo.

Ovviamente questo schema non è rigido, ma è semplicemente un modello diffuso che quindi semplifica la vita degli sviluppatori di driver: altre varianti comuni vedono alcuni registri utilizzati sia in lettura che in scrittura accoppiando ad esempio il registro di controllo con quello di stato e quello di ingresso con quello di uscita. In genere ogni altro schema è possibile.

1.2 Interruzioni

Per definizione un'*interruzione* è un segnale che altera la sequenza di istruzioni eseguite dal processore.

Le interruzioni possono essere generate dal processore stesso ed in questo caso sono di tipo *sincrono* (negli x86 vengono chiamate *eccezioni*), oppure possono partire da dispositivi esterni, risultando quindi *asincrone*: la differenza principale è che le prime, essendo generate internamente dal processore, si originano sempre al termine di un'istruzione, mentre quelle asincrone sono dette tali proprio perché, venendo dall'esterno, non è in nessun modo prevedibile l'istante in cui possono arrivare.

Un'altra classificazione può essere fatta fra interruzioni *mascherabili* e *non mascherabili*: un'interruzione mascherata viene ignorata dal processore finché resta tale, mentre quelle non mascherabili sono tipicamente eventi critici che devono essere sempre gestiti. Nell'architettura x86 le istruzioni `cli` e `sti` rispettivamente disabilitano e riabilitano tutte le interruzioni mascherabili. Tutte le richieste di interruzione (*IRQ, Interrupt ReQuest*) generate dalle

periferiche di I/O originano interruzioni asincrone mascherabili ed è quindi su queste che ci concentreremo nel resto del paragrafo.

Le periferiche in genere usano le interruzioni per avvisare il programma che le controlla (cioè il loro driver) che si è verificato un particolare evento, come ad esempio:

- la periferica è pronta per la lettura o la scrittura;
- la periferica ha completato un'operazione di lettura o scrittura.

In genere ogni dispositivo usa una sola linea di IRQ ed attraverso uno dei registri di I/O si può leggere lo stato della periferica e comprendere il significato dell'interruzione. Va detto che le linee di IRQ possono essere *condivise*[3], quindi diverse periferiche possono inviare un'interruzione con lo stesso numero. In questo caso la periferica deve essere predisposta e permettere al suo driver di capire se l'interruzione è partita da lei o meno. Come vedremo meglio nel capitolo 6 la gestione delle interruzioni può rappresentare uno degli aspetti più delicati nella realizzazione di un driver.

Le periferiche non possono inviare direttamente un'interruzione al processore, bensì le loro richieste di interruzione sono filtrate da un apposito dispositivo: il *Programmable Interrupt Controller (PIC).* Il PIC funziona da intermediario fra la CPU e l'interfaccia di I/O, già citata nel paragrafo precedente, controllando tutte le linee di IRQ a cui è connesso: ogni qualvolta viene emesso un segnale valuta se inoltrarlo o meno al processore. Un solo PIC può controllare un numero di linee di IRQ limitato quindi spesso troviamo più PIC connessi per aumentare le linee gestibili: in questo modello un solo PIC è connesso alla CPU, mentre altri possono essere concatenati su una delle sue linee di IRQ. Inoltre i PIC furono progettati per macchine uniprocessore e quindi per i moderni sistemi multiprocessore simmetrici (SMP) non sono adeguati: uno dei motivi è che connettere un PIC ad un solo processore diminuirebbe la "simmetria" propria del sistema SMP. È stata quindi

[3]Il numero di IRQ è limitato quindi questa è una caratteristica indispensabile nei moderni sistemi dotati di un gran numero di dispositivi.

introdotta una nuova struttura con due livelli di *APIC (Advanced PIC)*, uno locale ad ogni singolo processore ed un I/O APIC centrale che raccoglie tutti gli IRQ e li distribuisce fra le varie CPU. Per ulteriori approfondimenti si veda [2, pag.131–188].

In ultimo va detto che ogni linea di IRQ può essere singolarmente disabilitata: bisogna notare che questa operazione non agisce direttamente sul processore, come mascherare le interruzioni, ma sulla "richiesta di interruzione" e quindi la disabilitazione è gestita direttamente dal PIC (o APIC). Le interruzioni disabilitate non vengono perse, ma vengono inviate non appena la linea viene riabilitata.

1.3 Bus PCI

Attualmente il bus *PCI (Peripheral Component Interconnect [26])* è il più usato sistema di interconnessione dei componenti periferici, probabilmente anche grazie al fatto che è stato progettato in maniera indipendente dall'architettura e questo ne ha permesso l'adozione in molti sistemi diversi (x86, Alpha, PowerPC, SPARC64, ...). In questo paragrafo verrà descritta l'interfaccia PCI verso le periferiche al fine di capire come queste possono essere indirizzate, riconosciute e configurate.

È utile precisare che il bus PCI è stato concepito molti anni fa e nel tempo ha subito diversi aggiornamenti ed ottimizzazioni per stare al passo con le crescenti esigenze dei moderni sistemi. Le sue caratteristiche quindi (soprattutto quelle prestazionali) variano molto in base alla revisione disponibile, infatti può avere un canale dati di 32 o 64 bit e una frequenza di clock di 25, 33, 66 o 133 Mhz. La descrizione che segue cercherà di essere il più possibile generica.

1.3.1 Indirizzamento di una periferica

Lo standard PCI prevede che in ogni sistema ci possa essere più di un bus, ognuno contenente più periferiche PCI che a loro volta possono possedere

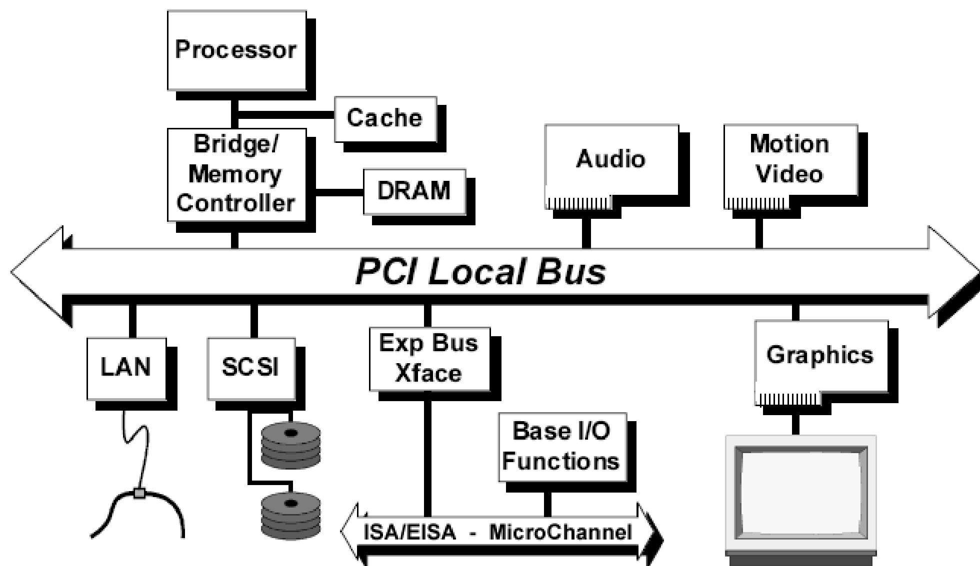

Figura 1.1: Diagramma a blocchi di un sistema con bus PCI (figura tratta da [26]

una o più *funzioni*; mentre il concetto di bus e di dispositivo è abbastanza ovvio, quello di “funzione” forse richiede di un esempio. Le cosidette “schede TV”, ovvero quei dispositivi in grado di catturare e visualizzare il segnale televiso su un computer, dovranno disporre come minimo di due funzioni: una di cattura video e l’altra di cattura audio. Il sistema operativo identifica quindi le due funzioni come oggetti separati che possono essere controllati in maniera indipendente uno dall’altro, anche se in realtà si trovano nello stesso dispositivo.

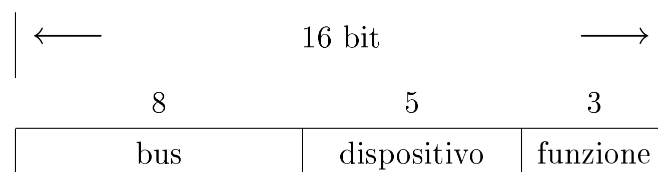

Figura 1.2: Bit dell’indirizzo di una periferica PCI

Ogni funzione può essere identificata da un indirizzo di 16 bit che viene

interpretato suddividendolo in tre blocchi, come mostrato in figura 1.2. I primi 8 bit sono letti come *numero di bus*: in ogni sistema è quindi possibile far coesistere fino a 2^8 (ovvero 256) bus.[4] Successivamente un blocco di 5 bit identifica il *numero del dispositivo* all'interno di un bus: possiamo avere massimo 2^5, cioè 32 dispositivi per bus. Infine, gli ultimi 3 bit servono ad associare un massimo di 8 *funzioni* ad ogni dispositivo.

1.3.2 Comunicazione

Una volta identificata una periferica PCI, esistono 3 spazi di indirizzi attraverso i quali si può raggiungere la periferica:

- spazio di I/O (attraverso le porte di I/O, vedi paragrafo 1.1);
- spazio di memoria;
- spazio (o registri) di configurazione.

Il canale di comunicazione del bus PCI usato per accedere ai primi due spazi di indirizzi è condiviso da tutte le periferiche collegate sullo stesso bus: questo determina la condivisione dei dati in transito e rende necessario un arbitraggio del suddetto canale per evitare collisioni. L'accesso ai registri di configurazione avviene invece attraverso un *indirizzamento geografico*, ovvero una periferica ala volta: in questo caso quindi non c'è condivisione e non è necessario alcun arbitraggio.

Ogni slot PCI ha 4 possibili linee di IRQ e ciascuna funzione usa una sola di queste linee: la scelta avviene in fase di configurazione, dove viene anche assegnato il numero di IRQ.

In un bus PCI lo spazio di I/O è acceduto mediante indirizzi fino a 32 bit, mentre per le locazioni di memoria sono possibili sia indirizzi a 32 bit, sia a 64 bit. Questi indirizzi vengono fissati durante la fase di inizializzazione della periferica da parte del *firmware* e quindi all'avvio del sistema oppure, nel

[4] In sistemi molto grandi questo può essere un limite eccessivo e Linux risolve il problema introducendo il concetto di *dominio* (vedi [1]).

caso di una periferica con supporto ad *hot plug*, durante il suo caricamento. Tali indirizzi vengono poi scritti nei registri di configurazione PCI e questo permette ad uno sviluppatore di driver semplicemente di leggerli li, evitando le spesso complesse procedure di ricerca delle risorse.

1.3.3 Spazio di configurazione

Ogni funzione di un dispositivo PCI dispone del proprio *spazio di configurazione*[5], ovvero alcuni registri presenti sul dispositivo che permettono, ad esempio, una facile identificazione della funzione e degli indirizzi delle risorse di cui quest'ultima dispone. Come già detto lo spazio di configurazione è acceduto attraverso apposite istruzioni e non usa né lo spazio di indirizzi della memoria, né le porte di I/O; tipicamente viene utilizzato durante la fase di boot, ma rimane comunque sempre accessibile.

<table>
<tr><th></th><th>0x0</th><th>0x1</th><th>0x2</th><th>0x3</th><th>0x4</th><th>0x5</th><th>0x6</th><th>0x7</th></tr>
<tr><td>0x00</td><td colspan="2">Vendor ID</td><td colspan="2">Device ID</td><td colspan="2">Command reg.</td><td colspan="2">Status reg.</td></tr>
<tr><td>0x08</td><td>RI</td><td colspan="3">Class Code</td><td>CL</td><td>LT</td><td>HT</td><td>BIST</td></tr>
<tr><td>0x10</td><td colspan="4">Base address 0</td><td colspan="4">Base address 1</td></tr>
<tr><td>0x18</td><td colspan="4">Base address 2</td><td colspan="4">Base address 3</td></tr>
<tr><td>0x20</td><td colspan="4">Base address 4</td><td colspan="4">Base address 5</td></tr>
<tr><td>0x28</td><td colspan="4">Card Bus CIS Pointer</td><td colspan="2">Subsystem vendor ID</td><td colspan="2">Subsystem device ID</td></tr>
<tr><td>0x30</td><td colspan="4">Expansion ROM base address</td><td>CP</td><td colspan="3">Reserved</td></tr>
<tr><td>0x38</td><td colspan="4">Reserved</td><td>IL</td><td>IP</td><td>MG</td><td>ML</td></tr>
</table>

Figura 1.3: Intestazione dello spazio di configurazione di un dispositivo PCI. Questi i significati delle sigle: RI=Revision ID, CL=Cache Line, LT=Latency Timer, HT=Header Type, BIST=Built-In Self Test, CP=Capabilities Pointer, IL=IRQ Line, IP=IRQ Pin, MG=MIN_GNT, ML=MAX_LAT

[5] Anche se esistono particolari classi di dispositivi che fanno eccezione.

Lo spazio di configurazione ha una dimensione minima di 256 byte: i primi 64 contengono un'intestazione comune a tutti i dispositivi ed hanno un significato ben definito nello standard, mentre gli altri sono eventualmente decisi dalla casa produttrice. All'interno dell'intestazione stessa, solo i primi 16 byte assumono sempre lo stesso significato, mentre i restanti 48 byte dipendono dal tipo di funzione. Il campo *HT (Header Type)* definisce il tipo di intestazione e conseguentemente determina il significato degli ultimi 48 byte: in tabella 1.1 vediamo degli esempi.

Valore	**Tipo di funzione**
0x00	Generico
0x01	PCI-to-PCI bridge
0x02	CardBus bridge

Tabella 1.1: Tipi di intestazioni per lo spazio di configurazione PCI

L'obiettivo principale dell'intestazione è quello di permettere una precisa identificazione della periferica e renderla controllabile da un driver. Di seguito sarà spiegato il significato solo di alcuni campi e si farà riferimento al caso di un *Header Type* uguale a `0x00`; la figura 1.3 mostra lo spazio di configurazione ed è realizzata con questa stessa assunzione. Per avere un esempio, in ambiente Linux si può ottenere una stampa degli spazi di configurazione delle periferiche presenti nel sistema con il semplice comando "`lspci -vvxxx`". È importante notare che i campi multi-byte seguono un'ordinamento dei byte di tipo *little-endian*, quindi l'indirizzo più basso contiene il byte meno significativo.

Per l'identificazione della funzione del dispositivo sono 5 i campi obbligatori:

- *Vendor ID*: identifica il produttore dell'hardware ed il suo valore appartiene ad una base di dati globale mantenuta dal PCI Special Interest Group [26]; `0xffff` è un valore nullo che indica un errore;

- *Device ID*: specifica il dispositivo ed è impostato dal produttore.

- *Revision ID*: è un identificativo di revisione specifico del dispositivo ed impostato dal produttore; estende il significato di *Device ID* e non può essere nullo;
- *Header Type*: determina il significato degli ultimi 48 byte dell'intestazione;
- *Class Code*: indica il tipo di funzione, quindi serve ad esprimere una classificazione per la funzione del dispositivo. È composta da 3 sottocampi di un byte ciascuno (ordinati in *little-endian*): Base Class Code, Sub Class Code, Specific Programming Interface. Quando il Base Class Code (quindi il byte con indirizzo più alto) è `0xff` significa che la funzione non appartiene a nessuna classe specifica.

Oltre a questi il produttore può usare i campi *Subsystem Vendor ID* e *Subsystem Device ID* per fornire un'ulteriore specificazione della funzione del dispositivo.

Nei 6 campi *Base address* vengono memorizzati gli eventuali indirizzi delle risorse, quali porte di I/O oppure locazioni di memoria: questi valori sono inseriti dal firmware del dispositivo in collaborazione con il software di boot del sistema, che prima di passare il controllo al kernel mappa ogni registro della periferica su un indirizzo dello spazio di I/O (solo per le porte di I/O) oppure dello spazio di memoria. Poiché gli indirizzi sono di 32 bit[6] esattamente come la dimensione dei campi *base address*, non sarebbe possibile sapere quale registro punta allo spazio di I/O e quale alla memoria, nonché conoscere la dimensione delle aree puntate. Per risolvere questo problema lo standard PCI impone che la dimensione delle aree di memoria accessibili dagli

[6] In realtà non è esattamente sempre così: ad esempio, lo spazio di I/O negli x86 è formato da indirizzi di 16 bit ed esistono sistemi con indirizzamento della memoria a 64 bit. Il fatto è che il PCI vuole essere indipendente dall'architettura e quindi prevede dimensioni standard di 32 bit, ma poi è in grado di adattarsi alle specifiche esigenze: per tornare sugli esempi precedenti vediamo che per le porte di I/O a 16 bit non si fa altro che azzerare i restanti bit di un *base address*, mentre per gli indirizzi a 64 bit si usano due campi contigui.

indirizzi contenuti nei *base address* sia sempre una potenza di 2 compresa tra 16 byte e 2 gigabyte: questo, unito all'allineamento degli indirizzi permette di avere in ogni campo almeno 4 bit liberi che possono essere usati come flag per distinguere fra porte di I/O, indirizzi di memoria a 32 bit e indirizzi di memoria a 64 bit. In ultimo va detto che questi bit hanno anche altre funzioni e che la dimensione precisa delle aree referenziate si ricava dall'indirizzo, che quindi deve essere scelto con cura dal software di boot. Per approfondimenti si rimanda a [26].

Il campo *Interrupt Line*, a cui si può accedere in lettura e scrittura, indica quale ingresso del controllore di interruzioni viene utilizzato dalla funzione: vi troviamo quindi il numero di IRQ. *Interrupt Pin* indica invece quale dei 4 pin a disposizione per ciascuna scheda PCI viene utilizzato.

1.4 Galil DMC 1800

Il driver scritto in questa tesi è dedicato ai controllori dinamici di movimento *Galil DMC Serie Optima 1800*, interfacciati al sistema attraverso il bus PCI. Attualmente sono 8 i modelli disponibili, tutti con le medesime caratteristiche e come unica differenza il numero di assi di movimento che sono in grado di controllare: il modello 1810 controlla un solo asse, il 1820 fino a due assi, il 1830 fino a tre e così via, fino al modello 1880 in grado di controllare da 1 ad 8 assi contemporaneamente. La scheda disponibile per lo sviluppo di questa tesi è stata la Galil DMC 1860, che può quindi controllare da 1 a 6 assi di movimento.

Le schede incorporano un microcomputer Motorola 68331 a 32 bit dotato sia di memoria RAM volatile che di memoria EEPROM non volatile e dispongono di molte caratteristiche avanzate, quali ad esempio:

- preazione con regolatori PID[7] per accelerazione e velocità degli assi;

[7] Ovvero con azione Proporzionale, Integrativa e Derivativa.

Figura 1.4: Scheda Galil DMC 1840

- 4 megabyte di memoria flash EEPROM non volatile per eseguire fino ad 8 programmi contemporaneamente, memorizzare parametri e firmware;
- FIFO[8] ausiliaria per il monitoraggio degli assi;
- controllo ad alta velocità per motori passo-passo (fino a 3 milioni di step al secondo) o servo-assistiti (fino a 12 milioni di passi encoder al secondo);
- supporto per molte modalità di movimento (posizionamento passo-passo, movimento ad impulsi, interpolazione lineare o circolare, controllo elettronico del cambio di velocità attraverso ingranaggi, ...);
- sincronizzazione con eventi esterni grazie a diversi ingressi ed uscite sia analogici che digitali che si possono interfacciare con sensori, joystick o trasduttori di pressione.

I controllori DMC 1800 sono comandati attraverso un semplice linguaggio composto da circa 200 comandi ASCII, ciascuno composto da due lettere.

[8]Con il termine FIFO, salvo diversa indicazione, si intenderà un buffer di memoria dove i dati entrano ed escono secondo la politica FIFO, ovvero First In First Out, dove il primo ad essere arrivato sarà il primo ad essere letto.

Molti comandi ovviamente accettano uno o più parametri; inoltre c'è la possibilità di comporre programmi *batch* che, una volta scaricati sulla memoria della scheda, vengono poi eseguiti autonomamente. Il processore è ovviamente in grado di eseguire solo comando codificati in binario e per questo la scheda è dotata di un'unità di traduzione dei comandi da ASCII a binario, che impiega circa 0.350 millisecondi per ogni comando; alcuni comandi, per aumentare le prestazioni possono essere scritti direttamente in binario. Per maggiori informazioni sul formato e sulle caratteristiche dei comandi si veda [25].

1.4.1 Buffer FIFO ed interruzioni

Per la realizzazione di un driver l'aspetto di maggiore interesse è il metodo di comunicazione adottatato dal dispositivo verso il resto del sistema. La figura 1.5 mostra un primo schema semplificato dell'interfaccia che la scheda espone attraverso il bus PCI.

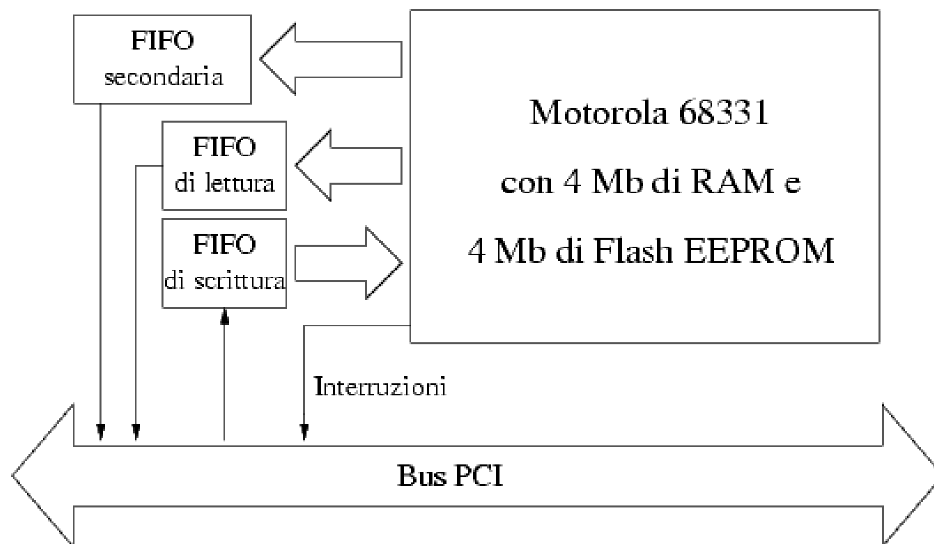

Figura 1.5: Schema semplificato dell'interfaccia verso il bus PCI delle schede Galil DMC 1800

Ci sono due FIFO primarie, entrambe in grado di immagazzinare 512 ca-

ratteri (ognuno dei quali occupa un byte) e vengono utilizzate una in scrittura ed una in lettura. Il funzionamento è il seguente:

1. i comandi (ASCII o binari) vengono scritti nella FIFO di scrittura
2. se necessario vengono tradotti in binario
3. vengono eseguiti dal processore secondo l'ordine *First In First Out* (primo arrivato primo servito)
4. al termine dell'esecuzione vengono posti nel buffer di lettura i seguenti caratteri:[9]
   - "?" se il comando non è stato riconosciuto;
   - ":", se il comando è stato eseguito con successo e non restituisce alcun tipo di dato;
   - per comandi che devono ritornare dei dati, saranno restituiti questi ultimi seguiti da "CR LF :" (carriage return, line feed, due punti).

   È anche possibile abilitare una funzione di eco, nel qual caso il nome del comando precederà quanto descritto nell'elenco precedente.

È presente anche una FIFO secondaria, di sola lettura, che fornisce informazioni dettagliate per ogni asse controllato. La dimensione del buffer varia in base al modello specifico: è composto di 40 byte comuni più altri 28 byte per ogni asse. Ad esempio la Galil DMC 1860 (6 assi) ha una FIFO secondaria di: $40 + 6 * 28 = 208$ bytes.

Viene utilizzata anche una linea di IRQ e la scheda può generare interruzioni in 16 diversi eventi (riportati in tabella 1.2), ognuno dei quali però deve essere esplicitamente abilitato attraverso il comando `EIm,n`, dove `m` è una mappa di 16 bit (uno per ogni evento) ed i bit ad 1 segnalano gli eventi da riportare attraverso le interruzioni. Il secondo parametro `n` viene valutato solo

[9]Ciò che viene descritto qui si discosta in alcuni punti da quanto affermato in [24], ma corrisponde a quanto è stato verificato sperimentalmente.

nel caso in cui l'ultimo bit di `m` sia 1 ed è anch'esso una mappa di 8 bit: i bit ad 1 indicano per quali, fra le 8 linee di ingresso esterne, si richiede un'interruzione ogni qualvolta ci siano dati pronti. Per ogni interruzione segnalata dal dispositivo, quest'ultimo prepara in un apposito registro (che vedremo descritto più avanti) un *byte di stato* che indica il motivo dell'interruzione; la lista dei possibili stati è anch'essa indica in tabella 1.2.

Oltre a quanto appena descritto esiste un ulteriore modo in cui si possono generare interruzioni: l'*user interrupt*. Infatti esiste un comando `UIn` che genera semplicemente un'interruzione. Il parametro `n` può essere un numero compreso tra 0 e 15 ed il vettore di stato corrispondente è un valore tra `0xf0` e `0xff`. In ultimo, per completezza, va detto che il byte di stato vale `0x00` quando non ci sono interruzioni.

Bit di `m`	**Condizione**	**Byte di stato**
0	Movimento dell'asse X completo	0xD0
1	Movimento dell'asse Y completo	0xD1
2	Movimento dell'asse Z completo	0xD2
3	Movimento dell'asse W completo	0xD3
4	Movimento dell'asse E completo	0xD4
5	Movimento dell'asse F completo	0xD5
6	Movimento dell'asse G completo	0xD6
7	Movimento dell'asse H completo	0xD7
8	Movimento di tutti gli assi completo	0xD8
9	Eccessiva posizione di errore	0xC8
10	Limite dello switch raggiunto	0xC0
11	Timer di guardia	0xD9
12	Riservato	
13	Esecuzione di un programma conclusa	0xDB
14	Comando eseguito	0xDA
15	Dati in ingresso pronti (usa la mappa `n`)	0xE1–0xE8

Tabella 1.2: Eventi che le schede Galil DMC 1800 sono in grado di segnalare attraverso interruzioni

1.4.2 Interfaccia di comunicazione

La figura 1.5 di pagina 18 illustra in maniera semplificata le possibilità di comunicazione con la scheda. Di seguito scenderemo ad un livello di dettaglio superiore, mostrando ad esempio in che modo è possibile accedere alle aree di memoria delle FIFO e come leggere i byte di stato delle interruzioni.

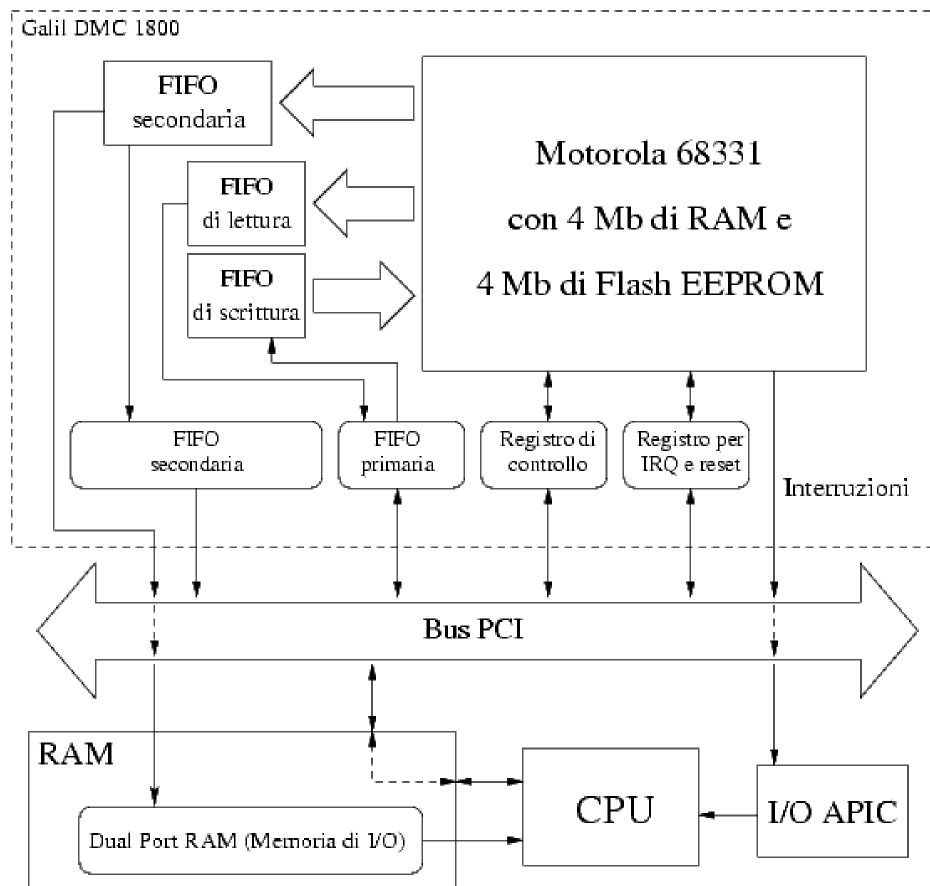

Figura 1.6: Porte e memoria di I/O delle schede Galil DMC 1800 (le porte di I/O sono rappresentate dai blocchi con gli angoli arrotondati all'interno della scheda Galil, mentre la memoria di I/O è esplicitamente indicata all'interno della RAM).

Lo schema di figura 1.6 illustra le porte e la memoria di I/O di cui dispongono le schede della serie Galil DMC 1800 ed i metodi che la CPU del sistema

deve usare per accedervi: le frecce verso il bus PCI mettono in evidenza i registri di sola lettura e quelli di lettura e scrittura.

Dalla figura 1.6 possiamo vedere che le schede Galil DMC 1800 dispongono di 4 registri, accessibili mediante altrettante porte di I/O. Inoltre, ma solo nelle ultime revisioni hardware[10], è disponibile anche una locazione di memoria di I/O (il Dual Port RAM) che rappresenta un modo alternativo di accedere alla FIFO secondaria. Purtroppo la scheda a disposizione per questo lavoro di tesi non possiede il supporto al Dual Port RAM e quindi, poiché non è stato possibile effettuare dei test, l'argomento non sarà ulteriormente approfondito.

1.4.3 Spazio di configurazione PCI

Le schede Galil DMC 1800 sono dei dispositivi PCI sui quali è presente un'unica funzione (vedi paragrafo 1.3) e quindi dispongono di un solo spazio di configurazione: vediamo come è fatto e che cosa contiene.

Innanzi tutto il campo *Header Type* contiene `0x00`, quindi il tipo di intestazione è esattamente quello di figura 1.3 di pagina 13. Vediamo invece in tabella 1.3 il contenuto dei campi necessari ad identificare la funzione della scheda PCI, così come descritto nel paragrafo 1.3.3.

Campo	**Valore**
Vendor ID	0x10B5
Device ID	0x9050
Subsytem Vendor ID	0x1079
Subsystem Device ID	0x1800
Base Class Code	0xFF

Tabella 1.3: Campi per l'indentificazione della funzione nello spazio di configurazione delle schede Galil DMC 1800

[10] In particolare dalla revisione H per i DMC 1810-1840 e dalla revisione E per i DMC 1850-1880.

Ci sono altri tre campi molto importanti il cui valore però non è statico, come per quelli appena descritti, bensì è impostato durante la fase di inizializzazione del sistema. Vediamo quali sono:

- *IL (Interrupt Line)*: qui troveremo la linea di interruzione assegnata al dispositivo; è importante notare che queste schede supportano la condivisione della linea di IRQ;
- *Base Address 0*: in questo campo si trova l'indirizzo iniziale della memoria di I/O, definita Dual Port RAM, per l'accesso alla FIFO secondaria; come già detto, questa memoria non è sempre disponibile e la dimensione della memoria varia in base al modello e può essere ricavata dall'indirizzo stesso, secondo le regole definite dallo standard PCI;
- *Base Address 2*: qui si trova l'indirizzo iniziale di un gruppo di indirizzi appartenenti allo spazio di I/O e corrispondenti alle quattro porte di I/O schematizzate in figura 1.6; anche in questo caso la dimensione è ricavabile dall'indirizzo, ma è sempre il minimo definito dallo standard, ovvero 16 byte.

1.4.4 Le porte di I/O

I dispositivi DMC 1800 possono essere interamente controllati attraverso 4 porte di I/O: l'indirizzo di base che si trova nel campo Base Address 2 dello spazio di configurazione viene impostato ad ogni avvio del sistema e quindi per semplicità lo indicheremo con `N`.

Direttamente all'indirizzo `N` troviamo la porta di I/O principale grazie alla quale si può accedere alle FIFO primarie. I trasferimenti devono avvenire un byte alla volta. Scrivendo su questa porta si aggiungono dati alla FIFO di scrittura, mentre leggendo si prelevano dati dalla FIFO di lettura. È per mezzo di questa porta quindi che si passano i comandi alla scheda e si possono leggere le risposte del controllore.

Da una seconda porta di I/O, all'indirizzo `N+0x4`, è possibile accedere sia in lettura che in scrittura ad un registro di controllo: la sua funzione è

sopratutto quella di descrivere lo stato delle FIFO, ma anche di gestire l'uso degli IRQ. In tabella 1.4 è indicato il significato di ogni suo bit.

Bit	Accesso	Significato
0	Lettura	Se 1, FIFO di scrittura piena
1	Lettura	Se 1, FIFO di scrittura contiene più di 256 byte, quindi è piena per più di metà della sua dimensione
2	Lettura	Se 1, FIFO di lettura vuota
3	Lettura	Se 1, FIFO secondaria in aggiornamento
4	Lettura/Scrittura	Congelamento della FIFO secondaria: Scrivendo 1 si congela la FIFO Scrivendo 0 si riabilita la FIFO Se 1, FIFO congelata
5	Lettura/Scrittura	Stato dell'IRQ: Se 1, IRQ pendente Scrivendo 1 si da l'ack per l'IRQ
6	Lettura/Scrittura	Gestione IRQ: Scrivendo 1 si abilitano gli IRQ Scrivendo 0 si disabilitano gli IRQ Se 1, IRQ abilitati
7	Lettura	Se 1, FIFO secondaria vuota

Tabella 1.4: Significato dei bit del registro di controllo per le schede Galil DMC 1800

All'indirizzo di I/O `N+0x8` troviamo la terza porta di I/O. Questa possiede una doppia funzione: usata in lettura, successivamente all'invio di un'interruzione, restituisce uno dei byte di stato elencati in tabella 1.2 (a pagina 20), oppure un valore compreso tra `0xf0` e `0xff`, se l'interruzione era stata generata attraverso il comando `UI` (vedi paragrafo 1.4.1). Usata in scrittura funziona invece come registro di reset; in tabella 1.5 sono elencati i possibili valori ed il loro significato.

In ultimo all'indirizzo `N+0xC` è accessibile l'ultima porta di I/O, di sola lettura, dalla quale è possibile estrarre i dati della FIFO secondaria. Questa

Byte da scrivere	**Significato**
10000000	Reset hardware del controller
00000100	Reset della FIFO di scrittura
00000010	Reset della FIFO di lettura

Tabella 1.5: Valori possibili per il registro di reset per i DMC 1800

porta è l'unica che ha una larghezza dati di 32 bit e quindi può essere acceduta leggendo 4 byte per volta.

L'uso in dettaglio di questi registri e le procedure di lettura e scrittura saranno descritte nel capitolo 4, dove viene descritto il driver di basso livello.

Capitolo 2

Kernel Linux 2.6

L'obiettivo di questo capitolo è quello di descrivere alcune parti del kernel di Linux che è necessario conoscere al fine di poter sviluppare un driver per una scheda PCI. La descrizione non sarà eccessivamente dettagliata (per questo si rimanda a [2] e [1]), ma si cercherà di introdurre la maggior parte degli strumenti necessari.

Il kernel di Linux è in continua evoluzione ed il driver è stato sviluppato sempre sull'ultima versione disponibile. Ora per ottenere una descrizione coerente è necessario fissare una versione del kernel sulla quale si potrà fare riferimento per approfondire i concetti spiegati in questa tesi: questa sarà la versione 2.6.15che, non a caso, è anche la minima versione necessaria per poter compilare il codice sorgente del driver.

2.1 Creare un modulo del kernel

Come accennato nell'introduzione è possibile sviluppare parti del kernel sottoforma di moduli esterni, che possono essere poi inseriti nell'immagine del kernel mentre quest'ultima è in esecuzione. Naturalmente esiste una ben precisa interfaccia che i moduli devono rispettare al fine di essere correttamente riconosciuti da Linux; inoltre anche la fase di compilazione del modulo deve rispettare alcune regole. Di seguito vedremo descritti questi due aspetti al

fine di ottenere le conoscenze minime per comprendere la struttura di base di un modulo e come può interagire con il kernel.

2.1.1 Struttura di un modulo

Linux prevede la possibilità che, al caricamento di un modulo, sia eseguita una funzione di inizializzazione e che analogamente, quando il modulo viene rimosso, venga eseguita una funzione di uscita; inoltre un modulo può esportare delle sue funzioni in maniera tale da renderle disponibili nel resto del kernel. Queste semplici caratteristiche sono i metodi con cui, attraverso un modulo, si possono estendere le funzioni del kernel. Vediamo come possono essere utilizzate:

- La funzione di inizializzazione, essendo la prima ad essere eseguita, deve preparare il modulo a svolgere le proprie funzioni; i dettagli dipendono ovviamente dal tipo di modulo che si sta realizzando. Vediamo alcune tipiche operazioni da effettuarsi al caricamento del modulo:
  - allocare le strutture dati dinamiche;
  - inizializzare le strutture dati usate dal modulo;
  - se il modulo controlla un dispositivo (quindi è un driver) deve ricercare il dispositivo, inizializzarlo ed eventualmente richiedere al kernel le risorse specifiche, come la linea di IRQ oppure delle regioni o porte di I/O; inoltre deve informare il kernel su quali sono le funzioni che ridefiniscono a basso livello le chiamate di sistema di interesse per il driver (vedi introduzione) ed iscriverlo nel *sysfs* filesystem;
  - un altro importante compito di una funzione di inizializzazione può essere quello di lanciare un kernel thread, oppure creare nuovi file nel *proc* filesystem.
- La funzione di uscita in genere ha il compito di rilasciare tutte le risorse ottenute durante la fase di inizializzazione: spesso ha una serie di istru-

zioni corrispondenti, ma eseguite in ordine inverso, a quelle presenti nella funzione di inizializzazione.

- Una funzione di un modulo *esportata* è utilizzabile in tutti gli altri file sorgenti del kernel.[1] Altre parti del kernel possono quindi utilizzare tali funzoni e questo quindi evidenzia come tale metodo aggiunga funzioni al kernel stesso.

Vediamo ora un esempio di un piccolo modulo per poi discuterne la struttura:

```
#include <linux/module.h>

MODULE_AUTHOR("Emiliano Betti");
MODULE_ALIAS("Example_Module");
MODULE_DESCRIPTION("Example module for Linux Kernel 2.6");
MODULE_VERSION("0.1");
MODULE_LICENSE("GPL");

static int func_init(void)
{
        printk(KERN_INFO "Example module loaded\n");
        return 0;
}

static void func_cleanup(void)
{
        printk(KERN_INFO "Example module unloaded\n");
}

module_init(func_init);
module_exit(func_cleanup);
```

[1] Ci sono solo dei vincoli imposti dalla licenza che copre i file del modulo e quelli del kernel, ma questi saranno discussi più avanti in questo capitolo.

```
void func_extern(void)
{
        printk(KERN_INFO "Example of function visible also"
                         " in all other kernel code\n");
}

EXPORT_SYMBOL_GPL(func_extern);
```

Queste poche righe rappresentano la struttura minima di un modulo. Il file `linux/module.h` contiene la definizione delle macro necessarie a creare un modulo, le funzioni `func_init()` e `func_clenup()` implementano le istruzioni da eseguire rispettivamente in caricamento e rilascio del modulo, mentre i comandi `module_init()` e `module_exit()` indicano al kernel quali sono appunto le funzioni di `init` ed `exit`. Le macro `MODULE_AUTHOR`, `MODULE_ALIAS`, etc, sono facoltative e permettono di inserire alcuni attributi. La funzione `func_extern()`, mediante il comando `EXPORT_SYMBOL_GPL` è esportata e quindi resa disponibile anche al resto del kernel dove il modulo sarà caricato.

Licenze

È importante fare qualche considerazione sui problemi di licenza. Come si può notare sia il comando `MODULE_LICENSE`, sia `EXPORT_SYMBOL_GPL` fanno riferimento alla licenza GPL (GNU Public License [21]): questi comandi specificano che il modulo è rilasciato sotto licenza GPL e che la funzione esportata può essere usata solo dai moduli del kernel che rispettano tale licenza. Si possono scegliere altre licenze e l'effetto è naturalmente equivalente, oppure si può non indicare alcuna licenza ed in questo caso il modulo non potrà usare le funzioni esportate sotto licenza, mentre tutti potranno usare le funzioni da lui esportate.

2.1.2 La compilazione

Nella versione 2.6 di Linux è strettamente necessario che un modulo venga compilato sul kernel nel quale verrà caricato[2]. Questo significa che è necessario non solo che nel sistema ci siano gli header file del kernel in uso (come era sufficiente nella versione 2.4), ma che quest'ultimo sia stato precedentemente compilato e che quindi siano disponibili il file di configurazione ed alcuni file oggetto che saranno collegati con il modulo che si sta creando.

Inoltre la compilazione deve avvenire attraverso il `Makefile`[3] del kernel stesso, quindi è necessario che il modulo disponga di un proprio `Makefile` che deve richiamare quello del kernel e che, a sua volta, verrà poi chiamato da quest'ultimo: quelle che seguono sono alcune righe del `Makefile` del modulo di esempio mostrato nel precedente paragrafo ed evidenziano proprio questo aspetto.

```
ifneq ($(KERNELRELEASE),)
obj-m:= example_module.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)
default:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

2.1.3 Il caricamento

Una volta compilato un modulo del kernel di Linux si ottiene un file oggetto con estensione *ko* (kernel object) e questo può essere inserito e collegato con l'immagine del kernel attualmente in esecuzione, con il comando:

```
insmod ./example_module.ko
```

[2]In realtà questo aspetto sta subendo delle modifiche ed è probabile che in futuro sarà possibile caricare dei moduli compilati su kernel con versioni precedenti a quella in esecuzione.

[3]File di script per l'applicazione *GNU Make*: vedi [22] per maggiori informazioni.

Durante il caricamento è anche possibile passare dei parametri al modulo, purché questi siano stati previsti dallo sviluppatore. Oltre ad `insmod`, esistono altri programmi che permettono di manipolare i moduli:

- `lsmod`: mostra i moduli attualmente caricati
- `rmmod`: rimuove un modulo
- `modprobe`: carica un modulo senza doverne indicare il percorso preciso (lo cerca in un albero di cartelle predefinite)

2.2 Modi di esecuzione dei processi

Le CPU, e di conseguenza i sistemi operativi, definiscono diversi modi di esecuzione per i processi. Questo in genere permette di distinguere dei livelli di privilegio secondo i quali un processo può accedere o meno a certe aree di memoria, oppure ha il permesso di eseguire determinate istruzioni. Linux, in maniera molto semplice, definisce due livelli di privilegio:

- *User mode* per i processi utente,
- *Kernel mode* per eseguire operazioni senza alcuna restrizione.

Un'applicazione normalmente viene eseguita in *User mode*, mentre quando richiede un servizio del kernel, come ad esempio l'esecuzione di una chiamata di sistema, deve passare in *Kernel mode*. Infatti, tutti i servizi provveduti dal kernel vengono forniti in *Kernel mode*, quindi anche il codice di un driver, ovvero di un modulo del kernel, sarà eseguito in *Kernel mode*. Questo comporta alcune conseguenze da tenere in considerazione:

- non esistono restrizioni sulle operazioni che si possono eseguire; ad esempio è possibile mascherare le interruzioni oppure accedere direttamente alle porte di I/O;
- l'assenza di restrizione impone un elevato livello di attenzione da parte dello sviluppatore del kernel, poiché, mentre in un'applicazione *User*

mode un errore provoca in genere il crash del solo processo, programmando in *Kernel mode* si può bloccare l'intero sistema;

- lo *stack* del kernel ha una dimensione inferiore a quella di due *page frame*, che nella maggior parte dei sistemi hanno una dimensione di 4 Kilobyte ciascuna; meno di 8 Kb di stack totali possono essere molto pochi se non si fa un'attento utilizzo della memoria.

2.3 Strumenti del kernel utilizzati nel driver

Il kernel di Linux dispone di una serie di strumenti la cui funzione è quella di risolvere problemi di programmazione piuttosto comuni o semplicemente volti a semplificare lo svolgimento di operazioni molto frequenti. Inoltre vi sono anche strumenti utili per gli sviluppatori di driver. Alcuni esempi sono:

- l'identificazione di un dispositivo PCI;
- l'allocazione dinamica della memoria;
- i problemi di sincronizzazione e quindi la gestione della concorrenza sulle strutture dati condivise;
- la possibilità di creare code di lavoro o particolari processi per svolgere compiti che devono essere differiti nel tempo o venir eseguiti in parallelo.

Questi strumenti (e molti altri) sono stati usati per sviluppare il software di controllo per le schede Galil DMC 1800. Per questo motivo seguirà un'introduzione alle funzioni di comune utilizzo nello sviluppo di un driver.

2.3.1 Inizializzazione del bus PCI

Le schede Galil DMC 1800 descritte nel paragrafo 1.4 sono basate su bus PCI e quindi per realizzarne il driver è necessario prima di tutto sviluppare un modulo che, al suo caricamento, identifichi il dispositivo PCI da controllare

e successivamente avvisi il kernel dell'associazione modulo-periferica, ovvero si registri come driver per quel dispositivo. A questo scopo il kernel mette a disposizione alcune strutture dati e delle funzioni che semplificano notevolmente il compito dello sviluppatore: tutto questo fa parte di una complessa porzione del kernel che si occupa di gestire il bus PCI e le periferiche ad esso connesse. Tale sistema sarà in seguito chiamato *PCI core*.

Strutture dati

Quando il PCI core rileva un dispositivo PCI[4] viene allocata un'apposita struttura dati: la `struct pci_dev`. Essa è composta da numerosi campi il cui scopo è descrivere il dispositivo a cui si riferisce. Fra le altre cose troviamo qui il contenuto di molti campi dello spazio di configurazione PCI, il numero di IRQ assegnato alla periferica ed un puntatore ad una `struct pci_driver`, descritta in seguito. Si può affermare che per il kernel di Linux un dispositivo PCI è rappresentato da un'istanza della struttura dati `pci_dev`. Oltre a questa, ve ne sono altre che contribuiscono all'individuazione ed alla gestione del dispositivo: di seguito saranno discusse solo le più importanti.

La `struct pci_device_id` viene utilizzata per l'identificazione dei dispositivi che si è in grado di controllare. I suoi campi sono infatti predisposti per essere confrontati con quelli dello spazio di configurazione PCI: il kernel, confrontando questi valori, è in grado di fare una prima associazione tra un driver ed una periferica. In tabella 2.1 è mostrata la corrispondenza tra alcuni campi della `struct pci_device_id` e i corrispettivi nello spazio di configurazione PCI.

Ogni modulo che ha funzione di driver per un dispositivo PCI deve inizializzare un array di strutture `pci_device_id` contenenti una struttura per ogni dispositivo compatibile, più un'ultima contente tutti zeri. Il codice che segue è un esempio che crea il suddetto array per la scheda Galil DMC 1800.

[4]Questo in genere poteva avvenire solo all'avvio del sistema, ma attualmente, grazie a nuove tecnologie come il *PCI hotplug*, in ogni momento un dispositivo PCI può apparire e scomparire.

Campi `struct pci_device_id`	**Campi in spazio di conf. PCI**
`vendor`	Vendor ID
`subvendor`	Subsystem Vendor ID
`device`	Device ID
`subdevice`	Subsystem Device ID
`class` `classmask`	Class Code

Tabella 2.1: Corrispondenza fra i campi della struttura *pci_device_id* e lo spazio di configurazione PCI

```
struct pci_device_id galil1800_idtable[] = {
{  .vendor = 0x10B5,
   .device = 0x9050,
   .subvendor = 0x1079,
   .subdevice = 0x1800,
   .class = 0,
   .class_mask = 0,
   .driver_data = 0 },
{ 0, }
};
```

I sistemi di hotplug e di caricamento dei moduli hanno bisogno di conoscere la corrispondenza tra moduli e dispositivi, quindi è necessario esportare nello spazio utente l'array di strutture `pci_device_id`. A questo scopo esiste la macro `MODULE_DEVICE_TABLE()` che ha esattamente questa funzione e riceve come argomento il puntatore all'array di strutture.

La `struct pci_driver` è la struttura principale di tutti i driver per dispositivi PCI: essa è infatti molto importante nella fase di registrazione del driver nel PCI core. Fra i suoi campi più significativi troviamo:

- `name`: il nome del driver (deve essere unico nel sistema ed in genere corrisponde al nome del modulo);

- `id_table`: puntatore all'array di strutture `pci_device_id` descritto precedentemente;
- `probe`: puntatore ad una funzione che viene chiamata dal PCI core ogni volta che identifica un dispositivo PCI che corrisponde alle caratteristiche descritte nell'`id_table`; la funzione accetta come argomento un puntatore alla `struct pci_dev` che descrive il dispositivo e ritorna un intero (non nullo in caso di errore);
- `remove`: il PCI core chiama questa funzione quando un driver rilascia il controllo di una periferica PCI a cui era associato; la funzione prende come argomento un puntatore alla `struct pci_dev` che descrive il dispositivo.

Nella struttura troviamo anche altri puntatori a funzioni, per la maggior parte legate al sistema di risparmio energetico: tutti questi puntatori devono indirizzare funzioni scritte dallo sviluppatore del driver. Vedremo maggiori dettagli nel paragrafo 3.6, dove saranno descritte le funzioni realizzate per il driver Galil.

Funzioni

Nella funzione di inizializzazione di un modulo per un driver PCI deve essere chiamata la funzione:

```
int pci_register_driver(struct pci_driver*)
```

Tale funzione riceve come argomento la `struct pci_driver` definita per il modulo e si preoccupa di registrare il driver nel PCI core. Vediamone a grandi linee il funzionamento:

1. confronta ogni elemento dell'array di strutture `pci_device_id` (puntato dal campo `id_table` della `struct pci_driver`) con il contenuto degli opportuni campi dello spazio di configurazione PCI dei dispositivi presenti nel sistema;

2. per tutti i dispositivi che vengono identificati come compatibili viene lanciata la funzione `probe`;

3. qualora la funzione `probe` abbia esito positivo il modulo viene registrato come driver per il dispositivo ed un puntatore alla struttura `pci_driver` viene salvato in un apposito campo della struttura `pci_dev` del dispositivo.

Quando il modulo viene rimosso dal sistema, nella sua funzione di uscita, deve essere chiamata la funzione:

```
void pci_unregister_driver(struct pci_driver*)
```

In questo modo il driver viene deregistrato e la funzione `remove`, definita dallo sviluppatore del modulo, viene eseguita: tutto ciò avviene per ogni singolo dispositivo associato.

Fra le attività che le funzioni `probe` e `remove` devono completare c'è, rispettivamente, l'abilitazione e la disabilitazione del dispositivo. Queste operazioni sono svolte semplicemente usando le funzioni:

```
int pci_enable_device(struct pci_dev*)

void pci_disable_device(struct pci_dev*)
```

Spesso in fase di abilitazione vengono assegnate le regioni di I/O e la linea per l'interruzione, quindi queste funzioni devono essere sempre chiamate prima di utilizzare queste risorse.

Prima di utilizzare le regioni di I/O è anche necessario conoscerne gli indirizzi. Per questo esistono due funzioni che ritornano rispettivamente il primo e l'ultimo indirizzo di una regione di I/O:

```
unsigned long pci_resource_start(struct pci_dev *dev,
                                                 int bar)
unsigned long pci_resource_end(struct pci_dev *dev,
                                                 int bar)
```

Come spiegato nel paragrafo 1.3.3 dall'indirizzo di I/O memorizzato nel *base address register* è possibile ricavare alcune informazioni in più. Queste sono restituite sotto forma di flag dalla funzione:

```
unsigned long pci_resource_flags(struct pci_dev *dev,
                                               int bar)
```

Il valore ottenuto può essere confrontato con una delle macro elencate di seguito:

- `IORESOURCE_IO`: l'indirizzo appartiene allo spazio di I/O,
- `IORESOURCE_MEM`: l'indirizzo appartiene allo spazio di memoria,
- `IORESOURCE_PREFETCH`: la memoria di I/O può essere precaricata,
- `IORESOURCE_READONLY`: la memoria di I/O è di sola lettura.

Le ultime tre funzioni descritte hanno tutte come secondo argomento un valore intero `bar`: questo non è altro che un numero compreso tra 0 e 5 che indica quale dei sei *base address register* deve essere letto.

Un'altra risorsa importante, oltre alle regioni di I/O, è rappresentata dalla linea di interruzione. Per sapere quale numero di IRQ è stato assegnato alla periferica è sufficiente leggere il campo `irq` della struttura `pci_dev` che lo descrive. Si potrebbe, anche in questo caso, accedere direttamente allo spazio di configurazione PCI, ma questa operazione è generalmente sconsigliata.

È importante precisare che tutte queste procedure di acquisizione delle risorse devono essere svolte nella funzione `probe`, dopo l'abilitazione del dispositivo.

2.3.2 Gestione della memoria

È stato già detto che un modulo del kernel viene eseguito in *Kernel mode* e che una delle conseguenze è che si ha a disposizione uno stack molto piccolo: questo aumenta notevolmente l'importanza delle allocazioni dinamiche della memoria.

La gestione della memoria in Linux è estremamente complessa soprattutto a causa della ricerca di un elevato livello di efficienza: di seguito quindi non si potranno elencare tutte le molteplici possibilità che il kernel offre, bensì saranno introdotte sole alcune funzioni di base, maggiormente utilizzate nello sviluppo del driver.

Due funzioni sono estremamente ricorrenti nella gestione della memoria:

```
void *kmalloc(size_t size, unsigned int flags)
void kfree(void *address)
```

La prima alloca `size` byte ritornando l'indirizzo iniziale, la seconda libera la memoria allocata all'indirizzo `address`. Importanti varianti sono:

```
void *kzalloc(size_t size, unsigned int flags)
void *kcalloc(size_t n, size_t size,
                              unsigned int flags)
```

La funzione `kzalloc()` alloca memoria, come `kmalloc()`, poi la inizializza a zero[11]. Invece `kcalloc()` viene utilizzata per allocare array di `n` elementi, ognuno dei quali ha dimensione `size`; infine, anche in questo caso, viene azzerata tutta l'area di memoria allocata.

Molto importante per tutte queste funzioni è la scelta del valore da passare nell'argomento `flags`. Anche in questo caso le possibilità sono molte, ma di seguito vedremo sole le due più comuni:

- `GFP_KERNEL`: questo è il metodo più comune di richiedere memoria al kernel; è importante precisare che con questo flag, se non ci sono pagine libere, la funzione può bloccare il processo[5] e quindi non deve essere utilizzato in contesti dove si richiede che l'esecuzione sia atomica ed, in ogni caso, si deve prestare attenzione ai problemi di concorrenza;

[5] In Linux i processi possono assumere diversi stati: in alcuni di questi il processo viene definito *bloccato* in quanto ha rilasciato il processore e non verrà eseguito finché non si verificherà un evento che lo risveglierà. Maggiori informazioni si possono trovare in [2].

- `GFP_ATOMIC`: complementare al precedente, alloca memoria solo se ci sono pagine libere, altrimenti fallisce; in questo modo può essere utilizzato in contesti atomici come gestori di interruzione, *tasklet* o *kernel timer*[6].

2.3.3 Gestione della concorrenza

Il kernel di Linux si occupa di controllare tutte le periferiche del sistema, di gestire la memoria, i processi, il filesystem, oltre ad avere molte altre funzioni meno evidenti. È ovvio che tutti questi compiti devono essere svolti in parallelo e che quindi il kernel abbia in ogni istante decine di flussi di esecuzione o kernel thread attivi, i quali spesso concorrono su strutture dati condivise. Detto questo è chiaro come, sviluppando un driver, si debba prestare molta attenzione al fine di garantire l'integrità dei dati, seppur permettendovi un accesso concorrente.

I casi in cui si possono generare problemi di concorrenza sono molteplici, vediamone alcuni:

- più processi utenti potrebbero utilizzare contemporaneamente la periferica e quindi il driver;
- i moderni sistemi SMP (Symmetric MultiProcessor) potrebbero eseguire il codice del driver parallelamente su più processori;
- il kernel Linux 2.6 è *preemptive*, quindi una parte del driver in esecuzione può essere interrotta in qualsiasi momento; inoltre ad interromperla potrebbe essere un'altra porzione dello stesso driver;
- un'interruzione asincrona può arrivare in ogni istante ed immediatamente il processore inizia ad eseguire il relativo gestore di interruzione;
- come si vedrà nel prossimo paragrafo, nei driver è comune differire alcuni compiti e spesso in questo modo si creano situazioni di concorrenza;

[6] Vedi paragrafo 2.3.4

- le periferiche che supportano l'*hotplug* possono sparire e riapparire senza alcun preavviso.

La gestione della concorrenza impone prima di tutto molta attenzione, ma ovviamente il kernel viene in aiuto fornendo potenti strumenti. Come al solito ne saranno descritti solo alcuni.

Semafori e mutex

Il compito principale di un *semaforo* è quello di proteggere un oggetto (in genere una struttura dati) indicando quando quest'ultimo è accessibile o meno. Tipicamente è composto da un singolo valore intero, inizializzato al numero massimo di utenti che possono utilizzare contemporaneamente l'oggetto protetto. Chi vuole ottenere il *lock*, deve tentare di decrementare il semaforo e vi riuscirà solo se questo è ancora maggiore di zero. Quando si vuole ottenere la mutua esclusione, ovvero un solo utente alla volta può prendere il *lock* è sufficiente inizializzare il semaforo ad 1: in questo caso i semafori vengono chiamati *mutex*.

La cosa che distingue i semafori da altri tipi di protezioni, come i *spinlock* che vedremo in seguito, è ciò che avviene quando si cerca di decrementare un semaforo che è già a zero: il processo si blocca, rilasciando il processore. Quando il semaforo verrà incrementato i processi che sono bloccati in coda per quel semaforo verranno svegliati e soltanto uno riuscirà ad ottenere il *lock*, mentre gli altri saranno di nuovo bloccati. Ovviamente, affinché questo sistema funzioni, deve essere garantito che le operazioni sui semafori siano atomiche e ciò è possibile grazie a speciali istruzioni fornite dai processori.

---

[6]Questa flessibilità dei semafori in parte contrasta con l'efficienza: infatti mentre per implementare un semaforo è necessaria una variabile intera, per un mutex basterebbe un solo bit. Sulla basa di questa considerazione gli sviluppatori del kernel stanno realizzazando una nuova implementazione dei mutex, con strutture dati e funzioni proprie, separate da quelle degli attuali semafori. Questa nuova implementazione è comunque ancora in una fase di test e pertanto nel resto della discussione non verrà trattata. Per maggiori informazioni si veda [10] e [17].

Vediamo alcune funzioni per manipolare i semafori:

```
void sema_init(struct semaphore *sem, int val)
void init_MUTEX(struct semaphore *sem)

void down(struct semaphore *sem)
void down_interruptible(struct semaphore *sem)

void up(struct semaphore *sem)
```

La prima funzione inizializza un semaforo: `val` è il numero massimo di utenti, mentre `sem` è la struttura che implementa i semafori in Linux. La funzione `init_MUTEX` semplicemente inizializza un semaforo con `val` ad 1, quindi un mutex. Le funzioni `down()` ed `up()` rispettivamente decrementano ed incrementano il semaforo. Quando il semaforo vale già zero (o meno), le funzioni `down()` e `down_interruptible()` bloccano il processo finché il semaforo non assume nuovamente un valore superiore a zero e l'operazione di decremento va a buon fine. La differenza fra le due funzioni è che con `down_interruptible()` il processo può essere sbloccato anche dall'arrivo di un segnale; in questo caso la funzione ritorna un valore diverso da zero. Esistono molte altre varianti, ma queste poche funzioni sono quelle sufficienti nella maggior parte dei casi.

Spinlock

I semafori sono largamente utilizzati nel kernel, ma, come vedremo più avanti, ci sono contesti in cui il flusso di esecuzione non può essere bloccato: in questi casi i semafori non sono adeguati e quindi occorre una soluzione diversa.

Gli *spinlock* hanno un funzionamento molto simile ai mutex, essi infatti permettono di ottenere l'accesso ad un solo utente alla volta, ma nel caso in cui il lock non è disponibile il processo non si blocca e rilascia la CPU, bensì cicla continuamente controllando che il lock si liberi. Una volta libero lo ottiene, a meno che un altro processo che stava nella sua stessa condizione non lo anticipi.

È importante precisare che i spinlock hanno senso solo su sistemi SMP. Infatti, come si può immaginare, in una macchina uniprocessore il loro uso bloccherebbe semplicemente il sistema: un processo che cicla su uno spinlock, in attesa che quest'ultimo si liberi, deterrebbe l'unico processore del sistema, non permettendo quindi a chi aveva acquistito lo spinlock di rilasciarlo.

I casi in cui non si può bloccare sono diversi, ma in genere sono quei contesti dove è richiesto un alto livello prestazionale. L'esempio più semplice sono i gestori di interruzione che in genere devono essere eseguiti in pochi microsecondi, ma anche altri contesti dove è garantita un'esecuzione atomica come *tasklet* e *kernel timer* che vedremo in seguito.

Vediamo le funzioni necessarie ad inizializzare uno spinlock ed ottenere e rilasciare un lock:

```
void spin_lock_init(spinlock_t *lock)

void spin_lock(spinlock_t *lock)
void spin_unlock(spinlock_t *lock)
```

Come spiegato, la funzione `spin_lock()` non ritorna il controllo finché non riesce ad ottenere il lock. Esistono delle varianti molto importanti:

```
void spin_lock_irqsave(spinlock_t *lock,
                         unsigned long flags)
void spin_unlock_irqrestore(spinlock_t *lock,
                         unsigned long flags)

void spin_lock_irq(spinlock_t *lock)
void spin_unlock_irq(spinlock_t *lock)

void spin_lock_bh(spinlock_t *lock)
void spin_unlock_bh(spinlock_t *lock)
```

Le funzioni `spin_lock_irqsave()` e `spin_lock_irq()` disabilitano le interruzioni (solo nel processore locale) prima di ottenere il lock: la differenza

è che la prima salva lo stato delle interruzioni in `flags` e lo ripristina quando rilascia il lock, mentre la seconda non compie alcun tipo di controllo. La funzione `spin_lock_bh()` disabilita solo le interruzioni software, lasciando attive quelle hardware.

Con un esempio sarà possibile chiarire il motivo che può spingere a disabilitare le interruzioni, mentre si detiene un lock. Si immagini di dover proteggere una struttura dati che viene utilizzata in un generico punto del driver e nel gestore di interruzione. Se il driver ottenesse il lock senza disabilitare le interruzioni, il gestore di interruzioni potrebbe prendere il controllo del processore mentre l'altra parte di codice detiene il lock. Il risultato è che il sistema andrebbe in stallo poiché il lock non può essere rilasciato finché il gestore di interruzione non rilascia il processore, ma questo non avviene finché il gestore non ottiene il lock: il sistema sarebbe in uno stato che viene comunemente chiamato *deadlock*.

Operazioni sui bit

Quando si richiedono prestazioni particolarmente elevate ed è sufficiente operare a livello di bit si possono utilizzare le operazioni atomiche sui bit. La velocità di esecuzione e l'atomicità sono garantite dal fatto che con un'unica istruzione del processore si possono eseguire operazioni di test e modifica sul singolo bit. Vediamo alcune funzioni:

```
void set_bit(nr, void *addr)
void clear_bit(nr, void *addr)
void change_bit(nr, void *addr)

int test_bit(nr, void *addr)
int test_and_set_bit(nr, void *addr)
int test_and_clear_bit(nr, void *addr)
int test_and_change_bit(nr, void *addr)
```

I nomi delle funzioni dovrebbero essere sufficientemente esplicativi da far capire il tipo di operazione; il bit su cui si agisce è determinato contando `nr` bit a partire dall'indirizzo `addr`. Le ultime tre funzioni ritornano il valore del bit, prima di essere modificato.

2.3.4 Processi differiti

È molto comune per un driver dover ritardare l'esecuzione di porzioni di codice, ad esempio per attendere che l'hardware completi alcune operazioni. Di seguito saranno presentati tre diversi strumenti tutti più o meno con questa funzione.

Prima di iniziare a spiegare come il kernel ritarda alcune operazioni è necessario aprire una piccola parentesi per introdurre il *timer interrupt*, ovvero uno dei modi con cui il kernel misura il tempo. Normalmente la linea di IRQ 0 viene utilizzata da un apposito circuito che genera interruzioni ad intervalli regolari: il *Programmable Interval Timer (PIT)*. Linux tiene traccia di queste interruzioni incrementando una variabile globale chiamata `jiffies`. Il circuito è programmabile e la frequenza delle interruzioni viene decisa in fase di compilazione del kernel: nella versione 2.6 può variare da 50 a 1200 interruzioni al secondo e la macro `HZ` rende accessibile questo valore in ogni parte del kernel. L'intervallo di tempo che intercorre fra due interruzioni generate dal PIT è in genere definito *tick*. Per fare un esempio, leggendo due volte la variabile `jiffies` e dividendo la differenza per `HZ` si ottengono i secondi trascorsi tra le due letture, con una precisione che è tanto più elevata, quanto più `HZ` è grande.

Kernel timer

I *kernel timer* sono particolarmente utili quando si vuole che una operazione venga eseguita in un istante ben determinato, senza che il processo attenda mantenendo il controllo della CPU. Sono molto utili anche quando è necessario eseguire un'operazione periodica, come ad esempio un *polling* su una periferica per controllare se ha dati pronti.

Un kernel timer essenzialmente è una struttura dati che istruisce il kernel ad eseguire una data funzione, con un particolare argomento, ad un fissato istante. È anche possibile realizzare dei *timer periodici* grazie al fatto che la funzione del timer, quando viene eseguita, può reinizializzare il timer stesso. Ad esempio, è in questo modo che si può realizzare un polling. Vediamo la struttura dati:

```
struct timer_list {
    /* ... qui ci sono molti altri campi... */
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
};
```

Il campo `expires` determina l'istante in cui il timer scade e la funzione puntata da `function` viene eseguita; `data` è l'argomento da passare alla funzione. Il tempo contenuto nella variabile `expires` è in realtà il valore che deve avere la variabile `jiffies` quando il timer sarà eseguito.

Le seguenti funzioni manipolano i timer:

```
void init_timer(struct timer_list *timer)

void mod_timer(struct timer_list *timer,
                      unsigned long expires)

void del_timer(struct timer_list *timer)
void del_timer_sync(struct timer_list *timer)
```

La prima inizializza tutti i campi di `timer` tranne i tre citati sopra, che in genere sono inizializzati direttamente assegnandogli un valore. La seconda modifica il valore del campo `expires` e quindi attiva (o riattiva) il timer. Le ultime due annullano un timer con la differenza che `del_timer_sync()` non restituisce il controllo finché non è sicura che il timer non è più in esecuzione su nessuna CPU.

Il codice delle funzioni lanciate dai kernel timer ha diverse restrizioni. Innanzitutto la scadenza del timer è segnalata attraverso interruzioni software e quindi il contesto di esecuzione delle funzioni è atomico. Il processo quindi non può mai essere bloccato e, ad esempio, non si possono usare i semafori. Inoltre non essendo nel contesto di esecuzione di un processo non si può accedere allo spazio utente e non si può utilizzare il puntatore al processo corrente (ovvero la variabile globale `current`). In genere con i timer si possono avere facilmente problemi di concorrenza, quindi è necessario prestare particolare attenzione.

In ultimo va precisata una particolarità dei timer: essi sono implementati attraverso particolari strutture dati, chiamate *variabili per-CPU*, che assumono un valore diverso per ogni processore da cui sono accedute. Questo fa si che un timer venga eseguito sempre sulla stessa CPU nella quale è stato registrato: in molti contesti gli effetti di questa caratteristica possono essere poco evidenti, ma in alcuni casi può essere sfruttata per risolvere problemi di concorrenza.

Tasklet

Quando è necessario differire l'esecuzione di alcune funzioni, ma si ha comunque l'esigenza di eseguirle prima possibile, lo strumento più adeguato è probabilmente il *tasklet*. L'esempio più diffuso si trova nei gestori di interruzione, i quali sono tipicamente divisi in *top half*, ovvero le istruzioni ad alta priorità che vanno immediatamente eseguite, e *bottom half*, ovvero quelle istruzioni che completano la gestione dell'interruzione, ma che possono essere ritardate (anche se in genere di poco): questa seconda parte è molto spesso implementata attivando un tasklet.

Ci sono alcune somiglianze con i timer:

- l'esecuzione avviene attraverso interruzioni software, quindi il contesto di esecuzione è atomico;
- sono sempre eseguiti nella stessa CPU su cui sono stati registrati;

- il prototipo della funzione eseguita è identico;
- l'implementazione della funzione da eseguire deve rispettare le stesse limitazioni indicate per i timer;
- un tasklet può riattivare se stesso.

La differenza principale risiede nel fatto che non si può indicare quando la funzione deve essere eseguita, ma sarà il kernel che cercherà di eseguirlo prima possibile. In genere, se il sistema non è particolarmente sovraccarico, la funzione sarà eseguita dopo pochi tick rispetto a quello di registrazione.

Un tasklet è identificato da un'istanza di una particolare struttura dati che lo descrive: la `struct tasklet_struct`. Vediamo di seguito parte di questa struttura dati e la funzione per inizializzarla:

```
struct tasklet_struct {
   /* ... qui ci sono molti altri campi... */
   void (*func)(unsigned long);
   unsigned long data;
};

void tasklet_init(struct tasklet_struct *t,
                  void (*func)(unsigned long),
                  unsigned long data)
```

Una volta definito il tasklet per richiedere che sia eseguito si possono usare due funzioni:

```
void tasklet_schedule(struct tasklet_struct *t)
void tasklet_hi_schedule(struct tasklet_struct *t)
```

L'unica differenza è che la seconda indica che il tasklet ha alta priorità e quindi dovrà essere eseguito prima di quelli in priorità normale. Vale la pena notare un'importante caratteristica: tasklet diversi possono essere

parallelamente eseguiti su diverse CPU, ma i tasklet dello stesso tipo[7] sono sempre serializzati e quindi non saranno mai eseguiti parallelamente su più CPU. Questo va tenuto in considerazione quando si pensa ai problemi di concorrenza.

Seguono altre funzioni, utili nella manipolazione dei tasklet:

```
void tasklet_disable(struct tasklet_struct *t)
void tasklet_enable(struct tasklet_struct *t)

void tasklet_kill(struct tasklet_struct *t)
```

La funzione `tasklet_disable()` disabilita il tasklet `t`: i tasklet registrati dopo la disabilitazione non verranno eseguiti finché `tasklet_enable()` non viene eseguita. Il kernel conta quante volte viene utilizzata la funzione `tasklet_disable()` e per la riabilitazione deve essere usata per lo stesso numero di volte la funzione `tasklet_enable()`. Infine `tasklet_kill()` assicura che il tasklet `t` non sia più eseguito: qualora ve ne fosse uno in esecuzione non ritorna finché questo non termina.

Workqueue

Si è già detto come tasklet e timer siano, dal punto di vista implementativo, poco flessibili: il fatto di essere eseguiti nel contesto di un'interruzione software pone molti vincoli, seppur garantendo elevate prestazioni. In situazioni meno stringenti sotto l'aspetto prestazionale, oppure in casi in cui non è possibile rispettare i vincoli di un ambiente di esecuzione atomico, possono risultare utili le *workqueue*. Come i tasklet ed i timer sono strutture dati con una funzione da eseguire, ma, al contrario degli altri due casi, l'esecuzione non avviene in ambiente atomico e non c'è alcun vincolo su quando la funzione debba essere eseguita.

La traduzione del termine "workqueue" spiega bene il senso di questi oggetti che in effetti non sono altro che "code" nelle quali sottomettere uno o

[7] Ovvero definiti da una stessa istanza della struttura dati *tasklet_struct*.

più “lavori”. Infatti creare una workqueue corrisponde semplicemente a creare speciali *kernel thread*, chiamati *work thread*, che restano in attesa di funzioni da eseguire. Come già detto, il contesto di esecuzione di un work thread non ha forti limitazioni. I work thread possono bloccare e quindi, ad esempio, si possono usare i semafori, ma non possono comunque accedere allo spazio utente; inoltre mentre per tasklet e timer la durata dell’esecuzione doveva essere necessariamente breve, per un lavoro sottomesso ad una workqueue non ci sono limiti di tempo, anzi è comune avere work thread sempre attivi e continuamente pronti a gestire particolari eventi.

La struttura dati `workqueue_struct` descrive una workqueue e le seguenti funzioni creano rispettivamente una coda per ogni processore, oppure un’unica coda comune, restituendone il puntatore. Come timer e tasklet anche le workqueue vengono eseguite sullo stesso processore da cui sono registrate.

```
struct workqueue_struct *create_workqueue(const char *name)
struct workqueue_struct *create_singlethread_workqueue(
                                          const char *name)
```

Una volta creata una workqueue è possibile sottomettervi dei lavori. A questo scopo è necessario inizializzare una struttura `work_struct`. Ciò è possibile attraverso due funzioni:

```
INIT_WORK(struct work_struct *work,
          void (*func)(void*), void *data)
PREPARE_WORK(struct work_struct *work,
          void (*func)(void*), void *data)
```

La prima volta che viene inizializzata la struttura `work` deve essere usata la funzione `INIT_WORK`. In caso di successive modifiche è sufficiente utilizzare `PREPARE_WORK`. Dopo queste istruzioni il lavoro è stato solo preparato, ma non si è chiesto ancora a nessun work thread di eseguirlo. Quest’ultimo passo è possibile attraverso le funzioni:

```
int queue_work(struct workqueue_struct *queue,
                               struct work_struct *work)
int queue_delayed_work(struct workqueue_struct *queue,
          struct work_struct *work, unsigned long delay)
```

La funzione `queue_work` sottomette il lavoro `work` nella coda `queue`. La seconda funzione fa la stessa cosa, ma chiede che il lavoro non venga iniziato prima che sia trascorso un tempo `delay`, espresso in tick.

In ultimo sono degne di nota altre due funzioni. Una per sapere quando una workqueue non ha più lavori da eseguire ed un'altra per terminare un work thread:

```
int flush_workqueue(struct workqueue_struct *queue)
int destroy_workqueue(struct workqueue_struct *queue)
```

2.4 Approfondimenti

In questo capitolo, per ovvi motivi di spazio, è stato possibile solo presentare brevemente una minima parte degli strumenti forniti dal kernel di Linux: per eventuali approfondimenti si consiglia la lettura di [2] e [1]. Inoltre il kernel è in continua evoluzione e per aggiornamenti sul suo sviluppo si possono leggere gli articoli elencati in [8]. Infine, per quanto concerne i driver e altri aspetti utilizzati in questa tesi si possono consultare i seguenti articoli: [16, 4, 18, 7, 14, 15, 9, 19, 13, 12, 3].

Capitolo 3

Organizzazione del driver

In questo capitolo si vuol fornire una visione generale del driver sviluppato in questa tesi. Gli obiettivi principali saranno:

- descrivere le caratteristiche più importanti del driver,
- spiegare come il driver fornisce un'astrazione dei dispositivi Galil DMC 1800 e quindi come queste schede e le loro funzioni sono accessibili all'utente,
- descrivere la fase di inizializzazione del driver.

3.1 Informazioni generali e licenza

Il driver è stato sviluppato come modulo del kernel di Linux. Il codice è interamente scritto in linguaggio C e la versione del kernel utilizzata per lo sviluppo è la 2.6.15. Questa è anche la versione minima necessaria per compilare il modulo, poiché quest'ultimo utilizza delle nuove funzioni del kernel introdotte proprio nella versione 2.6.15 [8].

Il driver è stato sviluppato e rilasciato secondo le regole della licenza GNU GPL versione 2 [21], con copyright detenuto dai due autori: Emiliano Betti e Marco Cesati.

3.2 Caratteristiche principali

Il driver supporta tutte le caratteristiche delle schede Galil DMC 1800, a parte un'unica eccezione.

Nella revisione H e successive dei modelli da 1810 a 1840 e nella revisione E e successive dei modelli da 1850 a 1880 è stato inserito un Dual Port RAM (DPRAM), ovvero un'area di memoria di I/O che permette un'accesso in sola lettura alla FIFO secondaria. In realtà questo è solo un modo alternativo e più efficiente di leggere la FIFO secondaria che; essa, come spiegato nel paragrafo 1.4, è anche accessibile (in tutte le revisioni, di tutti i modelli) attraverso una specifica porta di I/O. Questo driver non supporta l'accesso al Dual Port RAM, neanche per le schede che ne sono provviste, ma permette sempre la lettura della FIFO secondaria attraverso la porta di I/O: tale scelta è stata forzata dal fatto che la scheda a nostra disposizione per i test non è sufficientemente recente e quindi non dispone dell'area di memoria di I/O. In conclusione non potendo fare alcun test si è scelto, almeno per il momento, di non supportare questa caratteristica.

A parte l'eccezione appena descritta il driver supporta tutte le altre caratteristiche di base:

- invio dei comandi
- lettura delle risposte
- gestione delle interruzioni, con esportazione del vettore di stato delle interuzioni nello spazio utente
- lettura della FIFO secondaria
- supporto per il *sysfs* filesystem.[1]

Inoltre sono state sviluppate anche alcune caratteristiche avanzate:

- supporto per più schede

[1] Per informazioni vedi il paragrafo 3.6.5 e [2, 1, 5, 6, 20, 27].

- letture e scritture bloccanti (configurabile dall'utente)
- scrittura dei comandi bufferizzata (configurabile dall'utente)
- gestione di eventi speciali per monitorare lo stato del buffer di scrittura (attraverso questo stesso sistema si ottengono anche i vettori di stato delle interruzioni)
- predisposizione a pseudo-comandi; attualmente è implementato uno speciale comando di pausa che interrompe la scrittura sulla FIFO primaria per un intervallo di tempo selezionabile a runtime;
- possibilità di configurazione di alcuni parametri di funzionamento attraverso la chiamata di sistema `ioctl`, la scrittura su file del *sysfs*, oppure opzioni di caricamento del modulo.

Queste caratteristiche saranno meglio descritte man mano che si procede nella trattazione.

3.3 Comunicazione con lo spazio utente

Un driver che controlla una *periferica di I/O*, ovvero un dispositivo che in genere scambia informazioni con lo spazio utente, è detto *driver di I/O*: nella maggior parte dei casi, tale tipo di driver deve permettere alle applicazioni dello spazio utente di ricevere e inviare dati da e verso la periferica che controlla.

Come già introdotto, le schede Galil DMC 1800 controllano meccanismi che hanno da 1 ad 8 gradi di libertà ed a tale scopo sono predisposte per ricevere comandi dallo spazio utente secondo un particolare linguaggio sviluppato dalla Galil [25]. Esse sono quindi periferiche di I/O e conseguentemente il driver sviluppato in questa tesi è un driver di I/O.

In questo paragrafo vedremo come Linux permette alle applicazioni utente di interfacciarsi con i dispositivi di I/O; successivamente sarà introdotto lo

schema di comunicazione con le schede Galil DMC 1800 stabilito da questo driver.

3.3.1 I device file

Il filesystem di Linux distingue diversi tipi di file:

- *file regolari*: contengono dati oppure programmi (quindi codice eseguibile),
- *directory* e *symbolic link*: file utili per la costruzione dell'albero del filesystem,
- *FIFO* e *socket*: pseudo-file per la comunicazione fra processi,
- *device file*: pseudo-file che rappresentano i dispositivi di I/O; in particolare, ad ogni dispositivo di I/O corrisponde uno o più device file.

Nel kernel esiste inoltre uno strato software che gestisce le chiamate di sistema relative ai file (come, ad esempio, `open`, `close`, `read`, `write`,...), il cui scopo è quello di nascondere allo spazio utente le differenze tra i diversi tipi di file e quindi permettere l'utilizzo delle stesse funzioni. Ad esempio, attraverso i device file è possibile accedere alle periferiche di I/O utilizzando le stesse chiamate di sistema che operano sui file regolari. Si possono così leggere dati da un dispositivo hardware attraverso la seguente procedura:

- aprire il device file del dispositivo (chiamata di sistema `open`),
- leggere dati dal descrittore di file ottenuto (chiamata di sistema `read`),
- chiudere il device file (chiamata di sistema `close`).

Ogni chiamata di sistema è, a tal scopo, costituita da due parti: una parte comune ad ogni tipo di file, implementata dal kernel, ed una parte specifica. Quest'ultima è realizzata con funzioni spesso chiamate *file operation*: come vedremo, per i device file le file operation sono implementate dal driver del dispositivo.

È importante precisare che esistono due tipi di device file: *device file a blocchi* e *device file a caratteri*. Questa distinzione deriva direttamente da una classificazione dei dispositivi di I/O in due gruppi:

- **dispositivi a caratteri**: sono dispositivi che possono essere acceduti come un flusso di byte ed in genere l'accesso è possibile solo in modo sequenziale;

- **dispositivi a blocchi**: la differenza principale con i dispositivi a caratteri è che le operazioni di trasferimento dati hanno come unità minima di lavoro un blocco di byte di dimensione prefissata. Questo comporta una notevole differenza nel driver di basso livello, ovvero nel sistema di comunicazione con l'hardware. Inoltre il metodo di accesso in questo caso è tipicamente casuale. Un ottimo esempio sono i dischi rigidi.

Le schede Galil DMC 1800 sono un esempio tipico di dispositivo a caratteri.

Major e minor number

Un device file è identificato da 3 valori memorizzati nell'*inode* del file.[2] Il primo è un valore booleano che indica se si tratta di un device file a blocchi oppure a caratteri. Gli altri due valori sono variabili intere definite come *Major number* e *Minor number*.

Il major number identifica il driver associato con il dispositivo, anche se Linux permette a diversi driver di condividere lo stesso major number. Il minor number identifica il dispositivo a cui ci si riferisce. Va precisato che lo stesso major number ha un significato diverso per i device file a blocchi ed a caratteri.

Un driver deve quindi informare il kernel su quale sia il major number che lo identifica ed inoltre registrare il necessario numero di device file, ognuno

[2] L'*inode* è una struttura dati che descrive un file memorizzato in un filesystem. Per maggiori informazioni si veda [2].

con il proprio minor number. Più avanti in questo capitolo vedremo come questo driver effettua queste operazioni.

3.3.2 I device file per le schede Galil DMC 1800

In fase di progettazione del driver Galil si è scelto di interfacciare ogni scheda Galil DMC 1800 con le applicazioni utente tramite tre device file a caratteri. Vediamo di seguito quali sono i nomi logici utilizzati nel driver per questi device file ed una prima descrizione:

- **card**: attraverso questo device file è possibile inviare i comandi al buffer di scrittura e leggere le risposte generate dalla scheda; la scrittura può (opzionalmente) essere bufferizzata;
- **event**: è un device file di sola lettura che permette di leggere informazioni circa eventi speciali riguardanti lo stato del buffer di scrittura e le interruzioni; in merito a quest'ultime il device event permette di leggere i vettori di stato prodotti dalla scheda in seguito ad ogni interruzione;
- **info**: è un device file di sola lettura che permette di esportare nello spazio utente il contenuto della FIFO secondaria.

La figura 3.1 mostra un'evoluzione dello schema di figura 1.6 (pagina 21); essa mette in evidenza l'interazione tra i registri delle schede Galil DMC 1800 ed i tre device file creati dal driver.

Dal punto di vista hardware, il device file *card* permette l'accesso alla prima porta di I/O e quindi lo scambio di dati con le due FIFO principali. La caratteristica più importante di questo device file è il buffer software introdotto dal driver. Tale buffer, se abilitato, interviene quando si usa la chiamata di sistema `write` sul device file card: in questo caso infatti i dati passati come argomento alla funzione `write` non vengono inviati direttamente alla scheda, bensì accumulati nel buffer e inoltrati alla porta di I/O da un apposito kernel thread. I dettagli sul funzionamento del device file card e quindi anche sul buffer di scrittura saranno trattati nel capitolo 5.

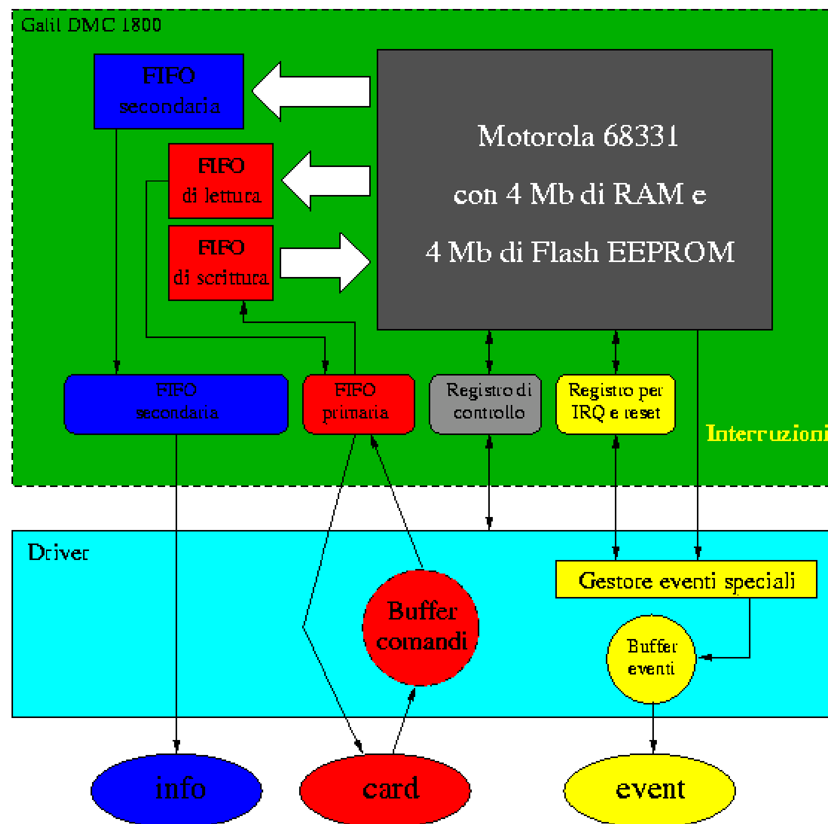

Figura 3.1: Interazione tra i registri delle schede Galil DMC 1800 ed i device file creati dal driver. I diversi colori evidenziano gli elementi coinvolti in ciascuno dei tre device file.

Il device file *event* permette invece di leggere un altro buffer introdotto dal driver: come già accennato, in questo buffer si possono trovare informazioni sulla percentuale di utilizzo del buffer dei comandi, ma anche leggere il contenuto del vettore di stato delle interruzioni. Comunque, come vedremo in dettaglio nel capitolo 6, tale informazione viene letta dal gestore di interruzione nella terza porta di I/O: è per questo che nella figura 3.1 si mostra una relazione tra una porta di I/O ed il device file event.

Infine il device file *info* permette di leggere il contenuto dell'intera FIFO secondaria e sarà discusso in dettaglio nel capitolo 7.

3.4 Il codice sorgente

L'intero codice sorgente del modulo è organizzato in 15 file scritti in linguaggio C ed un `Makefile` per la compilazione. Nella tabella 3.1 sono elencati i file che compongono il modulo, seguiti da una breve descrizione.

Nome del file	**Descrizione**
galil1800.h	Header file generico che può essere anche incluso nelle applicazioni utente.
galil1800_core.h galil1800_core.c	Definizione ed inizializzazione delle principali strutture dati. Inizializzazione del modulo e registrazione PCI.
galil1800_io.h galil1800_io.c	Driver di basso livello: implementazione della comunicazione con l'hardware.
galil1800_init.h galil1800_init.c	Funzioni di inizializzazione della scheda e di alcune parti del driver.
galil1800_sysfs.h galil1800_sysfs.c	Registrazione nel *sysfs* filesystem.
galil1800_card.h galil1800_card.c	Implementazione delle funzioni per il device file card.
galil1800_event.h galil1800_event.c	Implementazione delle funzioni per il device file event. Gestione interruzioni.
galil1800_info.h galil1800_info.c	Implementazione delle funzioni per il device file info.

Tabella 3.1: Elenco dei file sorgenti

3.5 Le strutture dati principali

Il driver definisce sette strutture dati: alcune per la normale gestione dell'hardware e dei device file e per il supporto a più schede contemporaneamen-

te, altre dedicate alla gestione dei buffer. Nella tabella 3.2 è mostrato un elenco con i nomi delle nuove strutture dati ed i file in cui sono definite.

Nome	**File**
galil1800_module_struct	galil1800_core.c
galil1800_state	galil1800_core.h
galil1800_devfile_state	galil1800_core.h
galil1800_card_wbuffer	galil1800_card.h
galil1800_event_buffer	galil1800_event.h
galil1800_irq_stack	galil1800_event.h
galil1800_event_data	galil1800.h

Tabella 3.2: Elenco delle strutture dati definite nel driver

La struttura `galil1800_module_struct` contiene alcune informazioni relative all'intero driver. Ne esiste quindi un'unica istanza che viene utilizzata solo nel file `galil1800_core.c`. Vediamo di seguito la definizione della struttura ed una descrizione di ognuno dei suoi campi:

```
struct galil1800_module_struct {
    dev_t dev;
    struct list_head cards;
    int count;
    spinlock_t lock;
    struct class sysfs_class;
};
```

`dev` è il primo device number (major e minor) assegnato al driver. `cards` è una lista di strutture `galil1800_state`, quindi rappresenta la lista delle schede presenti nel sistema; `count` è la cardinalità di tale lista. Lo spinlock `lock` serve per sincronizzare gli accessi concorrenti a questa struttura dati. Infine `sysfs_class` è la struttura che descrive la nuova classe creata nel *sysfs* filesystem.[3]

[3]Maggiori dettagli si trovano nel paragrafo 3.6.5

La struttura `galil1800_state` serve a gestire e descrivere un dispositivo Galil DMC 1800: ne esiste quindi un'istanza per ogni scheda presente nel sistema. Come vedremo nel paragrafo 3.6 queste strutture vengono allocate ed inizializzate sia durante la fase di *probe* dei dispositivi che nella funzione di *init* del modulo.

Tutte le restanti strutture dati definite nel driver sono annidate nella struttura `galil1800_state` e saranno descritte principalmente nei capitoli successivi. Vediamo ora la definizione della struttura `galil1800_state`:

```
struct galil1800_state {
   struct list_head devs;
   struct class *sysfs_class;
   struct pci_dev *dev;
   unsigned long io;
   unsigned long ctrl_reg;
   unsigned long irq_reg;
   unsigned long chan2_reg;
   unsigned int irq;
   spinlock_t hwlock;
   int index;
   char serial[GALIL1800_SERIAL_MAXLEN + 1];
   struct device_attribute dev_attr_serial;
   unsigned int open_devs;
   struct semaphore open_devs_sem;
   struct galil1800_devfile_state
              devfile_state[GALIL1800_MINOR_PER_CARD];

   struct galil1800_card_wbuffer card_wbuff;
   wait_queue_head_t card_read_wait;
   unsigned int write_delay;
   unsigned int read_delay;
   unsigned int write_timeout;
   unsigned int read_timeout;
   struct class_device_attribute cd_attr_write_delay;
```

```
    struct class_device_attribute cd_attr_read_delay;
    struct class_device_attribute cd_attr_write_timeout;
    struct class_device_attribute cd_attr_read_timeout;

    struct galil1800_event_buffer event_buff;
    struct galil1800_irq_stack irq_stack;
    struct tasklet_struct irq_tasklet;

    unsigned int fifo2_timeout;
    struct class_device_attribute cd_attr_fifo2_timeout;
};
```

Procedendo nella presentazione del driver si vedrà in dettaglio come i campi di questa struttura vengono utilizzati. Per il momento ne viene data una breve descrizione:

- `devs`: è il campo inserito nella lista `cards` presente nella struttura `galil1800_module_struct`;
- `sysfs_class`: è un puntatore all'oggetto `sysfs_class` della struttura `galil1800_module_struct`;
- `dev`: è un puntatore alla struttura `pci_dev` che descrive il dispositivo;
- `io`: indirizzo della prima porta di I/O, utilizzata per i comandi;
- `ctrl_reg`: indirizzo della seconda porta di I/O, utilizzata per il registro di controllo;
- `irq_reg`: indirizzo della terza porta di I/O, utilizzata per la lettura del vettore di stato delle interruzioni e per il registro di reset;
- `chan2_reg`: indirizzo dell'ultima porta di I/O, utilizzata per la lettura della FIFO secondaria;
- `irq`: numero di IRQ assegnato al dispositivo;

- `hw_lock`: sincronizza gli accessi sul registro di controllo e su quello di reset; i dettagli saranno descritti nel capitolo 4;
- `index`: ad ogni scheda rilevata viene assegnato un numero progressivo (a partire da zero), memorizzato in questo campo;
- `serial`: stringa per il numero seriale della scheda;
- `dev_attr_serial`: attributo del *sysfs* per esportare il numero seriale;
- `open_devs`: contiene il numero totale di device file aperti per il dispositivo;
- `open_devs_sem`: mutex per gestire l'accesso ad `open_devs`;
- `devfile_state`: array di tre strutture `galil1800_devfile_state`, una per ognuno dei tre device file (card, event, info); la struttura è così definita:

```
struct galil1800_devfile_state {
    struct cdev cdev;
    struct class_device class_dev;
    mode_t open_mode;
    struct semaphore open_sem;
    wait_queue_head_t open_wait;
};
```

Segue il significato dei campi:

- `cdev`: descrive il device file a caratteri;
- `class_dev`: permette di esportare attributi del device file nel *sysfs* (per dettagli si veda il paragrafo 3.6.5);
- `open_mode`: gestisce il modo di apertura del device file (lettura e/o scrittura);

  - `open_sem`: mutex per sincronizzare chiamate `open` concorrenti;
  - `open_wait`: le chiamate di sistema `open` su tutti i device file bloccano se la risorsa è occupata oppure un'altra `open` è in corso; questa coda d'attesa permette di implementare questa caratteristica.

Continua la descrizione della struttura con alcuni campi propri solo del device file card. Per ognuno di questi si possono trovare maggiori dettagli nel capitolo 5:

- `card_wbuff`: struttura per il buffer dei comandi;
- `card_read_wait`: coda per bloccare la chiamata di sistema `read`;
- `write_delay`: intervallo di polling per la chiamata di sistema `write`, quando usata in modo bloccante;
- `read_delay`: intervallo di polling per la chiamata di sistema `read`, quando usata in modo bloccante;
- `write_timeout`: timeout per il controllo della possibilità di scrittura sulla FIFO primaria, utilizzato solo con `write` non bloccante; maggiori dettagli nel capitolo 4;
- `read_timeout`: timeout per il controllo della possibilità di lettura dalla FIFO primaria, utilizzato solo con `read` non bloccante; maggiori dettagli nel capitolo 4;
- `cd_attr_write_delay`: esporta `write_delay` nel *sysfs* filesystem;
- `cd_attr_read_delay`: esporta `read_delay` nel *sysfs* filesystem;
- `cd_attr_write_timeout`: esporta `write_timeout` nel *sysfs* filesystem;
- `cd_attr_read_timeout`: esporta `read_timeout` nel *sysfs* filesystem.

Altri campi relativi solo al device file event, per maggiori dettagli si veda il capitolo 6:

- `event_buff`: buffer per gli eventi;
- `irq_stack`: piccolo buffer utilizzato per dati temporanei dal gestore di interruzioni;
- `irq_tasklet`: bottom half handler del gestore di interruzioni.

Infine gli ultimi campi, relativi solo al device file info, dei quali si possono trovare maggiori informazioni nel capitolo 7:

- `fifo2_timeout`: timeout per il controllo della possibilità di lettura dalla FIFO secondaria; maggiori dettagli nel capitolo 4;
- `cd_attr_fifo2_timeout`: esporta `fifo2_timeout` nel *sysfs* filesystem.

3.6 Inizializzazione

Il modulo del driver Galil all'avvio esegue la sua funzione di *init*, così come descritto nel paragrafo 2.1. Il driver sfrutta questa funzione per:

- controllare i parametri modificabili tramite il passaggio di opzioni durante il caricamento del modulo;
- inizializzare le strutture dati;
- registrare se stesso come driver per le periferiche PCI Galil DMC 1800;
- registrare un major number ed il numero necessario di minor number;
- creare una nuova classe nel *sysfs* filesystem dedicata al driver;
- creare il numero necessario di device file a caratteri e per ciascuno di essi:
  - assegnare le *file operation* per l'implementazione delle chiamate di sistema sul device file;
  - esportare alcuni attributi del device file nel *sysfs* filesystem.

Quando il modulo viene rimosso la sua funzione di *exit*:

- rimuove i device file a caratteri creati nella funzione di *init* ed i loro attributi nel *sysfs* filesystem;
- rilascia il major number ed i minor number allocati dal driver;
- rimuove la classe nel *sysfs* filesystem;
- deregistra il modulo come driver per le schede Galil DMC 1800.

È importante ricordare che, come è stato descritto nel paragrafo 2.3.1, la registrazione del driver PCI prevede anche l'implementazione di una funzione *probe* chiamata ogni qualvolta una periferica Galil DMC 1800 viene rilevata, noché di una funzione *remove* eseguita quando la periferica o il modulo vengono rimossi. Queste due funzioni e quelle di *init* ed *exit* del modulo, implementano tutte le operazioni necessarie all'inizializzazione del driver e delle schede presenti nel sistema.

Nei paragrafi successivi vedremo i dettagli di queste fasi di inizializzazione.

3.6.1 Opzioni di caricamento del modulo

Il modulo dispone di molte possibilità di configurazione sia in termini di quantità di opzioni, che di metodi utilizzabili per la configurazione. Vediamo quali sono i parametri modificabili al caricamento del modulo:

- `major`: permette di specificare un major number per il driver; senza questa opzione il driver lascia che il kernel ne assegni uno dinamicamente;
- `event_buff_sz`: numero di eventi memorizzabile nel buffer del device file event;
- `card_wbuff_sz`: dimensione in bytes del buffer dei comandi per il device file card; passando 0 il buffer viene disabilitato;

- `write_delay`, `read_delay`, `write_timeout`, `read_timeout`, `fifo2_timeout`: valori per i corrispondenti campi della struttura `galil1800_state`; in quest'ultima i valori sono memorizzati in numero di tick, ma l'utente deve impostare le opzioni usando valori in millisecondi;
- `pause_enable`: assegnando 1 lo pseudo-comando di pausa viene abilitato, con 0 viene disabilitato (default).

Le prime operazioni svolte dalla funzione di init del modulo sono il controllo e l'eventuale correzione delle opzioni passate dall'utente.

Tutti i parametri elencati, tranne il major number, possono essere modificati anche dopo il caricamento del modulo attraverso il *sysfs*, oppure con comandi della chiamata di sistema `ioctl`: i dettagli saranno spiegati in seguito, durante la trattazione degli argomenti relativi ad ogni singolo parametro.

3.6.2 Registrazione PCI

Sulla base di quanto descritto nel paragrafo 2.3.1 vediamo come viene effettuata la registrazione PCI per il driver Galil DMC 1800.

L'array di strutture `pci_device_id` è così definito:

```
struct pci_device_id galil1800_idtable[] = {
   {  .vendor = PLX_PCI_VENDOR_ID,
      .device = PLX_PCI_DEVICE_ID,
      .subvendor = GALIL_SUB_VENDOR,
      .subdevice = GALIL_SUB_DEVICE_1800,
      .class = 0,
      .class_mask = 0,
      .driver_data = 0 },
   { 0, }
};
```

Le macro usate per l'inizializzazione sono definite nel file `galil1800_io.h` e contengono i valori elencati nella tabella 1.3 di pagina 22.

L'inizializzazione della struttura `pci_driver` è stata invece effettuata in questo modo:

```
struct pci_driver galil1800_driver  = {
   .name = GALIL1800_MODULE_NAME,
   .id_table = galil1800_idtable,
   .probe = galil1800_probe,
   .remove = __exit_p(galil1800_remove),
   .suspend = NULL,
   .resume = NULL,
   .enable_wake = NULL,
   .shutdown = NULL
};
```

I dispositivi Galil DMC 1800 non hanno il supporto per il risparmio energetico, conseguentemente è stato necessario implementare solo le funzioni `probe` e `remove`. La macro `__exit_p()` restituisce `NULL` se nella configurazione del kernel si è scelto di non permettere lo scaricamento dei moduli a runtime e il dispositivo non supporta l'hotplug (come nel nostro caso); altrimenti restituisce l'indirizzo della funzione passata come argomento.

La funzione di init del modulo esegue come prima operazione un controllo delle opzioni passate dall'utente. Successivamente inizializza la struttura `galil1800_module_struct` e poi invoca la seguente funzione:

```
pci_register_driver(&galil1800_driver);
```

In questo modo viene completata la registrazione del modulo come driver PCI per le schede Galil DMC 1800; per ognuno di tali dispositivi presenti nel sistema viene chiamata la funzione di `galil1800_probe()`, così come indicato nella struttura `galil1800_driver`. Dato che l'hotplug non è supportato, la funzione `galil1800_remove()` viene chiamata solo nel caso in cui il modulo venga rimosso dal sistema. È il caso di ricordare che queste funzioni ricevono come argomento un puntatore alla struttura `pci_dev` che descrive il dispositivo.

Funzione probe

La funzione di probe, essendo chiamata una volta per ogni dispositivo, deve essere sfruttata per eseguire quelle operazioni che vanno ripetute per ogni scheda presente nel sistema. Vediamo quali sono le attività svolte dalla funzione `galil1800_probe()`:

1. controlla che sia stata assegnata una linea di interruzione;
2. controlla che la dimensione delle porte di I/O sia quella attesa;
3. alloca memoria per una struttura `galil1800_state` che descriverà il dispositivo all'interno del driver;
4. inizializza la struttura allocata e ne salva il puntatore all'interno di un apposito campo della struttura `pci_dev`; in questo modo la struttura `galil1800_state` sarà accessibile anche in altre parti del driver;
5. richiede al kernel l'utilizzo esclusivo delle porte di I/O presenti nel dispositivo; questo avviene attraverso la funzione:

   ```
   request_region(s->io, GALIL1800_REG_PCI_RESOURCE_LEN,
                                    GALIL1800_MODULE_NAME);
   ```

   dove `s` è un puntatore alla struttura `galil1800_state` e quindi `io` è l'indirizzo della prima porta di I/O; il secondo argomento è la dimensione delle porte di I/O, così come indicata nelle specifiche, mentre il terzo è il nome del modulo;
6. abilita il dispositivo attraverso la funzione:

   ```
   pci_enable_device(pcidev);
   ```

   dove `pcidev` punta alla struttura `pci_dev` del dispositivo;

7. aggiorna la lista di dispositivi presente nella struttura `galil1800_module_struct`;

8. esegue alcune operazioni di inizializzazione dell'hardware, descritte nel paragrafo 3.6.3.

Al termine di questa funzione il driver è pronto ad utilizzare il dispositivo, ma per renderlo accessibile all'utente deve eseguire ancora alcune operazioni che saranno descritte nel paragrafo 3.6.4.

Funzione remove

La funzione `galil1800_remove()` esegue le operazioni opposte a quelle della funzione `galil1800_probe()`. In particolare:

1. ottiene il puntatore alla struttura `galil1800_state` ricavandolo da quella `pci_dev`;

2. aggiorna la lista dei dispositivi;

3. disabilita il dispositivo;

4. rilascia le porte di I/O;

5. libera la memoria occupata dalla struttura `galil1800_state`.

Deregistrazione PCI

Nella funzione di exit del modulo è necessario informare il kernel che il modulo non funzionerà più come driver per le schede Galil DMC 1800. Per questo viene chiamata la funzione:

```
pci_unregister_driver(&galil1800_driver);
```

Successivamente a questa operazione, per ogni dispositivo registrato il kernel esegue la funzione `galil1800_remove()`.

3.6.3 Inizializzazione dell'hardware

Fra le ultime istruzioni di `galil1800_probe()` troviamo la seguente funzione:

```
galil1800_init_controller_probe_step(s);
```

dove `s` è un puntatore alla struttura `galil1800_state`. Tale funzione, implementata nel file `galil1800_init.c`, svolge alcune attività con lo scopo di ottenere informazioni dalla scheda e di prepararla ad essere utilizzata. Fra queste troviamo:

- reset delle FIFO primarie;
- reset dell'intero controller effettuato scrivendo un apposito comando `RS` nella FIFO primaria;
- lettura del numero di versione del firmware installato sulla scheda;
- lettura del numero di serie.

Le prime due operazioni sono particolarmente importanti poiché assicurano che al caricamento del driver l'hardware si trovi in uno stato ben determinato, ovvero con le FIFO vuote ed i parametri di funzionamento impostati ai valori di default (comando `RS`).

3.6.4 Registrazione dei device file a caratteri

Al termine della fase di probe le schede presenti nel sistema sono state individuate ed inizializzate. La fase successiva è la creazione dei device file a caratteri che, come spiegato nel paragrafo 3.3, rendono accessibile l'hardware nello spazio utente.

La registrazione dei device file a caratteri è sostanzialmente composta di due fasi:

1. il driver richiede al kernel un unico major number ed un minor number per ogni device file da registrare;

2. si registrano i necessari device file assegnandogli un device number ed informando il kernel su quali sono le funzioni che implementano le file operation per quel device file.

Come già detto, nel driver Galil per ogni dispositivo vengono creati tre device file a caratteri, ognuno con il proprio gruppo di file operation[4]: tali device file sono creati e registrati nella funzione di init del modulo, subito dopo la registrazione del driver PCI.

Richiesta dei device number

Dal campo `count` della struttura `galil1800_module_struct` è possibile sapere quanti dispositivi Galil DMC 1800 sono stati trovati ed inizializzati: il driver quindi deve richiedere `count * 3` minor number (uno per ogni device file). A tale scopo, se fra le opzioni di caricamento del modulo è stato specificato un major number, viene utilizzata la seguente funzione:

```
int register_chrdev_region(dev_t from, unsigned count,
                                         const char *name)
```

`from` è il primo device number, composto dal major richiesto e con minor 0; `count` è il totale dei minor number da richiedere, mentre name è il nome del driver. Se non si specifica un major number il driver lascia che il kernel ne assegni uno dinamicamente, usando la seguente funzione:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
                             unsigned count, const char *name)
```

dove `dev`, ovvero il primo device number, è inizializzato dal kernel con un major non ancora assegnato e minor `baseminor`, che nel nostro caso è impostato a zero.

Registrazione

Una volta ottenuti i device number necessari è possibile registrare i device file a caratteri. A questo scopo, nella funzione init del modulo viene chiamata la

[4]Vedremo nei capitoli 5, 6 e 7 come queste sono implementate.

funzione galil1800_add_cdev_for_all_cards() che, sulla base del numero di dispositivi trovati, crea tutti i device file necessari.

Il kernel di Linux descrive un device file a caratteri con una struttura chiamata cdev. Nella struttura galil1800_state è allocato un array di 3 strutture galil1800_devfile_state che, a propria volta, contiene un campo cdev: questi campi sono utilizzati nella registrazione dei device file a caratteri dalla funzione galil1800_add_cdev_for_all_cards() . Per ognuno di essi vengono chiamate le seguenti funzioni:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops)
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

cdev_init() inizializza tutti i campi di cdev. fops è una struttura che contiene tutti i puntatori alle file operation che il device file dovrà utilizzare: il driver implementa le file operation e poi prepara queste strutture per inizializzare i device file. cdev_add() effettua la registrazione di count device file, dove il primo ha device number dev.

Deregistrazione e rilascio dei device number

Nella funzione di *exit* del modulo, prima di effettuare la deregistrazione PCI, è necessario deregistrare anche i device file a caratteri: a questo punto, infatti, si è certi che i dispositivi non sono in uso.

La funzione galil1800_release_cdevs_for_all_card() esegue la deregistrazione di ogni device file utilizzando la seguente funzione del kernel:

```
void cdev_del(struct cdev *p)
```

Successivamente si rilasciano i device number con la funzione:

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

dove from è il primo device number e count è il totale di minor number allocati.

3.6.5 *sysfs* filesystem

Il *sysfs* filesystem è uno pseudo-filesystem che ha lo scopo di esportare alcune strutture dati ed informazioni del kernel nello spazio utente [27]. Tale filesystem è in genere montato nella directory `/sys` ed alcuni esempi di sue sottocartelle sono:

- `block`: contiene informazioni su ogni device file a blocchi presente nel sistema;
- `bus`: qui troviamo una directory per ogni tipo di bus, contenente a sua volta due directory:
  - `devices`: i dispositivi connessi al bus;
  - `drivers`: i driver registrati sul bus;
- `class`: qui, teoricamente, ogni driver che registra un nuovo major number dovrebbe creare una propria classe, all'interno della quale può esportare informazioni.[5]

Il driver sviluppato in questa tesi utilizza il *sysfs* filesystem per esportare alcune informazioni e rendere possibile la modifica dei parametri elencati nel paragrafo 3.6.1. A questo scopo nella funzione di init del modulo viene creata una nuova classe chiamata `galil` nella directory `/sys/class/`; questa operazione è svolta chiamando una funzione implementata in `galil1800_sysfs.c`:

```
int galil1800_sysfs_class_register(struct class *c)
```

All'interno di `/sys/class/galil/` viene poi aggiunta una directory per ogni device file registrato dal driver. La creazione di queste directory è effettuata dalla funzione `galil1800_add_cdev_for_all_cards()` dopo la registrazione del device file a caratteri.

[5] In realtà l'uso del *sysfs* filesystem è tuttora in grande evoluzione, quindi non ci sono ancora chiare regole su come i driver devono usare questo strumento. Maggiori informazioni si possono trovare in [6, 5, 20, 3, 13, 12].

Per il primo dispositivo Galil DMC 1800 sono create le directory card0, event0, info0, per il secondo card1, event1, info1, e così via: all'interno di ogni directory è presente un file per ogni opzione legata a quel particolare device a caratteri e la modifica dei parametri è effettuabile scrivendo un valore sul relativo file. Maggiori dettagli si troveranno nel seguito della trattazione.

All'interno di ogni directory relativa ad un device file il driver crea uno pseudo-file dev che permette di leggere il major ed il minor number del device file nella forma major:minor. Questa è una convenzione utile ad un particolare demone chiamato udev il cui scopo è creare dinamicamente i device file. In questo modo e con un'opportuna configurazione, il demone, al momento della creazione dei file dev, crea un relativo device file. Maggiori informazioni si possono trovare in [28].

Infine il kernel automaticamente inserisce in ogni directory dei device file un link simbolico, chiamato device, ad un'altra directory che descrive il dispositivo fisico a cui si riferisce: in questa directory il driver crea un nuovo pseudo-file chiamato serial che permette di leggere il numero seriale della scheda.

3.6.6 Completamento dell'inizializzazione durante l'apertura del device file

Per preservare alcune risorse di sitema, come ad esempio le linee di IRQ, la fase di inizializzazione viene completata durante l'apertura del primo device file. Tutte e tre le implementazioni della file operation open relative ai tre device file chiamano la seguente funzione:

```
static int galil1800_init_controller_open_step(
                                        struct galil1800_state *s)
```

Quest'ultima controlla che non sia stato ancora aperto alcun device attraverso la lettura del campo open_devs della struttura galil1800_state: se open_devs è zero, vengono eseguite le seguenti operazioni:

- svuotamento delle FIFO primarie; tale operazione viene già effettuata durante il caricamento del modulo, quindi la prima volta che il device file viene utilizzato questa operazione è superflua; d'altra parte, nei successivi utilizzi le FIFO potrebbero essere inizialmente sporche, ad esempio a causa di un crash dell'ultima applicazione che ha utilizzato il device file;
- richiesta di una linea di IRQ e registrazione di un gestore delle interruzioni; ciò avviene chiamando la seguente funzione:

```
request_irq(s->irq, galil1800_interrupt, SA_SHIRQ,
                                GALIL1800_MODULE_NAME, s);
```

  dove `s` è un puntatore alla struttura `galil1800_state`, quindi `s->irq` è la linea di IRQ letta dallo spazio di configurazione PCI; la funzione `galil1800_interrupt()` implementa il gestore di interruzioni e sarà descritta nel capitolo 6; infine `SA_SHIRQ` indica che la linea di IRQ può essere condivisa;
- inizializzazione del buffer `irq_stack` definito nella struttura `galil1800_state`;
- abilitazione del tasklet `irq_tasklet`, ugualmente definito nella struttura `galil1800_state`, che si occupa di completare la gestione delle interruzioni;
- abilitazione delle interruzioni nel dispositivo Galil DMC 1800 (vedi paragrafo 4.2).

Per liberare queste risorse, tutte le implementazioni della file operation `release` (utilizzata dalla chiamata di sistema `close`) chiamano la seguente funzione:

```
void galil1800_clean_controller_open_step(
                                      struct galil1800_state *s)
```

Quest'ultima, se il device file che si sta chiudendo era l'ultimo rimasto aperto, esegue alcune operazioni complementari a quelle appena elencate, ripristinando lo stato del driver precedente alla prima apertura di un device file.

Capitolo 4

Comunicazione con l'hardware

Questo capitolo descrive ciò che spesso viene chiamato *driver di basso livello*, ovvero la parte di un driver che si occupa della comunicazione con l'hardware.

Nei file `galil1800_io.h` e `galil1800_io.c` vengono implementate una serie di funzioni che permettono di eseguire tutte le operazioni più comuni che richiedono l'accesso ai registri delle schede Galil DMC 1800. Di seguito vedremo quali sono queste operazioni e come sono state implementate.

4.1 Strutture dati coinvolte

Tutte le funzioni che saranno descritte in questo capitolo accettano come unico argomento un puntatore alla struttura `galil1800_state` che descrive il dispositivo con il quale si vuole comunicare. In particolare i campi di tale struttura che interessano questa parte del driver sono solo i seguenti:

```
struct galil1800_state {
   ...
   unsigned long io;
   unsigned long ctrl_reg;
   unsigned long irq_reg;
   unsigned long chan2_reg;
   ...
   spinlock_t hwlock;
```

```
    ...
    unsigned int write_timeout;
    unsigned int read_timeout;
    ...
    unsigned int fifo2_timeout;
    ...
};
```

I primi quattro campi, di tipo `unsigned long`, contengono gli indirizzi delle 4 porte di I/O presenti sui dispositivi Galil DMC 1800. Seguendo quanto indicato nelle specifiche tecniche descritte in [24], i suddetti campi vengono inizializzati come mostrato in tabella 4.1.

Nome campo	**Inizializzazione**	**Utilizzo**
`io`	BAR2	Accesso alle FIFO primarie
`ctrl_reg`	BAR2 + 4	Registro di controllo
`irq_reg`	BAR2 + 8	Lettura vettore di stato IRQ e registro di reset
`chan2_reg`	BAR2 + 12	Lettura FIFO secondaria

Tabella 4.1: Inizializzazione degli indirizzi per l'accesso alle porte di I/O. Con BAR2 si intende l'indirizzo contenuto nel Base Address Register 2 dello spazio di configurazione PCI.

Nelle funzioni descritte in questo capitolo i campi `io`, `ctrl_reg`, `irq_reg` e `chan2_reg` che contengono gli indirizzi di I/O non vengono mai usati direttamente, bensì attraverso alcune macro definite nel file `galil1800_io.h` ed elencate di seguito:

```
#define GALIL1800_READ_REG(A)      ((A)->io)
#define GALIL1800_WRITE_REG(A)     GALIL1800_READ_REG(A)
#define GALIL1800_CONTROL_REG(A)   ((A)->ctrl_reg)
#define GALIL1800_IRQ_REG(A)       ((A)->irq_reg)
#define GALIL1800_RESET_REG(A)     GALIL1800_IRQ_REG(A)
#define GALIL1800_2CHAN_REG(A)     ((A)->chan2_reg)
```

Il parametro A, accettato da queste macro, deve essere un puntatore alla struttura galil1800_state; in questo modo si favorisce la leggibilità e la flessibilità del codice.

I tre campi di tipo unsigned int contengono il valore di tre timeout utilizzati nella lettura e scrittura delle FIFO; nei paragrafi successivi se ne vedranno i dettagli.

4.1.1 Utilizzo dello spinlock

Oltre ai campi già descritti, merita particolare attenzione il modo in cui viene utilizzato lo spinlock hwlock. Esso protegge tutte le operazioni di scrittura sul registro di controllo e su quello di reset. Nel paragrafo 4.2, descrivendo le operazioni di controllo della periferica, sarà spiegato il motivo che ha portato ad utilizzare un unico lock per due registri differenti.

Innanzitutto vediamo due buone ragioni che giustificano l'utilizzo di uno spinlock, piuttosto che di un semaforo:

- nella maggior parte delle situazioni dove è necessario accedere all'hardware i tempi di esecuzione sono molto importanti ed un semaforo, potendo bloccare, può introdurre maggiori ritardi rispetto ad uno spinlock;

- in questo driver si accede ai registri anche nel gestore di interruzioni; quest'ultimo è un contesto atomico dove non è possibile bloccare e quindi rilasciare il processore; utilizzare un semaforo rispettando queste condizioni sarebbe maggiormente complicato, senza produrre particolari vantaggi.

L'utilizzo dello spinlock anche nel gestore di interruzioni impone l'acquisizione del lock ed il successivo rilascio mediante le seguenti funzioni:

```
void spin_lock_irqsave(spinlock_t *lock,
                             unsigned long flags)
void spin_unlock_irqrestore(spinlock_t *lock,
                             unsigned long flags)
```

Come descritto nel paragrafo 2.3.3 la funzione `spin_lock_irqsave()` disabilita le interruzioni sul processore locale. Questo previene da una situazione di deadlock, altrimenti possibile: ad esempio, un flusso di istruzioni, dopo aver acquisito il lock, potrebbe essere interrotto dal gestore di interruzione e quest'ultimo ciclerebbe all'infinito tentando di ottenere lo spinlock. Questo tipo di problema si può risolvere solo disabilitando temporaneamente le interruzioni sul processore locale.

4.2 Operazioni di controllo

Le operazioni di controllo descritte in questo paragrafo saranno solo quelle finalizzate all'abilitazione e disabilitazione delle interruzioni ed all'utilizzo del registro di reset. Altri tipi di operazioni che riguardano la lettura e la scrittura sulle FIFO saranno trattate nei paragrafi successivi.

Di seguito sono mostrate alcune macro che vengono utilizzate durante le operazioni di accesso ai registri di controllo e di reset:

```
#define GALIL1800_FIFO_FULL_FLAG      (0x01)
#define GALIL1800_FIFO_HALF_FLAG      (0x02)
#define GALIL1800_FIFO_EMPTY_FLAG     (0x04)
#define GALIL1800_FIFO2_BUSY_FLAG     (0x08)
#define GALIL1800_FIFO2_FREEZE_FLAG   (0x10)
#define GALIL1800_IRQ_STATUS_FLAG     (0x20)
#define GALIL1800_IRQ_ENABLE_FLAG     (0x40)
#define GALIL1800_FIFO2_EMPTY_FLAG    (0x80)

#define __GALIL1800_RESET_FIFO_R      (0x02)
#define __GALIL1800_RESET_FIFO_W      (0x04)
#define __GALIL1800_RESET_HARDWARE    (0x80)
```

Le prime otto macro descrivono il significato degli otto bit del registro di controllo, così come elencate in tabella 1.4 a pagina 24. Le ultime tre contengono il valore da scrivere nel registro di reset per effettuare rispettivamente:

- lo svuotamento della FIFO di lettura,
- lo svuotamento della FIFO di scrittura,
- il reset hardware della scheda.

4.2.1 Interruzioni

Se non esplicitamente indicato le schede Galil DMC 1800 non emettono interruzioni. Queste ultime vanno esplicitamente abilitate impostando ad uno il settimo bit del registro di controllo.

Per effettuare questa operazione senza modificare il resto del registro è necessario leggere il valore di tutti gli otto bit e riscriverli modificando soltanto il valore del settimo bit: sono quindi necessarie due istruzioni, le quali rendono l'operazione non atomica. Infatti altre parti del driver, successivamente alla prima operazione di lettura potrebbero scrivere sul registro: gli effetti di questa operazione andrebbero però persi non appena viene effettuata la seconda istruzione di scrittura che si basava su un'immagine del registro acquisita precedentemente.

Il problema appena descritto è il principale motivo che giustifica l'uso dello spinlock. Infatti, acquisendo il lock ogni volta che si deve effettuare anche una sola operazione di scrittura, si assicura l'atomicità delle operazioni di modifica del registro.

Di seguito vediamo il codice che abilita e disabilita le interruzioni, facendo uso dello spinlock, così come appena descritto:

```
static inline void galil1800_enable_IRQ(struct galil1800_state *s)
{
   unsigned long flags;
   spin_lock_irqsave(&s->hwlock, flags);
   outb(inb(GALIL1800_CONTROL_REG(s)) | GALIL1800_IRQ_ENABLE_FLAG,
                                         GALIL1800_CONTROL_REG(s));
   spin_unlock_irqrestore(&s->hwlock, flags);
}
```

```
static inline void galil1800_disable_IRQ(struct galil1800_state *s)
{
   unsigned long flags;
   spin_lock_irqsave(&s->hwlock, flags);
   outb(inb(GALIL1800_CONTROL_REG(s)) & ~GALIL1800_IRQ_ENABLE_FLAG,
                                          GALIL1800_CONTROL_REG(s));
   spin_unlock_irqrestore(&s->hwlock, flags);
}
```

Meritano particolare attenzione anche quelle istruzioni da eseguire all'arrivo di un'interruzione: tale argomento non sarà però oggetto di questo capitolo, bensì verrà trattato nel capitolo 6, discutendo globalmente la gestione delle interruzioni.

4.2.2 Reset

In fase di inizializzazione del dispositivo o in particolari situazioni di errore può essere necessario effettuare operazioni di reset. A questo scopo le specifiche Galil prevedono 3 sequenze di 8 bit che possono essere scritte per effettuare altrettanti reset. Di seguito ne è mostrata l'implementazione:

```
#define GALIL1800_RESET_TIMEOUT 100
static
inline void __galil1800_reset(struct galil1800_state *s, char reset)
{
   unsigned long flags;
   spin_lock_irqsave(&s->hwlock, flags);
   outb(reset, GALIL1800_RESET_REG(s));
   spin_unlock_irqrestore(&s->hwlock, flags);
   schedule_timeout_uninterruptible(GALIL1800_RESET_TIMEOUT);
}

#define galil1800_reset_fifo_r(_s)      \
   __galil1800_reset(_s, __GALIL1800_RESET_FIFO_R)
```

```
#define galil1800_reset_fifo_w(_s)       \
   __galil1800_reset(_s, __GALIL1800_RESET_FIFO_W)

#define galil1800_reset_fifo_rw(_s)      \
   do {                                  \
      galil1800_reset_fifo_r(_s);        \
      galil1800_reset_fifo_w(_s);        \
   }while(0)

#define galil1800_reset_hardware(_s)     \
   __galil1800_reset(_s, __GALIL1800_RESET_HARDWARE)
```

La prima cosa che si può notare è che viene utilizzato lo stesso spinlock che regola la scrittura sul registro di controllo: ciò è necessario in quanto le operazioni di reset hanno effetti indiretti anche sul registro di controllo. Ad esempio, dopo un reset hardware le interruzioni vengono nuovamente disabilitate: senza l'uso del lock, se il reset avvenisse fra le due istruzioni che abilitano le interruzioni (lettura-scrittura), il controller si troverebbe in uno stato anomalo.

Infine si può notare l'uso di un timeout dopo l'istruzione di reset. Si sono effettuati alcuni test che dimostrano che le operazioni di reset possono richiedere alcuni millisecondi: in considerazione di questo e del fatto che un reset, per sua natura, non è mai effettuato in contesti *time-critical*[1], è sembrato ragionevole introdurre un breve ritardo artificiale che rende più sicura l'operazione.

4.3 Accesso alle FIFO primarie

Dalla prima porta di I/O è possibile accedere alle FIFO primarie, ovvero due buffer hardware di 512 byte dove vengono accumulati i dati scritti dal driver

[1]Con questo termine è comune intendere quelle situazioni critiche rispetto ai tempi di esecuzione, quindi dove le prestazioni hanno particolare rilevanza.

(FIFO di scrittura) e quelli generati dalla scheda e pronti per la lettura (FIFO di lettura). L'accesso a queste FIFO è puramente sequenziale ed i trasferimenti devono avvenire un byte alla volta.

Va precisato che le schede Galil DMC 1800, diversamente da altri dispositivi, non emettono interruzioni quando vi sono dati pronti per la lettura oppure c'è nuovo spazio disponibile per la scrittura. Quindi l'unico modo per sapere se è possibile leggere o scrivere sulle FIFO primarie è quello di effettuare un polling su alcuni bit del registro di controllo.

4.3.1 Lettura

Per quanto riguarda la FIFO di lettura, sappiamo che se quest'ultima è vuota, quindi non vi sono dati da leggere, il terzo bit del registro di controllo è impostato ad 1. Conseguentemente, prima della lettura di ogni singolo byte è necessario testare il valore di questo bit. A questo scopo sono state preparate due funzioni, mostrate di seguito:

```
/* Check for a byte in the Read FIFO
   Return 1 if a byte is available in the FIFO, 0 otherwise. */
static inline int __galil1800_may_read(struct galil1800_state *s)
{
   return !(inb(GALIL1800_CONTROL_REG(s)) &
                                      GALIL1800_FIFO_EMPTY_FLAG);
}

/* Wait for a byte in the Read FIFO, or until a timeout occurs.
   Return 1 if a byte is available in the FIFO, 0 otherwise.
   Interrupts MUST be enabled, or the kernel will freeze */
static inline int galil1800_may_read(struct galil1800_state *s)
{
   unsigned long timeout = jiffies + s->read_timeout;
   do {
      if (__galil1800_may_read(s))
         return 1;
```

```
  } while (time_before(jiffies,timeout));
  return 0;
}
```

Come si può vedere la prima funzione effettua semplicemente il controllo del bit, mentre la seconda lo ripete finché non è possibile leggere oppure scade un timeout di `read_timeout` ticks[2]. La lettura di ogni byte prevede il controllo del bit e, solo in caso positivo, la lettura del byte: quindi per leggere più byte in maniera consecutiva è necessario ripetere più volte queste due istruzioni. Come si può immaginare l'implementazione della chiamata di sistema `read` sul device file card ha sicuramente bisogno di poter leggere più byte in maniera consecutiva. A tale scopo fa uso della seguente funzione:

```
/* Read max len bytes from the read fifo */
ssize_t galil1800_recv(struct galil1800_state *s,
                                        char *buf, ssize_t len)
{
  int count;
  if (!galil1800_may_read(s))
    return 0;
  count = 0;
  do {
    *buf++ = inb(GALIL1800_READ_REG(s));
    ++count;
  } while(count < len && __galil1800_may_read(s));
  return count;
}
```

Si prenda in esame una situazione in cui la FIFO di lettura è vuota, ma è stato appena inviato un comando, quindi il controller sta preparando alcuni byte per la risposta. In considerazione del fatto che il controller può

[2]È necessario che durante l'uso della funzione `time_before` siano abilitate le interruzioni: il tick di sistema viene infatti incrementato all'arrivo di ogni interruzione sulla linea di IRQ numero 0. Ad interruzioni disabilitate la variabile `jiffies` non verrebbe incrementata ed il kernel andrebbe in stallo.

impiegare alcuni millisecondi a preparare ciascun byte nella FIFO, senza il timeout la funzione appena mostrata potrebbe terminare immediatamente senza leggere nulla. In questo modo l'applicazione utente sarebbe costretta a chiamare nuovamente la funzione `read`, introducendo quindi un secondo passaggio da user mode a kernel mode. Il timeout quindi riduce la possibilità che la funzione di lettura ritorni zero byte, introducendo però un piccolo ritardo quando la FIFO è effettivamente vuota e non ci sono byte in preparazione.

Un'ultima considerazione sull'uso di questo timeout: come si può notare, nella funzione `galil1800_recv()` il timeout viene utilizzato solo per il controllo del primo byte, ma non per tutti i successivi. Questa scelta evita che ci sia un ritardo al termine di ogni lettura dalla FIFO, ma comporta anche un'altra conseguenza: il device file card fornisce la possibilità di effettuare sia `read` bloccanti che non bloccanti, ma nel caso di una `read` bloccante il timeout non produce alcun effetto. Ciò deriva dal fatto che la file operation per la chiamata di sistema `read`, qualora sia usata in modo bloccante, utilizza la funzione `galil1800_recv()` solo dopo essersi accertata che vi siamo dati da leggere: in tal caso quindi il primo controllo va immediatamente a buon fine, ignorando il timeout. Al contrario, in una chiamata di sistema `read` non bloccante, il timeout produce gli effetti precedentemente descritti: infatti, in questo caso, nella file operation la funzione `galil1800_recv()` viene utilizzata senza effettuare precedenti controlli quindi, se non vi sono dati pronti, si continua a testare la FIFO per l'intero timeout. I dettagli di queste operazioni saranno descritti nel capitolo 5.

Il parametro `read_timeout` è impostato di default ad un numero di tick che corrisponde a 20 ms. L'utente può comunque in ogni istante modificare questo valore, indicando il numero di millisecondi desiderati, attraverso:

- il parametro di caricamento del modulo `read_timeout`;
- la scrittura sul file `read_timeout` presente nella directory del *sysfs* dedicata a quel particolare device file card; ad esempio per il device file card del primo dispositivo rilevato si deve accedere a:
  `/sys/class/galil/card0/read_timeout`

- un comando della chiamata di sistema `ioctl` (vedi capitolo 5).

4.3.2 Scrittura

Anche per l'accesso alla FIFO di scrittura è necessario effettuare alcuni controlli prima dell'invio dei byte che, anche in questo caso, devono essere scritti uno alla volta.

Dal registro di controllo si possono trarre due informazioni sullo stato della FIFO di scrittura:

- se il primo bit è impostato ad 1, allora la FIFO è piena e nessun byte può essere scritto;
- se il secondo bit è impostato ad 1, allora la FIFO è piena per più di metà, ovvero vi sono almeno 256 byte occupati sui 512 disponibili.

Unendo queste due informazioni ci si può trovare in 3 situazioni diverse:

- se il primo bit vale 1 anche il secondo bit vale obbligatoriamente 1 e non si può scrivere alcun byte;
- se il primo bit vale 0, la FIFO non è completamente piena quindi:
  - se il secondo bit vale 1, la FIFO è piena per più di metà, ma si può sicuramente scrivere almeno un byte; per il successivo si dovrà di nuovo effettuare il controllo;
  - se il secondo bit vale 0 nella FIFO ci sono meno di 256 byte occupati, quindi, secondo le specifiche Galil, possono essere scritti 255 byte senza rieffettuare alcun controllo.

Le due funzioni mostrate di seguito effettuano i test su questi bit di controllo:

```
/* Check for a free slot in the Write FIFO, without timeout.
   Return:
     2 if FIFO is less than half full (at least 255 free slots),
     1 if a slot is available in the FIFO, 0 otherwise. */
```

```
static inline int __galil1800_may_write(struct galil1800_state *s)
{
   char t = inb(GALIL1800_CONTROL_REG(s));
   if(!(t & GALIL1800_FIFO_HALF_FLAG))
      return 2;
   if(!(t & GALIL1800_FIFO_FULL_FLAG))
      return 1;
   return 0;
}

/* Wait for a free slot in the Write FIFO,
   or until a timeout occurs. Return:
     2 if FIFO is less than half full (at least 255 free slots),
     1 if a slot is available in the FIFO, 0 otherwise.
   Local interrupts MUST be enabled */
static inline int galil1800_may_write(struct galil1800_state *s)
{
   unsigned long timeout = jiffies + s->write_timeout;
   char t;
   do {
      t = inb(GALIL1800_CONTROL_REG(s));
      if (!(t & GALIL1800_FIFO_HALF_FLAG))
         return 2;
      if (!(t & GALIL1800_FIFO_FULL_FLAG))
         return 1;
   } while (time_before(jiffies,timeout));
   return 0;
}
```

Come si può notare, anche in questo caso sono presenti due funzioni che controllano lo stato della FIFO di scrittura, con e senza un timeout. Il motivo dell'introduzione del timeout è molto simile al caso della lettura: prima di affermare che non si può scrivere alcun byte nella FIFO è vantaggioso attendere un breve timeout. Vediamo in dettaglio la funzione per scrivere

una sequenza di byte, usata nell'implementazione della chiamata di sistema `write`:

```
/* write max len bytes to the write fifo */
ssize_t galil1800_send(struct galil1800_state *s,
                              const char *buf, size_t len)
{
   size_t count = len;
   int i, burst;
   const char *p = buf;
   burst = galil1800_may_write(s);
   if(!count || !burst)
      return 0;
   if(burst == 2) {
      do {
         burst = count > GALIL1800_HALF_FULL_WRITE_BURST ?
                        GALIL1800_HALF_FULL_WRITE_BURST : count;
         for (i=0; i < burst; ++i, ++p)
            outb_p(*p, GALIL1800_WRITE_REG(s));
         count -= burst;
      } while(count && __galil1800_may_write(s) == 2);
   }
   while(count && __galil1800_may_write(s)) {
      outb(*p, GALIL1800_WRITE_REG(s));
      --count;
      ++p;
   }
   return len-count;
}
```

Come si può notare l'uso del timeout è equivalente a quello della funzione `galil1800_recv()`. Per il resto questa funzione risulta maggiormente complessa poiché, quando possibile, sfrutta la possibilità di scrivere fino a 255 byte senza rieffettuare controlli. Tutte le considerazioni precedentemente

espresse per la variabile `read_timeout`, valgono anche per `write_timeout`, anche quelle relative alle possibilità di modifica ed all'implementazione di un I/O bloccante e non bloccante.

4.4 Lettura della FIFO secondaria

In ultimo resta da discutere l'accesso alla FIFO secondaria. Si ricorda che quest'ultimo è un buffer di dimensioni variabili a seconda del numero di assi che la scheda è in grado di controllare. Il dispositivo, qualora esplicitamente richiesto dall'utente attraverso il comando `DU`, può aggiornare tale FIFO ad intervalli regolari con informazioni sui vari assi. Per dettagli sull'abilitazione e sul significato dei dati letti si può far riferimento rispettivamente a [25] e [24].

Il quarto, quinto e ottavo bit del registro di controllo sono dedicati alla FIFO secondaria:

- se il quarto bit è 1, la FIFO è in aggiornamento e non è possibile accedervi;
- scrivendo 1 sul quinto bit, si congela la FIFO interrompendone l'aggiornamento;
- scrivendo 0 sul quinto bit, si sblocca la FIFO riabilitandone l'aggiornamento;
- se l'ottavo bit è impostato ad 1 la FIFO è vuota; questo si verifica in due casi:
  - l'aggiornamento della FIFO è disabilitato (comando `DU0`),
  - la FIFO è stata congelata e si sono letti tutti i byte disponibili.

Sulla base di quanto appena detto e seguendo le specifiche fornite dalla Galil, la procedura per la lettura della FIFO secondaria è la seguente:

- congelare la FIFO;

- attendere che l'aggiornamento della FIFO sia completato;
- leggere tutti i byte della FIFO finché l'ottavo bit del registro di controllo non vale zero; la porta di I/O della FIFO secondaria è larga 4 byte, quindi si può leggere un `long` ad ogni passo;
- riattivare la FIFO.

Ad ogni lettura è necessario ricontrollare il valore dell'ottavo bit del registro di controllo ed, anche in questo caso, è stato introdotto un timeout per dare il tempo al controller di preparare i dati. Il timeout è memorizzato nel campo `fifo2_timeout` della struttura `galil1800_state` ed ha le stesse caratteristiche di quelli già discussi per le FIFO primarie.

Vediamo le funzioni che operano sui bit del registro di controllo dedicati alla FIFO secondaria:

```
/* Check for a byte in the secondary FIFO.
   Return 1 if a byte is available in the 2nd FIFO, 0 otherwise */
static
inline int __galil1800_may_read_FIFO2(struct galil1800_state *s)
{
   return !( inb(GALIL1800_CONTROL_REG(s))
                               & GALIL1800_FIFO2_EMPTY_FLAG );
}


/* Wait for a byte in the secondary FIFO, or until a timeout occurs.
   Return 1 if a byte is available in the 2nd FIFO, 0 otherwise.
   Interrupts MUST be enabled, or the kernel will freeze */
static
inline int galil1800_may_read_FIFO2(struct galil1800_state *s)
{
   unsigned long timeout = jiffies + s->fifo2_timeout;
   do {
      if (__galil1800_may_read_FIFO2(s))
         return 1;
```

```
   } while (time_before(jiffies,timeout));
   return 0;
}

/* Freeze the secondary communication channel */
static inline void galil1800_freeze_FIFO2(struct galil1800_state *s)
{
   unsigned long flags;
   spin_lock_irqsave(&s->hwlock, flags);
   outb(inb(GALIL1800_CONTROL_REG(s)) | GALIL1800_FIFO2_FREEZE_FLAG,
                                         GALIL1800_CONTROL_REG(s));
   spin_unlock_irqrestore(&s->hwlock, flags);
}

/* Unfreeze the secondary communication channel */
static
inline void galil1800_unfreeze_FIFO2(struct galil1800_state *s)
{
   unsigned long flags;
   spin_lock_irqsave(&s->hwlock, flags);
   outb(
    inb(GALIL1800_CONTROL_REG(s)) & ~GALIL1800_FIFO2_FREEZE_FLAG,
                                        GALIL1800_CONTROL_REG(s) );
   spin_unlock_irqrestore(&s->hwlock, flags);
}

/*Wait until the controller ends transferring data in the 2nd FIFO*/
static
inline void galil1800_wait_transfer_FIFO2(struct galil1800_state *s)
{
   while(inb(GALIL1800_CONTROL_REG(s)) & GALIL1800_FIFO2_BUSY_FLAG);
}
```

Il driver, attraverso il device file info, permette di leggere la FIFO seconda-

ria facendo uso della chiamata di sistema `read`. Quest'ultima è implementata solo in modo non bloccante e, ad ogni esecuzione, tenta di leggere l'intera FIFO sequenzialmente, facendo uso di una funzione che segue la procedura precedentemente indicata attraverso l'uso delle funzioni appena mostrate. Vediamo l'implementazione di tale funzione:

```
ssize_t galil1800_read_FIFO2(struct galil1800_state *s,
                                    char *buf, size_t len)
{
   char *pc = buf;
   ssize_t count = len;
   unsigned long *pl = (unsigned long*)buf;
   size_t adjust = len%4;
   count -= adjust;
   if (!__galil1800_may_read_FIFO2(s))
      return 0; /* if secondary fifo is disabled */
   galil1800_freeze_FIFO2(s);
   galil1800_wait_transfer_FIFO2(s);
   do {
      *pl++ = inl(GALIL1800_2CHAN_REG(s));
      count-=4;
   } while (count && galil1800_may_read_FIFO2(s));
   pc = (char*)pl;
   count += adjust;
   while(count && galil1800_may_read_FIFO2(s)) {
      *pc++ = inb(GALIL1800_2CHAN_REG(s));
      --count;
   }
   galil1800_unfreeze_FIFO2(s);
   return len-count;
}
```

Capitolo 5

Strategie di memorizzazione dei comandi

In questo capitolo saranno discusse le caratteristiche del device file card ed i dettagli della sua implementazione. Come si è già detto, per i device file è compito del driver implementare le *file operation*, ovvero quella parte delle chiamate di sistema dipendente dall'hardware. In questo capitolo vedremo quali chiamate di sistema sono supportate dal device file card e come queste sono state realizzate.

L'aspetto più rilevante sarà comunque la presentazione del buffer di memorizzazione dei comandi: infatti, come vedremo, sono state realizzate due implementazioni della chiamata di sistema `write`, con e senza bufferizzazione.

5.1 Introduzione al device file card

Il device file card permette l'accesso alle FIFO primarie dei dispositivi Galil DMC 1800, quindi, attraverso di esso è possibile:

- inviare comandi al controller,
- leggere le risposte prodotte dopo ogni comando.

Inoltre attraverso il device file card si possono effettuare alcune operazioni speciali come, ad esempio, il reset delle FIFO oppure la lettura del numero di serie della scheda.

Per realizzare le funzionalità appena elencate sono state implementate le seguenti file operation:

- `open`,
- `release`,
- `read`,
- `write`,
- `ioctl`.

In particolare, la chiamata di sistema `write` è stata sviluppata in due versioni: con e senza buffer. Nel primo caso si tenta di scrivere i comandi direttamente nella FIFO hardware, finché vi è spazio sufficiente. Nella versione bufferizzata i comandi inviati alla funzione `write` vengono scritti temporaneamente su un buffer software: un apposito kernel thread, realizzato attraverso una workqueue, si occupa di copiare il contenuto del buffer nella FIFO.

Quanto appena detto sarà dettagliatamente discusso nei paragrafi successivi, dove saranno trattate, una alla volta, tutte le file operation implementate.

5.1.1 Opzioni

Abbiamo già visto che il driver dispone di diversi parametri configurabili attraverso opzioni di caricamento del modulo, *sysfs* filesystem e comandi della chiamata `ioctl`. In tabella 5.1 sono riassunte le opzioni riguardanti il device file card ed i possibili metodi di configurazione.

I parametri `write_delay`, `read_delay`, `write_timeout` e `read_timeout` devono essere indicati in millisecondi; apposite funzioni, definite nel file `galil1800_io.h`, effettuano la conversione in tick, ovvero l'unità di misura in cui sono memorizzati nei rispettivi campi della struttura `galil1800_state`.

Parametro	Descrizione	Modulo	sysfs	ioctl
card_wbuff_sz	Dimensione in byte del buffer dei comandi (0 per disabilitare)	Si	Si*	No
write_delay	Intervallo di polling durante una write bloccante	Si	Si	Si
read_delay	Intervallo di polling durante una read bloccante	Si	Si	Si
write_timeout	Timeout per il controllo della possibilità di scrittura (solo per write non bloccante)	Si	Si	Si
read_timeout	Timeout per il controllo della possibilità di lettura (solo per read non bloccante)	Si	Si	Si
pause_enable	Pseudo comando di pausa: 1 abilita, 0 disabilita	Si	Si*	No

Tabella 5.1: Opzioni del device file card (* solo se non ci sono device file aperti)

Selezione delle file operation

Nella fase di registrazione del device file a caratteri (si veda paragrafo 3.6.4) deve essere indicato il gruppo di file operation da utilizzare. Poiché la funzione `write` è diversa a seconda che il buffer dei comandi è attivo o meno, sono state definite due strutture `file_operations`:

```
struct file_operations galil1800_card_unbuffered_fops = {
   .owner = THIS_MODULE,
   .read = galil1800_card_read,
   .write = galil1800_card_unbuffered_write,
   .unlocked_ioctl = galil1800_card_unlocked_ioctl,
   .open = galil1800_card_open,
   .release = galil1800_card_release,
   .llseek = no_llseek
};
```

```
struct file_operations galil1800_card_buffered_fops = {
   .owner = THIS_MODULE,
   .read = galil1800_card_read,
   .write = galil1800_card_buffered_write,
   .unlocked_ioctl = galil1800_card_unlocked_ioctl,
   .open = galil1800_card_open,
   .release = galil1800_card_release,
   .llseek = no_llseek
};
```

Al momento della registrazione del device file a caratteri viene valutato il valore dell'opzione `card_wbuff_sz` al fine di scegliere l'adeguata struttura `file_operations`. In ogni caso, è sempre possibile sostituire il puntatore a questa struttura e ciò è quanto avviene ogni volta che si cambia lo stato di abilitazione del buffer dei comandi. Se ne può trovare un esempio nella funzione che viene chiamata quando si modifica `card_wbuff_sz` attraverso il *sysfs*; tale funzione è definita nel file `galil1800_card.c` ed è mostrata di seguito:

```
static ssize_t galil1800_sysfs_card_wbuff_sz_store(
          struct class_device *cd, const char *buf, size_t count)
{
   struct galil1800_devfile_state *df_s =
      (struct galil1800_devfile_state *)(cd->class_data);
   struct galil1800_state *s = container_of(df_s,
                           struct galil1800_state,
                           devfile_state[GALIL1800_CARD_DEVICE]);
   if(down_interruptible(&df_s->open_sem))
      return -ERESTARTSYS;
   if(df_s->open_mode) {
      up(&df_s->open_sem);
      printk(KERN_ERR PFX "card_wbuff_sz not modified."
                          " There are files opened\n");
      return -EBUSY;
   }
```

```
    if(count>10 || strlen(buf)!=count) {
       up(&df_s->open_sem);
       printk(KERN_ERR PFX "card_wbuff_sz not modified."
                           " Too much characters\n");
       return -EINVAL;
    }
    sscanf(buf, "%u", &s->card_wbuff.size);
    galil1800_card_wbuff_size_check(&s->card_wbuff.size);
    if(s->card_wbuff.size==0)
       df_s->cdev.ops = &galil1800_card_unbuffered_fops;
    else
       df_s->cdev.ops = &galil1800_card_buffered_fops;
    up(&df_s->open_sem);
    return count;
}
```

Come si può notare, prima di effettuare la modifica della dimensione del buffer, si controlla che il device file non sia già aperto: in tal modo si è certi che il buffer non è in uso. Altri dettagli che permettono di comprendere meglio il funzionamento di questa funzione saranno discussi più avanti in questo capitolo.

Dalle strutture `file_operations` precedentemente mostrate è possibile vedere quali sono le funzioni che implementano le varie file operation; tali funzioni sono tutte definite nel file `galil1800_card.c`.

5.2 Apertura e chiusura del device file

L'implementazione della file operation `open` è particolarmente importante in quanto svolge alcune operazioni fondamentali per il corretto funzionamento del driver:

1. ottiene un puntatore alla struttura `galil1800_state` che descrive il dispositivo e lo rende disponibile per tutte le altre file operation;

2. controlla il modo di apertura del device file;
3. qualora sia attivo, inizializza il buffer dei comandi;
4. se è il primo device file ad essere aperto, completa l'inizializzazione del driver (si veda paragrafo 3.6.6 a pagina 74).

Di seguito viene mostrato l'inizio della funzione `galil1800_card_open()` che implementa la file operation per la chiamata di sistema `open`:

```
static
int galil1800_card_open(struct inode *inode, struct file *filp)
{
   int ret;
   struct galil1800_state *s;
   struct galil1800_devfile_state *df_s;
   s = container_of(inode->i_cdev, struct galil1800_state,
                    devfile_state[GALIL1800_CARD_DEVICE].cdev);
   filp->private_data = s;
   df_s = &(s->devfile_state[GALIL1800_CARD_DEVICE]);
   ... ...
```

Come si può vedere le prime istruzioni ottengono un puntatore alla struttura `galil1800_state`; ciò avviene basandosi sulle seguenti considerazioni:

- durante la registrazione del device file a caratteri un puntatore all'oggetto `cdev` contenuto nella struttura `galil1800_state` viene salvato nel campo `i_cdev` della struttura `inode`;
- avendo a disposizione questo puntatore la macro `container_of()` è in grado di calcolare il valore del puntatore alla struttura che contiene l'oggetto `cdev`.

Poiché solo le funzioni `open` e `release` ricevono fra gli argomenti un puntatore al descrittore dell'`inode` è necessario salvare il puntatore alla struttura `galil1800_state` nel campo `private_data` della struttura `file`; quest'ultima sarà infatti accessibile da tutte le file operation.

La parte centrale della funzione `galil1800_card_open()` si occupa di garantire che, in ogni istante, esista un solo processo che ha aperto il file con permesso di scrittura ed un solo processo che ha permesso di lettura; questo comporta che un processo che apre il device file faccia un attento uso dei flag della chiamata di sistema `open`, quali `O_RDONLY`, `O_WRONLY` e `O_RDWR`. Qualora un processo apra il device file specificando il flag `O_RDWR` nessun altro potrà aprire il file; infine un massimo di due processi potranno aprire contemporaneamente il device file, purché uno usi il flag `O_RDONLY`, mentre l'altro `O_WRONLY`.

Questa scelta è fondata sulla base di due importanti considerazioni:

- la restrizione ad un massimo di un processo per ogni tipo di accesso (lettura o scrittura) semplifica molte operazioni che fanno uso di alcune risorse del driver; ad esempio, in questo modo si è certi che non possono esserci due funzioni `write` in esecuzione contemporaneamente; lo stesso vale per la chiamata `read`;

- per le caratteristiche dei dispositivi Galil DMC 1800 non è necessario che più di un processo acceda in scrittura e che più di un processo acceda in lettura; infatti, nella maggior parte delle situazioni, questo complicherebbe solo la gestione del dispositivo. Ad esempio, si immaginino due funzioni `write` che inviano contemporaneamente comandi: sarebbe molto complicato separare i flussi di byte e stabilire il corretto ordine di scrittura dei comandi.

Da ciò si deduce che questa limitazione imposta dal driver non crea alcuna evidente difficoltà nell'uso dei controller Galil DMC 1800.

Se un processo tenta di aprire il device file card, ma quest'ultimo è già stato aperto nel modo richiesto (lettura o scrittura), allora la funzione `open` blocca il processo finché il device file non viene chiuso e quindi il nuovo processo può completarne l'apertura. La file operation `galil1800_card_open()` è stata quindi implementata in modo bloccante. In ogni caso, se non si desidera che la chiamata di sistema `open` possa bloccare il processo, è suffi-

ciente specificare in quest'ultima il flag O_NONBLOCK: in tal modo la funzione galil1800_card_open() si comporta in modo non bloccante.

I controlli sul modo di apertura del device file sono effettuati facendo uso dei campi della struttura galil1800_devfile_state descritta nel paragrafo 3.5. In particolare la variabile open_mode viene utilizzata per tenere traccia del modo in cui è stato aperto il file; open_sem è il mutex che protegge l'accesso ad open_mode; open_wait è la waitqueue per bloccare il processo che effettua la open, quando il device file è già stato aperto. Di seguito possiamo vedere il ciclo che blocca la funzione open quando il device file è occupato:

```
if(down_interruptible(&df_s->open_sem))
   return -ERESTARTSYS;
while (df_s->open_mode & filp->f_mode) {
   if (filp->f_flags & O_NONBLOCK) {
      up(&df_s->open_sem);
      return -EAGAIN;
   }
   up(&df_s->open_sem);
   if(wait_event_interruptible(df_s->open_wait,
                  !(df_s->open_mode & filp->f_mode)))
      return -ERESTARTSYS;
   if(down_interruptible(&df_s->open_sem))
      return -ERESTARTSYS;
}
...
```

Qualora il device file sia aperto anche con permesso di scrittura, la funzione galil1800_card_open() inizializza il buffer dei comandi attraverso un'istruzione simile alla seguente:

```
if(filp->f_mode & FMODE_WRITE)
   init_galil1800_card_wbuffer_open_dev(&s->card_wbuff);
```

Tale funzione sarà discussa successivamente in questo capitolo; al momento si precisa soltanto che, se il buffer dei comandi è disabilitato, la funzione termina senza eseguire alcuna istruzione.

Infine, come spiegato nel paragrafo 3.6.6, viene chiamata la funzione `galil1800_init_controller_open_step()`, che incrementa il contatore di device file aperti e, qualora si stia aprendo il primo device file per il dispositivo di riferimento, completa alcune fasi dell'inizializzazione.

Come di consueto, la funzione `galil1800_card_release()`, implementando la file operation per la chiamata di sistema `close`, esegue alcune istruzioni opposte a quelle appena discusse. In particolare:

- se necessario, libera le risorse occupate dal buffer dei comandi chiamando la funzione `clean_galil1800_card_wbuffer_close_dev()`;
- esegue la funzione `galil1800_clean_controller_open_step()` (si veda paragrafo 3.6.6);
- utilizzando il mutex `open_sem`, adegua il campo `open_mode` nella struttura `galil1800_devfile_state`.

Di seguito è riportata l'intera funzione:

```
static int galil1800_card_release(struct inode * inode,
                                      struct file * filp)
{
   struct galil1800_state *s =
         (struct galil1800_state *)filp->private_data;
   struct galil1800_devfile_state *df_s =
         &(s->devfile_state[GALIL1800_CARD_DEVICE]);
   if(down_interruptible(&df_s->open_sem))
      return -ERESTARTSYS;
   if(filp->f_mode & FMODE_WRITE)
      clean_galil1800_card_wbuffer_close_dev(&s->card_wbuff);
   galil1800_clean_controller_open_step(s);
   df_s->open_mode &= (~filp->f_mode) & (FMODE_READ|FMODE_WRITE);
```

```
    up(&df_s->open_sem);
    wake_up_interruptible_sync(&df_s->open_wait);
    return 0;
}
```

5.3 Comandi della chiamata di sistema `ioctl`

La funzione `galil1800_card_unlocked_ioctl()` implementa la file operation per la chiamata di sistema `ioctl` nel device file card. La tabella 5.2 mostra la lista delle macro dei comandi supportati ed il relativo tipo di dato richiesto nell'argomento della chiamata `ioctl`.

Macro per il comando	**Argomento**
GALIL1800_IOCTL_RESET_HARDWARE	
GALIL1800_IOCTL_RESET_FIFO_R	
GALIL1800_IOCTL_RESET_FIFO_W	
GALIL1800_IOCTL_RESET_FIFO_RW	
GALIL1800_IOCTL_UNSAFE_WRITE_STOP	
GALIL1800_IOCTL_GET_SERIAL_NUMBER	char*
GALIL1800_IOCTL_SET_PAUSE	unsigned long
GALIL1800_IOCTL_SET_WRITE_DELAY	unsigned int
GALIL1800_IOCTL_SET_READ_DELAY	unsigned int
GALIL1800_IOCTL_SET_WRITE_TIMEOUT	unsigned int
GALIL1800_IOCTL_SET_READ_TIMEOUT	unsigned int

Tabella 5.2: Lista dei comandi per la funzione `ioctl` del device file card.

Il comando `GALIL1800_IOCTL_RESET_HARDWARE` esegue un reset delle FIFO primarie e poi un reset hardware della scheda. Tale procedura deve essere utilizzata solo qualora il controller sia in uno stato di errore tale da non rispondere più al comando `RS`, previsto per il normale reset [25, 24].

I successivi 3 comandi di reset eseguono rispettivamente lo svuotamento della FIFO di lettura, di quella di scrittura oppure di entrambe.

Il comando GALIL1800_IOCTL_UNSAFE_WRITE_STOP può essere utilizzato in particolari situazioni di errore dove è necessario interrompere immediatamente il processamento dei comandi da parte della scheda. Questo comando fa uso della funzione galil1800_unsafe_write_stop() definita nel file galil1800_card.c la quale esegue le seguenti operazioni:

- se il buffer dei comandi è attivo, lo svuota;
- svuota le FIFO di lettura e di scrittura;
- attende un intervallo di tempo prefissato e svuota nuovamente le FIFO; questa operazione è necessaria poiché altre parti del driver potrebbero nello stesso istante aver scritto altri comandi nella FIFO di scrittura.

Questa operazione è assolutamente priva di ogni controllo per il mantenimento della coerenza dei dati, quindi va utilizzata solo in particolari situazioni di errore; successivamente è consigliato di ricaricare il modulo del driver.

Per ottenere il numero seriale del dispositivo Galil DMC 1800 si può utilizzare il comando GALIL1800_IOCTL_GET_SERIAL_NUMBER; l'argomento passato alla funzione ioctl deve essere un puntatore ad una stringa di almeno GALIL1800_SERIAL_MAXLEN+1[1] caratteri.

Elencando le caratteristiche del driver è stato introdotto uno pseudo-comando di pausa il cui scopo è quello di interrompere la scrittura dei comandi per un numero di millisecondi selezionabile. L'introduzione di una pausa è effettuabile attraverso il comando GALIL1800_IOCTL_SET_PAUSE; l'argomento da passare alla funzione ioctl è un unsigned long con il quale si può indicare la durata della pausa in millisecondi.[2]

Gli ultimi 4 comandi permettono di modificare i parametri write_delay, read_delay, write_timeout e read_timeout già descritti precedentemente. L'argomento da passare alla funzione ioctl è un unsigned int contenente un valore in millisecondi.

[1] Macro definita nel file galil1800.h (può essere incluso nelle applicazioni utente).

[2] Per utilizzare questo comando è necessario aprire il device file con permesso di scrittura. Maggiori dettagli nel paragrafo 5.6.

5.4 Lettura

Utilizzando la chiamata di sistema `read` sul device file card si accede al contenuto della FIFO di lettura, in cui il controller Galil DMC 1800 memorizza le risposte ai comandi eseguiti.

Vediamo di seguito il prototipo della funzione e la prima parte di definizione delle variabili:

```
static ssize_t galil1800_card_read(struct file * filp,
               char __user * buf,size_t size, loff_t *poff)
{
   struct galil1800_state *s =
      (struct galil1800_state *) filp->private_data;
   char *_buf;
   ssize_t count, request = (size < GALIL1800_READ_FIFO_SIZE ?
                              size : GALIL1800_READ_FIFO_SIZE);
   int rc;
   ...
```

La prima istruzione ottiene il puntatore alla struttura `galil1800_state`, inserito nella struttura `file` dalla funzione `galil1800_card_open()`. La variabile `request` determina il massimo numero di byte che saranno letti dal dispositivo.

Questa chiamata di sistema è stata implementata in modo bloccante: ciò significa che se non viene utilizzato il flag `O_NONBLOCK` e non ci sono dati pronti per la lettura la funzione `galil1800_card_read()` blocca il processo chiamante. Poiché le schede Galil DMC 1800 non emettono interruzioni quando viene modificato il contenuto delle FIFO, l'unico modo per capire se ci sono nuovi dati da leggere è quello di effettuare un polling sul registro di controllo. Vediamo la parte di codice che realizza quanto appena descritto:

```
   while(__galil1800_may_read(s)==0) {
      if(filp->f_flags & O_NONBLOCK) {
         if(galil1800_may_read(s))
            break;
```

```
            else
                return -EAGAIN;
        }
        rc = wait_event_interruptible_timeout(s->card_read_wait,
                        __galil1800_may_read(s)>0, s->read_delay);
        if(rc == -ERESTARTSYS)
            return rc;
    }
```

La funzione `__galil1800_may_read()` è già stata discussa nel paragrafo 4.3.1: senza attendere il timeout `read_timeout`, restituisce 1 se ci sono dati da leggere, 0 altrimenti. Se la FIFO è vuota e non è stato utilizzato `O_NONBLOCK`, ogni `read_delay` ticks viene riletto il registro di controllo: la funzione esce dal ciclo solo quando ci sono dati da leggere oppure il processo riceve un segnale. Il valore di `read_delay` è impostato di default ad un numero di tick equivalenti a 20 ms, ma, come già descritto, è facilmente configurabile. Se invece la FIFO è vuota, ma è stata richiesta una chiamata non bloccante, prima di uscire dalla funzione si ricontrolla la possibilità di lettura mediante la funzione `galil1800_may_read()`; quest'ultima attende l'arrivo di almeno un byte per `read_timeout` ticks, come discusso nel paragrafo 4.3.1. A questo punto è evidente come il parametro `read_delay` intervenga solo per le chiamate bloccanti, mentre `read_timeout` solo per quelle non bloccanti.

Quando le funzioni `__galil1800_may_read()` o `galil1800_may_read()` restituiscono un valore positivo si è certi che almeno un byte è presente nella FIFO. Non potendo determinare il numero di byte che potranno essere letti viene allocata un'area di memoria di `request` byte; come già mostrato, quest'ultima variabile viene inizializzata nelle prime istruzioni della funzione `galil1800_card_read()`. Vediamone ora la parte finale:

```
    _buf = kmalloc(request, GFP_KERNEL);
    if (!_buf) {
        printk(KERN_ERR PFX "card read: alloc _buf failed\n");
        return -ENOMEM;
    }
```

```
    count = galil1800_recv(s, _buf, request);
    rc = 0;
    if (count)
        rc = copy_to_user(buf, _buf, count);
    kfree(_buf);
    return (rc > 0 ? -EFAULT : count);
```

Dopo aver allocato memoria, saranno letti i primi `request` byte pronti nella FIFO; a tale scopo è necessario accedere alla prima porta di I/O e questo avviene tramite la funzione `galil1800_recv()`, descritta nel paragrafo 4.3.1. Tali dati vengono quindi copiati nello spazio di memoria utente indirizzato dal puntatore `buf`; infine la memoria allocata viene liberata e viene restituito il numero di byte letti.

5.5 Scrittura non bufferizzata

Attraverso la chiamata di sistema `write` sul device file card è possibile inviare comandi ai controller Galil DMC 1800. Tali comandi devono quindi essere scritti sulla FIFO primaria, accessibile tramite la prima porta di I/O. Attraverso le opzioni del driver è possibile attivare un buffer software che si interpone tra la chiamata di sistema `write` e l'effettiva fase di scrittura dei dati sulla FIFO. Più avanti in questo capitolo vedremo i dettagli relativi alla scrittura bufferizzata, mentre in questo paragrafo si discute l'implementazione della chiamata `write` con buffer disabilitato.

La funzione `galil1800_card_unbuffered_write()` riceve la sequenza di byte da inviare al dispositivo e tenta di scriverla nella FIFO primaria. Questa file operation è implementata solo in modo non bloccante quindi, quando la FIFO è piena, restituisce il controllo riportando il numero di byte scritti. Vediamo l'implementazione di tale funzione:

```
static ssize_t galil1800_card_unbuffered_write(struct file *filp,
                const char __user *buf, size_t size, loff_t *poff)
{
```

```
    struct galil1800_state *dev =
        (struct galil1800_state *)filp->private_data;
    char *_buf;
    ssize_t request;
    int rc=0;
    if (!size)
        return 0;
    if (!buf || size < 0)
        return -EINVAL;
    request = (size < GALIL1800_WRITE_FIFO_SIZE ?
                    size : GALIL1800_WRITE_FIFO_SIZE);
    _buf = kmalloc(request, GFP_KERNEL);
    if (!_buf) {
        printk(KERN_ERR PFX "card write: alloc _buf failed\n");
        return -ENOMEM;
    }
    if (copy_from_user(_buf, buf, request)) {
        rc = -EFAULT;
        goto out;
    }
    rc = galil1800_send(dev, _buf, request);
out:
    kfree(_buf);
    return rc;
}
```

Le prime operazioni eseguono un controllo degli argomenti e fissano il massimo numero di byte che potranno essere scritti (variabile `request`). Dopodiché viene allocata la memoria necessaria e prelevati i dati dallo spazio utente. Infine attraverso la funzione `galil1800_send()`, discussa nel paragrafo 4.3.2, si tenta di scrivere i dati nella FIFO del dispositivo. Quando quest'ultima è piena la funzione `galil1800_send()` termina, restituendo il numero di byte letti. Tale funzione, per il controllo sulla possibilità di scrittura del primo byte, usa la funzione `galil1800_may_write()`, quindi, quando la FIFO è

piena continua a provare per `write_timeout` ticks; per tutti gli altri byte fa invece uso della funzione `__galil1800_may_write()`, priva di timeout. Ulteriori dettagli si trovano nel paragrafo 4.3.2.

Come ultima considerazione si può notare che il flag `O_NONBLOCK` viene completamente ignorato e questa chiamata è sempre non bloccante, anche se non si riesce a scrivere neanche il primo byte.

5.6 Il buffer dei comandi

La possibilità di bufferizzare la scrittura è la caratteristica principale di questo device file: il buffer dei comandi, se attivo, accumula i dati inviati attraverso la chiamata di sistema `write`, poi, come vedremo, un kernel thread si occupa di inviare i dati al dispositivo.

Le schede Galil DMC 1800 dispongono già di un buffer hardware, ovvero la FIFO di scrittura: quest'ultima infatti accumula i dati scritti nella prima porta di I/O fino ad un massimo di 512 byte. Il processore della scheda preleva poi da questa FIFO un comando alla volta, lo esegue e solo quando è terminato legge il comando successivo.

Il buffer software introduce quindi un secondo livello di cache; le ragioni che giustificano la sua introduzione sono fondamentalmente due:

- garantire un flusso continuo di comandi al processore delle schede Galil DMC 1800, anche con un carico di sistema elevato;
- diminuire il numero di chiamate di sistema `write` necessarie ad inviare un blocco di comandi alla scheda; in questo modo si diminuisce il numero di passaggi tra *user mode* e *kernel mode* producendo in genere un aumento delle prestazioni.

La continuità nel flusso dei comandi può essere particolarmente importante durante l'esecuzione di movimenti sui motori di un robot; vediamo un esempio. Quando si deve afferrare un oggetto, ad esempio mediante una pinza, tipicamente il robot si avvicina all'oggetto con discreta velocità, poi

completa l'ultima fase del movimento diminuendo progressivamente la velocità dei motori per ottenere maggiore precisione e soprattutto per evitare un arresto troppo brusco; infine, una volta raggiunto il punto esatto aziona la pinza ed afferra l'oggetto. Ovviamente queste operazioni non sono effettuabili con un solo comando, ma piuttosto con una serie di comandi che specificano le posizioni da raggiungere e le tensioni da applicare ai vari motori. Se il buffer di scrittura si svuota ci potrebbe essere un istante di pausa fra due comandi di movimento; in questo modo il robot potrebbe subire un arresto molto brusco, soprattutto se il movimento che si stava effettuando era ad una velocità elevata.

Utilizzando la funzione di scrittura non bufferizzata presentata nel paragrafo 5.5 è molto probabile che il programmatore scriva i comandi attraverso una API[3] simile alla seguente:

```
int send_commands(int fd, char *commands,
                  unsigned int len, unsigned int delay)
{
   unsigned int count = 0, rc;
   if(!commands || len <= 0)
      return -1;
   do {
      rc = write(fd, commands+count, len-count);
      if(rc < 0)
         exit(1);
      if(!rc)
         nanosleep(delay);
      count += rc;
   }while(count < len);
   return 0;
}
```

In questo modo quando la FIFO si riempe l'applicazione inizia a chiamare continuamente la funzione `write`, producendo ogni volta un passaggio da

[3]Application Program Interface

user mode a kernel mode e viceversa. Se la FIFO viene svuotata lentamente il numero di volte in cui viene chiamata inutilmente la funzione `write` sarà molto elevato, compromettendo in parte le prestazioni del resto del sistema; se la FIFO viene svuotata molto velocemente non è detto che il processo che scrive riesca ad impedire che la FIFO si svuoti, in quanto questa funzione deve comunque concorrere con gli altri processi attivi. È evidente, quindi, come non possa essere assicurata neanche la continuità del flusso dei comandi di cui si è precedentemente discusso. Infine l'utente dovrebbe anche preoccuparsi di impostare opportunamente il parametro `delay`: quest'ultimo può essere utilizzato per introdurre un breve ritardo fra le varie chiamate alla funzione `write`, evitando così di sovraccaricare troppo il sistema. Trovare un valore adeguato per `delay` non è comunque semplice, poiché un tempo troppo lungo potrebbe portare ad uno svuotamento delle FIFO.

L'introduzione del buffer software mitiga questi problemi; vediamo alcune sue caratteristiche che ne mettono in evidenza gli aspetti positivi:

- la funzione `write` bufferizzata è stata implementata in modo bloccante, quindi anche se il buffer software si riempe il numero di passaggi tra user mode e kernel mode si riduce notevolmente;

- la dimensione del buffer è configurabile quindi, quando sufficientemente grande, permette alla funzione `write` di scrivere tutti i comandi con una sola chiamata nella maggior parte dei casi; ciò resta valido anche se si sceglie di usare la funzione `write` bufferizzata in modo non bloccante (flag `O_NONBLOCK`);

- una volta che i comandi sono stati accumulati nel buffer software la scrittura nella FIFO viene realizzata da un kernel thread; quest'ultimo fornisce maggiori garanzie in termini di continuità di trasferimento di dati verso il dispositivo, vediamo perché:

  – il kernel thread viene eseguito in kernel mode, quindi non sono necessari ulteriori passaggi da user mode a kernel mode;

  - il buffer è già nello spazio di memoria del kernel quindi non è necessario effettuare copie tra lo spazio di memoria utente e quello del kernel;
  - il cambio di processo verso un kernel thread richiede molte meno operazioni di quello necessario per un processo utente.

Questa è soltanto una discussione preliminare sull'impatto di un buffer software in un dispositivo come le schede Galil DMC 1800; mostrando l'implementazione del buffer potranno essere trattati alcuni altri aspetti. Da questa prima analisi però emerge una caratteristica fondamentale della scrittura bufferizzata: i problemi derivanti dalle caratteristiche tecniche del dispositivo vengono completamente spostati dallo spazio utente a quello del kernel. Infatti con il buffer di scrittura l'utente vedrà la maggior parte delle chiamate alla funzione `write` andare a buon fine, senza dover iterare per un numero eccessivo di volte. La API `send_commands()` mostrata precedentemente potrebbe non essere più necessaria: in particolare la funzione `nanosleep()` e quindi l'uso del parametro `delay` possono sicuramente essere eliminati. Infatti, se l'accesso alla FIFO è gestito dal kernel thread l'utente può ignorare lo stato dei buffer hardware: è il driver che se ne fa carico. Ovviamente anche il driver deve preoccuparsi di effettuare un polling sul dispositivo per sapere quando è possibile scrivervi: come vedremo però nel kernel vi sono strumenti che rendono queste operazioni molto semplici, precise e piuttosto leggere dal punto di vista computazionale.

5.6.1 Progettazione

In fase di progetto del buffer dei comandi, nella scelta delle caratteristiche di sviluppo, si è dovuto tener conto di alcuni aspetti determinanti:

- tutti i comandi scritti con la funzione `write` devono sempre essere inviati alla FIFO del dispositivo, quindi il buffer deve essere *non sovrascrivibile*; conseguentemente, quando il buffer è pieno la chiamata di sistema `write` dovrà poter bloccare il processo o restituire il controllo;

- l'ordine con cui i comandi sono scritti nel buffer deve essere lo stesso con cui vengono inviati alla FIFO, quindi anche il buffer software deve funzionare con *politica First In First Out*;

- il buffer deve poter essere acceduto in tempi molto brevi, quindi *sistemi di sincronizzazione come semafori e spinlock non sono adeguati*;

- il buffer deve permettere l'*inserimento di pseudo-comandi*, ad esempio, attraverso la chiamata di sistema `ioctl`.

Considerando solo i primi tre punti è evidente come un normale buffer circolare sembri la soluzione più adeguata. Infatti un buffer circolare quando utilizzato con politica FIFO richiede poche operazioni per il trattamento dei dati; inoltre può essere implementato in modo da non richiedere l'uso di sistemi di lock. Quest'ultima caratteristica necessita però di una condizione: il buffer deve essere acceduto da un solo utente in lettura e da un solo utente in scrittura. La funzione `write` sarebbe l'unica a poter aggiungere dati nel buffer[4], mentre l'unico utente in lettura può essere realizzato attraverso un singolo kernel thread che trasferisce i dati dal buffer alla FIFO.

Il quarto punto, lo pseudo-comando, complica notevolmente le cose; vediamo alcune ragioni:

- usando la chiamata di sistema `ioctl` per aggiungere lo pseudo-comando si introdurrebbe un terzo utente del buffer; infatti, se non controllata, la funzione `ioctl` può concorrere con la funzione `write`;

- senza pseudo-comandi si è certi che ogni byte contenuto nel buffer deve semplicemente essere scritto nella FIFO; con l'introduzione di uno pseudo-comando, il driver ogni qualvolta legge dati dal buffer deve essere in grado di distinguere i byte che vanno scritti nella FIFO e quelli appartenenti allo pseudo-comando.

[4] Ciò può essere affermato poiché la chiamata di sistema `open` consente ad un solo utente di aprire il device file con permesso di scrittura. Si veda il paragrafo 5.2.

Queste prime due considerazioni mettono già in evidenza come la gestione del buffer si complichi, soprattutto se si considera che le prestazioni non possono subire significativi degradi.

Se, per ragioni relative alle prestazioni, si esclude la possibilità di utilizzare sistemi di lock, tre utenti sono necessariamente da evitare. A questo punto un'alternativa poteva essere quella di non utilizzare la chiamata `ioctl` ed inserire gli pseudo-comando scrivendo speciali caratteri attraverso la funzione `write`: questo avrebbe però significato che tutti i caratteri presenti nel buffer dovevano subire un parsing prima di essere scritti nella FIFO hardware. In questo modo si sarebbe introdotto un notevole degrado delle prestazioni. Come vedremo si è infine scelto di limitare l'uso dei comandi `ioctl` che inseriscono gli pseudo-comandi: quest'ultimi saranno infatti utilizzabili esclusivamente dai processi che aprono il device file con permesso di scrittura. Ciò è sufficiente per garantire che un solo utente alla volta possa aggiungere dati nel buffer.

In fase di lettura del buffer resta comunque il problema di saper distinguere i byte degli pseudo-comandi, da quelli che invece vanno scritti nella FIFO. Volendo evitare anche in questo caso l'uso del parsing la soluzione adottata è stata quella di assegnare un flag ad ogni byte presente nel buffer. Come vedremo quindi, in fase di lettura sarà sufficiente testare il valore di un solo bit per individuare gli pseudo-comandi.

Nei paragrafi successivi saranno descritte in dettaglio tutte le caratteristiche del buffer e la sua implementazione. Maggiori dettagli sulle motivazioni di alcune scelte saranno forniti durante la discussione.

5.6.2 Descrizione

Sulla base delle considerazioni discusse nel paragrafo precedente, il buffer dei comandi è stato realizzato con le seguenti caratteristiche:

- è un buffer circolare che non permette il trabocco, quindi quando è pieno non si possono aggiungere nuovi dati;

- la sua dimensione è statica e viene stabilita in fase di inizializzazione, ovvero durante l'apertura in scrittura del device file card; il valore di default è pari a 8192 byte;
- essendo un buffer circolare l'area di memoria occupata e quella libera sono sempre ben delimitate;
- è stato implementato un unico pseudo-comando che inserisce una pausa nel trasferimento di dati dal buffer software alla FIFO;
- ad ogni byte memorizzato sono associati 2 bit di controllo con il seguente significato:
  - un *present flag*: se 1 il relativo byte contiene un'informazione valida, altrimenti può essere sovrascritto;
  - un *pause flag*: se 1 il byte relativo contiene informazioni di uno pseudo-comando di pausa, altrimenti è un dato da scrivere nella FIFO hardware;
- l'intera area di memoria relativa ai flag e quella relativa ai dati sono allocate in due zone distinte, ottenendo l'efficienza massima dai byte allocati; infatti, creando un'unica zona di memoria sarebbe stato complicato suddividerla in blocchi di 10 bit, senza produrre un'ulteriore spreco di spazio oppure un'eccessiva complessità nel calcolo degli offset;
- ogni inserimento di uno pseudo-comando di pausa necessita sempre di 4 byte consecutivi, nei quali viene memorizzata la durata della pausa in millisecondi;
- lettura e scrittura del buffer sono implementate in modo bloccante;
- in ogni istante è molto semplice determinare lo spazio occupato nel buffer e quindi calcolarne la percentuale di occupazione; come vedremo nel capitolo 6 questa caratteristica permette all'utente di essere informato sullo stato del buffer, attraverso il device file event;

- solo l'implementazione bufferizzata della chiamata di sistema `write` inserisce comandi nel buffer;
- solo l'implementazione della chiamata di sistema `ioctl` può inserire uno pseudo-comando di pausa; il processo deve però aver aperto il file con permesso di scrittura;
- il trasferimento dei dati dal buffer alla FIFO è affidato ad un unico "lavoro", a ciclo infinito, eseguito da una workqueue; la funzione che implementa la scrittura sulla FIFO è stata realizzata in due versioni:
  - una viene utilizzata quando il comando di pausa è abilitato; infatti in questo caso sono necessari alcuni controlli addizionali;
  - la seconda versione è usata se il comando di pausa è disabilitato ed ha quindi un numero di istruzioni inferiore;

  la selezione fra le due funzioni avviene in fase di inizializzazione del buffer e questo quindi produce un miglioramento delle prestazioni quando il comando di pausa non è necessario.

In figura 5.1 vediamo una schematizzazione del buffer.

Per ogni byte ci sono due bit dedicati ai flag: ciò introduce uno spreco di memoria pari al 25% della dimensione dedicata ai dati. Questa percentuale è sicuramente abbastanza alta, ma in realtà, date le sue caratteristiche, la dimensione del buffer è tipicamente nell'ordine di pochi kilobyte: quindi, considerato in valore numerico, lo spreco non è poi eccessivo. Ad esempio, per un buffer di 8 Kbyte, sono necessari ulteriori 2 Kbyte per memorizzare i flag: l'intero buffer occupa quindi 10 Kbyte, che non rappresenta sicuramente un valore critico, neanche in sistemi non particolarmente recenti.

D'altra parte i flag permettono un miglioramento delle prestazioni dovuto ad una maggior velocità nella valutazione dello stato del buffer: come vedremo nei dettagli di implementazione testando un solo bit si è in grado di capire se è possibile leggere o scrivere dati nel buffer. Inoltre attraverso il singolo bit del *pause flag* si può distinguere un byte relativo ad un vero

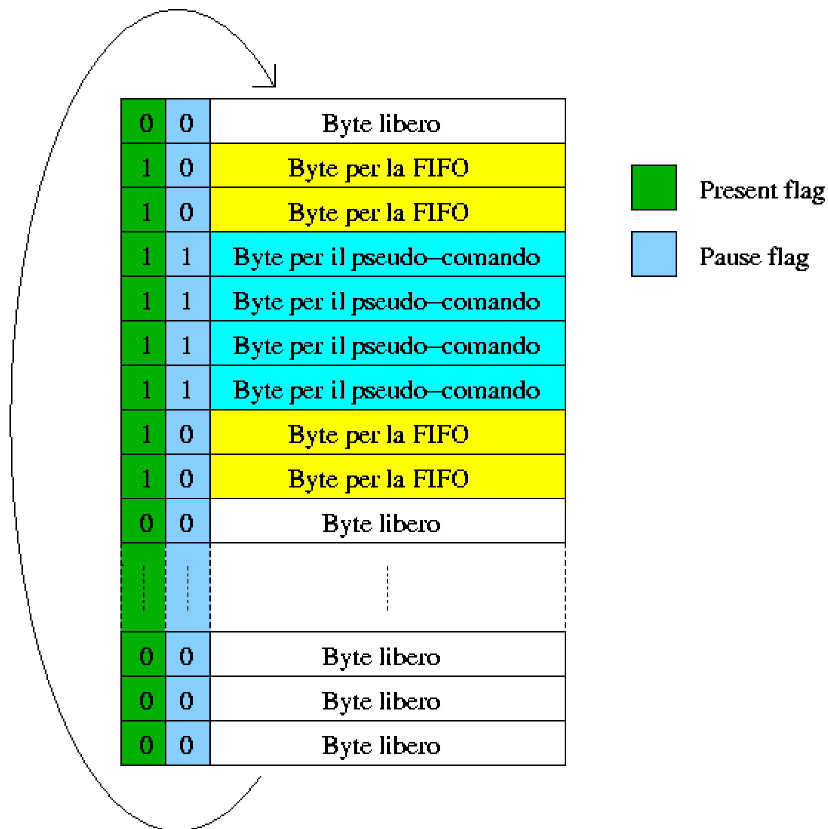

Figura 5.1: Schema rappresentativo del buffer dei comandi

comando, da uno dedicato allo pseudo-comando di pausa: in questo modo non è necessario effettuare alcun parsing sui dati inviati dall'utente.

5.6.3 Implementazione

Nei file `galil1800_card.h` e `galil1800_card.c` si possono trovare la definizione e le implementazioni della struttura dati e delle funzioni che realizzano il buffer dei comandi. In questo paragrafo sarà presentata in dettaglio la struttura dati e la sua inizializzazione e verranno descritte le funzioni principali che gestiscono il buffer.

Struttura dati

La struttura `galil1800_card_wbuffer` descrive il buffer dei comandi per il device file card. Una sua istanza, chiamata `card_wbuff`, è inclusa nella struttura `galil1800_state`: in questo modo, ogni dispositivo controllato dal driver dispone del proprio buffer.

Vediamo la definizione della struttura:

```
struct galil1800_card_wbuffer {
   unsigned short run;
   unsigned int size;
   unsigned long *flag;
   unsigned char *data;
   unsigned char *head;
   unsigned char *tail;
   unsigned char last_size_p10;
   unsigned char last_event_opt_flag;
   unsigned int pause_enable;
   unsigned long pause;
   struct timer_list pause_timer;
   wait_queue_head_t read_wait;
   wait_queue_head_t write_wait;
   struct workqueue_struct *wq;
   struct work_struct work;
   struct class_device_attribute cd_attr_size;
   struct class_device_attribute cd_attr_pause_enable;
};
```

Segue la descrizione di ogni suo campo:

- `run`: la funzione che trasferisce i dati dal buffer alla FIFO è a ciclo infinito e questo valore viene utilizzato come punto di uscita; se impostato a zero indica che è necessario completare il trasferimento di tutti i byte già presenti nel buffer e poi terminare la funzione;

- `size`: dimensione totale del buffer (in byte); se è zero indica che il buffer dei comandi è disabilitato;
- `flag`: puntatore al primo byte dell'area di memoria che contiene i flag;
- `data`: puntatore al primo byte dell'area di memoria che contiene i dati;
- `head`: puntatore al primo byte libero della zona dati;
- `tail`: puntatore al primo byte da leggere nella zona dati;
- `last_size_p10` e `last_event_opt_flag`: sono campi relativi al report nel device file event dello stato di occupazione del buffer e saranno discussi in seguito;
- `pause_enable`: se 0 lo pseudo-comando di pausa è disabilitato, se 1 è abilitato;
- `pause`: quando vi è una pausa in corso contiene il valore che deve avere la variabile globale `jiffies` per determinare la fine della pausa; se invece la scrittura nella FIFO non è in pausa vale zero;
- `pause_timer`: kernel timer (vedi paragrafo 2.3.4) che determina il risveglio dalla pausa;
- `read_wait`: coda di attesa in lettura; blocca il processo della workqueue quando non ci sono più dati da leggere, è attiva una pausa oppure la FIFO hardware è piena;
- `write_wait`: coda di attesa per la scrittura nel buffer; blocca il processo che ha chiamato la funzione `write` bufferizzata quando il buffer è pieno;
- `wq`: workqueue per il trasferimento dei dati dal buffer alla FIFO;
- `work`: lavoro assegnato alla workqueue, che effettua il trasferimento dei dati;
- `cd_attr_size`: esporta il campo `size` nel *sysfs* filesystem;

- `cd_attr_pause_enable`: esporta il campo `pause_enable` nel *sysfs* filesystem.

Inizializzazione

La funzione che si occupa di inizializzare la struttura dati appena descritta è la seguente:

```
static int init_galil1800_card_wbuffer_open_dev(
                    struct galil1800_card_wbuffer *wbuff_p)
```

Con l'obiettivo di risparmiare le risorse del sistema, su tale funzione sono applicati due particolari accorgimenti:

- come è stato descritto nel paragrafo 5.2, questa funzione viene chiamata solo quando il device file card viene aperto con permesso di scrittura;
- la prima istruzione verifica se il buffer dei comandi è stato attivato; in caso contrario, la struttura non viene inizializzata e la funzione ritorna immediatamente.

In questo modo, le risorse relative al buffer dei comandi vengono allocate solo quando sono effettivamente necessarie.

Due campi della struttura dati `galil1800_card_wbuffer` sono inizializzati durante il caricamento del modulo: `size` e `pause_enable`. Questi in effetti hanno valori di default che sono rispettivamente 8192 e 0; tali valori possono essere modificati utilizzando i parametri di caricamento del modulo `card_wbuff_sz` e `pause_enable`, oppure scrivendo sugli omonimi file presenti nelle directory `/sys/class/galil/card*` del *sysfs* filesystem. Un terzo campo risente dello stato di attivazione del comando di pausa e quindi viene inizializzato contestualmente a quest'ultimo: si tratta del campo `work`. Questa struttura viene inizializzata con la funzione che trasferisce i dati dal buffer alla FIFO; ne esistono due versioni:

- `galil1800_card_wbuffer_flush()`: quando lo pseudo-comando pause è disabilitato;

- `galil1800_card_wbuffer_flush_with_pause_cmd()`: se il comando di pausa è abilitato.

Alcuni altri campi sono inizializzati durante il caricamento del modulo: sono quei dati che, ad esempio, non necessitano di essere allocati o che possono essere staticamente definiti. I campi in questione sono: `pause_timer`, `read_wait`, `write_wait`, `cd_attr_size` e `cd_attr_pause_enable`. Questi ultimi due campi sono utilizzati in fase di creazione del device file a caratteri per creare i relativi attributi nel *sysfs* filesystem. Infine i campi `last_size_p10` e `last_event_opt_flag` sono azzerati durante il caricamento del modulo e poi ogni qualvolta il buffer dei comandi viene deallocato.

Vediamo ora le istruzioni che vengono eseguite per inizializzare il buffer dei comandi, qualora quest'ultimo sia attivo; i nomi delle variabili citati, salvo diversa indicazione, fanno riferimento ai campi della struttura `galil1800_card_wbuffer`:

1. viene inizializzato il campo `wq`, creando una workqueue con una singola coda[5] per il trasferimento dei dati dal buffer alla FIFO;

2. vengono allocati `size` byte per l'area dati del buffer, salvandone il puntatore nel campo `data`;

3. ai puntatori `head` e `tail` viene assegnato il valore di `data`;

4. vengono allocati `size/4` byte per l'area del buffer dedicata ai flag, salvandone il puntatore nel campo `flag`; per ogni byte dell'area dati sono necessari 2 bit di flag, quindi un byte di flag serve 4 byte di dati;

5. `pause` viene impostato a zero;

6. al campo `run` viene assegnato 1 ed il lavoro `work` viene assegnato alla workqueue `wq`; a questo punto una delle due funzioni di trasferimento dati verso la FIFO è attiva ed il buffer è pronto per essere utilizzato.

[5]Si veda il paragrafo 2.3.4.

Quando il device file card viene chiuso (dal processo che lo aveva aperto in scrittura) viene utilizzata la seguente funzione, che rilascia le risorse occupate dal buffer:

```
static inline void clean_galil1800_card_wbuffer_close_dev(
                        struct galil1800_card_wbuffer *wbuff_p)
```

Questa funzione, se il buffer era disattivato (`size` uguale a zero) non compie alcuna istruzione e ritorna il controllo, altrimenti:

- imposta a zero il campo `run`; in questo modo la funzione in esecuzione sulla workqueue finisce di svuotare il buffer di scrittura e poi termina;
- attende che il lavoro nella workqueue sia terminato, poi la rilascia;
- rilascia la memoria occupata indirizzata dai puntatori `data` e `flag`;
- azzera i seguenti campi: `data`, `flag`, `head`, `tail`, `last_size_p10` e `last_event_opt_flag` e `pause`.

Utilizzo dei flag

Per accedere in lettura o scrittura al buffer è necessario controllare il valore dei flag, al fine di sapere se il byte di riferimento è libero o meno e distinguere il tipo di dato contenuto. Prima di procedere con altre spiegazioni vediamo quindi come viene utilizzata l'area di memoria indirizzata dal campo `flag` della struttura `galil1800_card_wbuffer`.

L'intera area può essere logicamente suddivisa in blocchi di 2 bit rappresentanti rispettivamente il present flag ed il pause flag. Quindi i primi 2 bit si riferiscono al primo byte dall'area dati, 3 e 4 bit al secondo byte dell'area dati, e così via.

La seguente macro, con un massimo di 3 operazioni, calcola l'offset del bit relativo al byte `_byte_pt` dell'area dati:

```
GALIL1800_CARD_WB_FLAG_BIT_OFFSET(_wbuff_p,_pt_byte,_flag_type) \
 ( (((_pt_byte) - (_wbuff_p)->data) *                          \
    GALIL1800_CARD_WB_FLAG_BIT_FOR_BYTE) + (_flag_type) )
```

`_wbuff_p` punta alla struttura che descrive il buffer, mentre `_flag_type` indica se si tratta del present flag (valore 0), oppure del pause flag (valore 1). Nel primo caso il compilatore elimina l'operazione di somma con zero, quindi sono sufficienti due istruzioni per il calcolo dell'offset per il present flag.

Nel file `galil1800_card.h` è definita un'enumerazione per il tipo di flag ed alcune macro per effettuare le operazioni sui bit.

Aggiunta di dati nel buffer

Per aggiungere dati al buffer sono necessarie le seguenti operazioni:

1. controllare che il buffer non sia già pieno; poiché il primo byte in cui deve essere aggiunta una nuova informazione nel buffer è quello indirizzato dal puntatore `head`, per sapere se il buffer è pieno è sufficiente verificare che questo byte sia libero; quindi testare se il buffer è pieno significa testare il present flag del byte indirizzato da `head`;
2. se il buffer non è pieno, scrivere il nuovo dato nel byte indirizzato da `head`;
3. impostare ad 1 il present flag relativo al byte indirizzato da `head`;
4. incrementare `head`, verificando che quando raggiunge la fine dell'area allocata gli venga riassegnato il valore del puntatore `data`.

Inserimento di una pausa

Attraverso il comando `GALIL1800_IOCTL_SET_PAUSE` della chiamata di sistema `ioctl` è possibile inserire una pausa che interrompe la scrittura di dati nella FIFO; l'argomento passato alla funzione `ioctl` determina la durata della pausa espressa in millisecondi. Si è già detto che per utilizzare tale comando il processo deve aver aperto il device file con permesso di scrittura; questo evita che contemporaneamente un altro processo acceda al buffer per scrivere sul byte indirizzato dal puntatore `head`. La funzione per inserire una pausa nel buffer è la seguente:

```
static int galil1800_card_wbuffer_add_pause(
        struct galil1800_card_wbuffer *wbuff_p, unsigned long ms)
```

I millisecondi di durata della pausa, prima di essere introdotti nel buffer vengono trasformati in numero di tick e sommati alla variabile globale `jiffies`: il valore ottenuto indicherà l'istante in cui la scrittura potrà riprendere. In questo modo il numero di byte necessari a memorizzare la pausa nel buffer sarà pari a 4 byte, ovvero la dimensione di un `unsigned long`.

La funzione `galil1800_card_wbuffer_add_pause()` verifica che vi siano 4 byte liberi nel buffer; in caso positivo prepara i byte da scrivere e li aggiunge, impostando, per ogni byte, il relativo pause flag ad 1.

Trasferimento dati dal buffer alla FIFO

Una volta che i dati sono stati aggiunti nel buffer software devono, prima possibile, essere scritti nella FIFO hardware. A questo scopo, in fase di inizializzazione del buffer, è stata assegnata ad una workqueue l'esecuzione di una funzione a ciclo infinito. Come già detto vi sono due versioni di questa funzione e la selezione viene effettuata sulla base dello stato di attivazione dello pseudo-comando di pausa. Di seguito, per semplicità, verrà discusso il caso in cui il comando di pausa è disabilitato; alla fine saranno comunque mostrate alcune differenze fra le due funzioni.

La funzione `galil1800_card_wbuffer_flush()` è eseguita nella workqueue, qualora lo pseudo-comando di pausa è disabilitato; vediamo la sua implementazione, per poi discuterne le parti principali:

```
void galil1800_card_wbuffer_flush(void *data)
{
   struct galil1800_card_wbuffer *wbuff_p =
         (struct galil1800_card_wbuffer *)data;
   struct galil1800_state *s =
      container_of(wbuff_p, struct galil1800_state, card_wbuff);
   while(1) { /* Wait for data */
      while(!GALIL1800_CARD_WB_NOT_EMPTY(wbuff_p)) {
```

```
            if(!wbuff_p->run)
               return;
            wake_up_interruptible(&wbuff_p->write_wait);
            wait_event(wbuff_p->read_wait,
                GALIL1800_CARD_WB_NOT_EMPTY(wbuff_p) || !wbuff_p->run);
         }
         galil1800_add_wbuff_event(s);
         do {
            galil1800_card_wbuffer_write_to_fifo(s, wbuff_p);
            galil1800_add_wbuff_event(s);
         }while(GALIL1800_CARD_WB_NOT_EMPTY(wbuff_p));
         wake_up_interruptible(&wbuff_p->write_wait);
   }
}
```

A parte le prime due istruzioni, necessarie a reperire i puntatori alle strutture `galil1800_card_wbuffer` e `galil1800_state`, il resto della funzione è interamente contenuto in un ciclo infinito. Vediamo quali operazioni svolge:

- finché il buffer è vuoto:
  - se `run` è stato impostato a zero termina la funzione; questo, come precedentemente spiegato è ciò che avviene quando il buffer deve essere deallocato;
  - riattiva eventuali processi bloccati nella coda `write_wait`; questa coda è quella in cui la funzione `write` si blocca se il buffer è pieno;
  - blocca il processo finché non arrivano nuovi dati nel buffer oppure `run` viene posto a zero;
- quando ci sono dati nel buffer la funzione esce dal ciclo appena descritto e usa la funzione `galil1800_add_wbuff_event()` per generare un nuovo evento nel buffer del device file event che sarà descritto nel capitolo 6; in ogni caso tale funzione calcola la percentuale di occupazione del

buffer e genera un nuovo evento solo se tale valore è cambiato rispetto all'ultima volta in cui era stato riportato;

- a questo punto si è certi che almeno un byte è presente nel buffer quindi si entra in un ciclo che trasferisce i dati nella FIFO finché il buffer non diviene nuovamente vuoto; in tale ciclo:
  - si usa la funzione `galil1800_card_wbuffer_write_to_fifo()` per trasferire i dati nella FIFO in blocchi di uno o più byte;
  - si ricontrolla la percentuale di occupazione del buffer per il device file event;
- quando il buffer è nuovamente vuoto si tenta di riattivare un'eventuale funzione `write` bloccata perché il buffer era pieno e poi si torna alla prima istruzione.

La funzione `galil1800_card_wbuffer_write_to_fifo()` è in realtà una macro implementata come mostrato di seguito:

```
#define galil1800_card_wbuffer_write_to_fifo(_s, _wbuff_p) do { \
      int _burst;                                            \
      switch(__galil1800_may_write(_s)) {                    \
         case 0: /* FIFO full */                             \
            while(__galil1800_may_write(_s)==0) {            \
               wait_event_timeout((_wbuff_p)->read_wait,     \
                           __galil1800_may_write(_s)>0,      \
                                 (_s)->write_delay );        \
            }                                                \
            break;                                           \
         case 1: /* One slot free */                         \
            outb(*((_wbuff_p)->tail), GALIL1800_WRITE_REG(_s)); \
            GALIL1800_CARD_WB_CLEAR_PRESENT_FLAG(_wbuff_p);     \
            galil1800_card_wbuffer_inc_pt(_wbuff_p,          \
                                (_wbuff_p)->tail);           \
            break;                                           \
```

```
        case 2: /* Almost half FIFO free */                 \
           _burst=0;                                         \
           do {                                              \
              outb(*((_wbuff_p)->tail),                      \
                       GALIL1800_WRITE_REG(_s));             \
              GALIL1800_CARD_WB_CLEAR_PRESENT_FLAG(_wbuff_p);  \
              galil1800_card_wbuffer_inc_pt(_wbuff_p,        \
                                   (_wbuff_p)->tail);        \
              _burst++;                                      \
           }while(_burst < GALIL1800_HALF_FULL_WRITE_BURST &&  \
                    GALIL1800_CARD_WB_NOT_EMPTY(_wbuff_p));    \
           break;                                            \
     };                                                      \
  }while(0)
```

Come si può vedere l'intera macro esegue un controllo sullo stato della FIFO attraverso la funzione `__galil1800_may_write()` descritta nel paragrafo 4.3.2: tale funzione restituisce 0 se la FIFO di scrittura è piena, 1 se è possibile scrivere un byte, 2 se è possibile scrivere fino a 255 byte. Vediamo quindi come vengono trattati i tre casi:

- se la FIFO è piena si effettua un polling sul registro di controllo ogni `write_delay` ticks;
- se è possibile scrivere un byte, quest'ultimo viene prelevato dall'indirizzo contenuto nel puntatore `tail` del buffer, quindi il present flag viene azzerato ed il puntatore a `tail` incrementato;
- infine, se la FIFO è piena per meno di metà della sua dimensione, si entra in un ciclo che effettua il trasferimento dei byte finché il buffer software non si svuota oppure si scrivono 255 byte.

Nel caso in cui lo pseudo-comando di pausa è abilitato è necessario eseguire qualche istruzione in più per riconoscere i byte relativi ai comandi di pausa e per decidere quando è necessario fermare la scrittura nella FIFO. In particolare le due funzioni appena descritte vengono sostituite dalle seguenti:

```
void galil1800_card_wbuffer_flush_with_pause_cmd(void *data)
galil1800_card_wbuffer_write_to_fifo_with_pause_cmd(_s, _wbuff_p)
```

Nella prima funzione l'unica differenza risiede nel fatto che nel ciclo di scrittura dei dati viene effettuato anche il test del pause flag; qualora venga rilevata una richiesta di pausa sono svolte le seguenti operazioni:

- si leggono i 4 byte che contengono l'istante (in tick) in cui si può riattivare la scrittura;
- si attiva un kernel timer che sbloccherà il processo della workqueue nel suddetto istante;
- si blocca il processo.

Infine la macro per la scrittura nella FIFO differisce solo nel caso in cui è possibile scrivere fino a 255 byte; all'inizio del ciclo di scrittura viene aggiunta la seguente istruzione per testare il pause flag:

```
if(GALIL1800_CARD_WB_TEST_PAUSE(_wbuff_p))
   break;
```

5.7 Scrittura bufferizzata

Ora che è stato introdotto il funzionamento del buffer di scrittura è possibile descrivere l'implementazione della file operation `write` quando il buffer è attivo.

La funzione `galil1800_card_buffered_write()` è identica alla versione non bufferizzata tranne per due istruzioni:

- il calcolo del massimo numero di byte che possono essere scritti non si preoccupa più della dimensione della FIFO, bensì tiene in considerazione quella del buffer dei comandi; questa è l'istruzione che inizializza la variabile `request`:

```
    request = size < s->card_wbuff.size ?
              size : s->card_wbuff.size;
```

- la funzione `galil1800_send()` che scrive i dati nella FIFO viene sostituita da `galil1800_card_wbuffer_add()` che aggiunge i byte passati alla funzione `write` nel buffer dei comandi.

La funzione `galil1800_card_wbuffer_add()` svolge le seguenti operazioni:

- qualora il buffer di scrittura sia pieno, blocca il processo che ha chiamato la funzione `write`; solo la funzione in esecuzione nella workqueue potrà riattivarlo, utilizzando le istruzioni già descritte;
- quando si libera spazio nel buffer, vi aggiunge dati finché non scrive `request` byte, oppure il buffer è nuovamente pieno;
- infine restituisce il numero di byte scritti.

Vediamone l'implementazione:

```
static int galil1800_card_wbuffer_add(
                     struct galil1800_card_wbuffer *wbuff_p,
                     unsigned char *data, size_t len, int nonblock)
{
   int rc = 0;
   while(GALIL1800_CARD_WB_FULL(wbuff_p)) {
      if(nonblock)
         return -EAGAIN;
      wake_up(&wbuff_p->read_wait);
      if(wait_event_interruptible(wbuff_p->write_wait,
                           !GALIL1800_CARD_WB_FULL(wbuff_p)))
         return -ERESTARTSYS;
   }
   while(rc < len) {
```

```
        *(wbuff_p->head) = data[rc++];
        GALIL1800_CARD_WB_SET_PRESENT_FLAG(wbuff_p);
        galil1800_card_wbuffer_inc_pt(wbuff_p, wbuff_p->head);
        if(GALIL1800_CARD_WB_FULL(wbuff_p))
            break;
    }
    wake_up(&wbuff_p->read_wait);
    return rc;
}
```

È importante notare come mediante la funzione `wake_up()` si tenti di riattivare la funzione di trasferimento dati nella FIFO; ciò avviene in due casi:

- quando il buffer è pieno, anche se in questo caso la funzione è sicuramente già attiva;
- dopo aver aggiunto dati nel buffer; in questo caso è fondamentale riattivare la funzione in esecuzione sulla workqueue, poiché se il buffer era vuoto tale funzione era sicuramente bloccata.

Capitolo 6

Eventi speciali ed interruzioni

In questo capitolo sarà introdotta la *gestione degli eventi*. Nel driver per le schede Galil DMC 1800 è stato infatti previsto un apposito device file, chiamato *event*, il cui scopo è informare le applicazioni utente del verificarsi di particolari avvenimenti. Al momento i tipi di eventi che vengono riportati sono suddivisibili in due categorie:

- informazioni sullo stato di occupazione del buffer dei comandi;
- notifica delle interruzioni hardware.

Una parte di questo capitolo sarà anche dedicata ad illustrare la gestione delle interruzioni hardware emesse dalle schede Galil DMC 1800.

6.1 Introduzione al device file event

Nello sviluppo di un driver deve essere posta particolare attenzione sugli aspetti riguardanti le interruzioni emesse dal dispositivo. In particolare, in merito alle interruzioni, le operazioni che deve svolgere un driver possono essere suddivise in due gruppi:

- la gestione dell'interruzione, ovvero le istruzioni che devono essere eseguite ogni qualvolta la scheda emette un'interruzione; tramite queste operazioni, è possibile ottenere dalle schede Galil DMC 1800 un

byte che descrive la ragione dell'interruzione, chiamato *byte di stato* dell'interruzione;

- una volta compresa la ragione dell'interruzione un driver ha in genere due scelte:
  - se il motivo dell'interruzione riguarda aspetti di gestione del dispositivo, quest'ultima è di solo interesse del driver. Ad esempio, un'interruzione può informare il driver che esistono dati pronti per essere letti da un particolare registro: in questo caso il driver sblocca eventuali chiamate `read` in attesa dei dati di quel registro. L'utente non viene quindi direttamente informato dell'arrivo dell'interruzione, ma è il driver che agisce di conseguenza.
  - in alcuni casi il motivo dell'interruzione può invece essere privo di significato per il driver, ma piuttosto di interesse per l'utente che sta usando il dispositivo; in questo caso il driver deve solo preoccuparsi di informare l'utente dell'arrivo di un'interruzione, specificandone la ragione.

Nel capitolo 1 è stata discussa l'importanza delle interruzioni ed anche come queste ultime sono utilizzate dalle schede Galil DMC 1800. In particolare si è visto che tali schede non emettono mai interruzioni di esclusivo interesse per il driver: la tabella 1.2 di pagina 20 riassume i possibili significati delle interruzioni emesse dalle schede Galil DMC 1800 ed evidenzia come esse siano tutte di esclusivo interesse per l'utente. Questo driver quindi, ad ogni arrivo di un'interruzione, deve solo preoccuparsi di gestirla ed esportare nello spazio utente il relativo byte di stato.

Il device file event è un device file accessibile in sola lettura, progettato in primo luogo per rendere accessibili nello spazio utente i byte che specificano la ragione di un'interruzione. In realtà è stato poi implementato un sistema più complesso che permette, attraverso questo device file, di esportare anche altri tipi di informazioni. Ogni parte del driver può infatti segnalare un evento speciale che potrebbe essere di qualche interesse per l'utente: al momento

è implementato un report dello stato di occupazione del buffer dei comandi descritto nel capitolo 5.

La caratteristica principale del device file event è uno specifico *buffer degli eventi*: ogni parte del driver vi può aggiungere informazioni che segnalano il verificarsi di un particolare evento. Attraverso la chiamata di sistema `read` del device file event è possibile accedere al contenuto di tale buffer e quindi ricevere informazioni sugli eventi che si sono verificati.

L'idea principale è quindi quella di un device file, dotato di una chiamata di sistema `read` bloccante sulla quale un processo attende di ricevere informazioni: in questo caso, l'arrivo di un dato significherà il verificarsi di un evento, mentre le informazioni lette ne specificheranno le caratteristiche.

Per l'utilizzo del device file sono state implementate le file operation relative alle seguenti chiamate di sistema:

- `open`,
- `close`,
- `ioctl`,
- `read`,
- `poll`, `epoll` e `select`.

Le funzioni `poll`, `epoll` e `select` sono implementate da un'unica file operation chiamata `poll`. Di seguito è mostrata la struttura `file_operations` utilizzata per registrare il device file event:

```
struct file_operations galil1800_event_fops = {
   .owner = THIS_MODULE,
   .read = galil1800_event_read,
   .write = galil1800_event_write,
   .poll = galil1800_event_poll,
   .unlocked_ioctl = galil1800_event_unlocked_ioctl,
   .open = galil1800_event_open,
   .release = galil1800_event_release,
```

```
    .llseek = no_llseek
};
```

La funzione `galil1800_event_write()` restituisce soltanto l'errore `EPERM`, ad indicare che questo device file non permette in alcun modo la scrittura.

6.1.1 Prestazioni

La gestione di eventi, come ad esempio quelli per il monitoraggio dello stato di occupazione del buffer dei comandi, introduce un'inevitabile diminuzione delle prestazioni del driver. Comunque, come vedremo successivamente in questo capitolo, il buffer che raccoglie gli eventi è stato progettato minimizzando l'impatto sulle prestazioni, in modo tale da non compromettere il corretto funzionamento del driver.

6.1.2 Opzioni

Attraverso un'accurata selezione delle opzioni è possibile ottimizzare la gestione degli eventi al fine di ridurne ulteriormente l'impatto sulle prestazioni.

Innanzitutto, se il device file non viene aperto, il sistema di gestione degli eventi è completamente disabilitato. Questo comporta che il verificarsi degli eventi non viene registrato in alcun modo e l'impatto sulle prestazioni è praticamente nullo.

Attraverso comandi `ioctl` è possibile selezionare il tipo di eventi che devono essere riportati. Ovviamente, maggiore è il numero di eventi da riportare, maggiore sarà il degrado nelle prestazioni. È comunque importante sottilineare come tale degrado non divenga mai così elevato da compromettere il buon funzionamento del driver. Nel paragrafo 6.5 saranno descritti tutti i possibili comandi per la chiamata di sistema `ioctl`.

Quando la gestione eventi è attiva, ogni qualvolta si verifica un particolare evento, quest'ultimo viene registrato nell'apposito buffer: l'evento vi resta finquando l'utente non lo legge attraverso la chiamata di sistema `read` sul

device file event. Il buffer può accumulare una predefinita quantità di eventi, configurabile attraverso il parametro `event_buff_sz`: raggiunto questo valore gli eventi meno recenti vengono sovrascritti.

`event_buff_sz` è configurabile sia attraverso le opzioni di caricamento del modulo, sia attraverso la scrittura di un omonimo file presente nelle directory `/sys/class/galil/event*` del sysfs filesystem.

6.2 Apertura e chiusura del device file

Le implementazioni delle file operation `open` e `release` per il device file event possiedono esattamente le stesse caratteristiche di quelle realizzate per il device file card e discusse nel paragrafo 5.2.

La funzione `galil1800_event_open()` implementa la file operation `open` per il device file event ed è definita nel file `galil1800_event.c`. Di seguito sono riassunte le operazioni svolte:

1. ottiene un puntatore alla struttura `galil1800_state` che descrive il dispositivo e lo rende disponibile per tutte le altre file operation;
2. controlla il modo di apertura del device file;
3. inizializza il buffer degli eventi;
4. se è il primo device file ad essere aperto, completa l'inizializzazione del driver (si veda paragrafo 3.6.6 a pagina 74).

I punti 1 e 4 del precedente elenco sono trattati in modo identico al caso del device file card, quindi si possono trovare maggiori dettagli nel paragrafo 5.2.

Per quanto riguarda il modo di apertura del device file, la sola differenza risiede nel fatto che il device file event è accessibile in sola lettura, quindi il driver controlla che, in ogni istante, ci sia un unico processo che abbia aperto il device file e che questo abbia solo permesso di lettura.

Anche in questo caso la funzione `open` è bloccante, a meno che non venga specificato il flag `O_NONBLOCK`. In merito è valido quanto è stato già detto per il device file card.

Continuando il confronto con il device file card, l'inizializzazione del buffer degli eventi sostituisce quella del buffer dei comandi. I dettagli di questa operazione saranno trattati successivamente in questo capitolo.

La funzione `galil1800_event_release()` implementa la file operation per la chiamata di sistema `close` sul device file event. Le operazioni svolte sono:

- libera le risorse occupate dal buffer degli eventi;
- lancia la funzione `galil1800_clean_controller_open_step()` (si veda paragrafo 3.6.6);
- corregge lo stato delle variabili per il controllo sul modo di apertura del device file.

6.3 Il buffer degli eventi

Il buffer degli eventi è ciò che caratterizza maggiormente il device file event. Esso è accessibile in ogni parte del driver; "segnalare un evento" significa semplicemente aggiungere un dato nel buffer. L'utente, attraverso la chiamata di sistema `read`, può quindi leggerne il contenuto; gli eventi vengono inseriti e prelevati con politica FIFO (First In First Out).

In questo paragrafo vedremo in dettaglio le caratteristiche del buffer degli eventi e la sua implementazione.

6.3.1 Progettazione

Considerato lo scopo del device file event, il buffer degli eventi richiede le seguenti caratteristiche:

- deve sempre essere possibile aggiungere un nuovo evento, quindi il buffer deve essere sovrascrivibile;
- in caso di trabocco (*overflow*) il nuovo evento deve sovrascrivere quello meno recente;
- in caso di trabocco il numero di eventi persi deve essere registrato e riportato all'utente;
- ogni parte del driver può segnalare un evento; sono quindi possibili accessi in scrittura contemporanei;
- l'accesso in lettura è unico (solo la file operation della funzione `read`);
- le operazioni di scrittura devono essere particolarmente veloci, in quanto possono avvenire anche durante fasi critiche del driver (ad esempio, nella funzione che trasferisce i byte dal buffer dei comandi alla FIFO hardware);
- durante le operazioni di lettura deve essere possibile continuare ad aggiungere dati al buffer.

Un buffer veloce, che permetta facilmente di sovrascrivere i dati meno recenti, è facilmente implementabile attraverso un buffer circolare. Quest'ultimo, come si è già detto, può essere utilizzato senza l'uso di sistemi di sincronizzazione, purché abbia un solo utente in lettura ed un solo utente in scrittura: questa è infatti la soluzione adottata per il buffer dei comandi. In questo caso invece, dato che nel buffer degli eventi sono possibili accessi contemporanei in scrittura ed inoltre questi ultimi possono sovrascrivere i dati meno recenti, si è obbligati a fare uso di un sistema di sincronizzazione. Quindi il buffer degli eventi sarà un buffer circolare che però richiede l'acquisizione di un lock per essere modificato.

Nei buffer circolari tipicamente è possibile separare i puntatori modificati in fase di scrittura da quelli modificati in fase di lettura: questo facilita la sincronizzazione degli accessi al buffer. In questo caso però, nelle operazioni

di sovrascrittura dei dati meno recenti è necessario modificare anche i puntatori utilizzati in fase di lettura: ciò obbliga all'uso di un unico lock sia per le letture che per le scritture. Vedremo ora come questo ha un impatto importante nella progettazione del driver.

Affinché le operazioni di scrittura siano sufficientemente rapide, il lock che protegge i puntatori del buffer deve essere trattenuto per brevissimi istanti di tempo.

Per quanto riguarda le aggiunte di dati nel buffer, dovendo essere esse stesse operazioni molto veloci, anche il lock viene ovviamente trattenuto per un breve periodo. Maggiori problemi possono esserci invece nel caso delle letture. Le operazioni necessarie a prelevare un singolo evento dal buffer possono sicuramente essere sviluppate in poche istruzioni: ciò però può non essere sufficiente in quanto la lettura dei dati avviene all'interno della chiamata di sistema `read` e non è certo che ogni evento venga letto singolarmente. Infatti l'utente potrebbe non leggere eventi per un periodo di tempo relativamente lungo e poi prelevarne molti tutti insieme: continue operazioni di lettura (un evento dopo l'altro) terrebbero continuamente occupato il lock e quindi potrebbero rallentare eventuali richieste di scrittura.

Per evitare questa situazione si è sviluppato un buffer suddiviso in blocchi di dimensione prefissata: questi ultimi in seguito saranno chiamati *frame*. A livello logico i frame sono ordinati fra loro in modo circolare, attraverso un array di puntatori; inoltre esiste un frame libero aggiuntivo. In fase di lettura, il primo frame da leggere viene sostituito con quello libero: in tal modo è necessario trattenere il lock solo per un breve istante di tempo, ovvero quello necessario a scambiare i puntatori ai due frame. Una volta effettuata la sostituzione la lettura sul frame estratto può avvenire senza l'uso del lock e quindi non in concorrenza con le scritture. Un schema di esempio per la procedura appena descritta è mostrato in figura 6.1.

Nel resto di questo paragrafo sarà descritta l'implementazione del buffer degli eventi e durante tale discussione se ne potranno evidenziare tutti i dettagli di funzionamento ed ulteriori caratteristiche rilevanti.

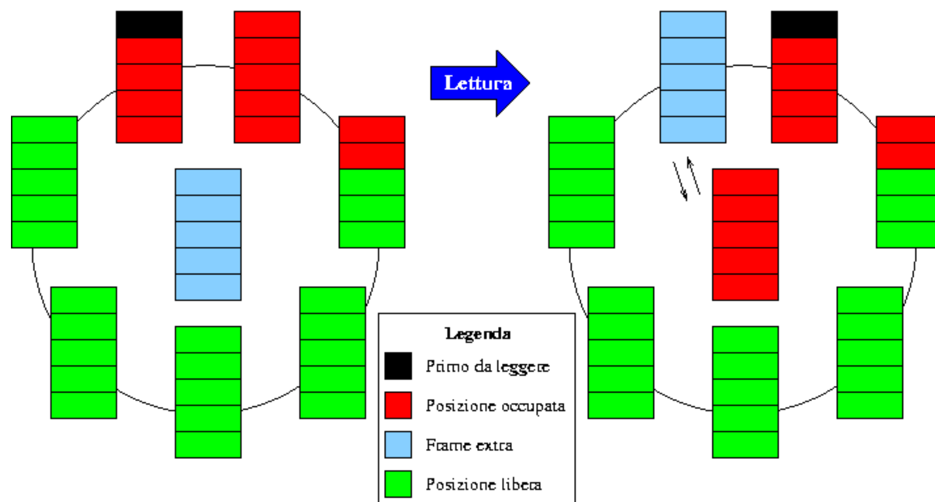

Figura 6.1: Esempio di operazione di lettura nel buffer degli eventi. L'immagine di sinistra mostra il buffer nello stato normale: quando viene richiesta una lettura si sostituisce il frame contenente il primo dato da leggere con il frame extra (vuoto). Quest'ultimo è subito disponibile per essere eventualmente scritto, mentre quello prelevato viene letto senza bisogno del lock: al termine della lettura è pronto per sostituire il frame successivo.

6.3.2 Implementazione

L'implementazione del buffer degli eventi è stata realizzata all'interno dei file `galil1800_event.h` e `galil1800_event.c`. Nell'header file troviamo la definizione della seguente struttura che descrive il buffer:

```
struct galil1800_event_buffer {
   unsigned int size;
   unsigned char opt_flag;
   struct galil1800_event_data *frames;
   struct galil1800_event_data **table;
   unsigned int first_frame;
   struct galil1800_event_data *read_frame;
   unsigned int written;
   unsigned int overwritten_frames;
   spinlock_t lock_pt;
```

```
    wait_queue_head_t read_wait;
    struct class_device_attribute cd_attr_size;
};
```

Segue una descrizione di ogni singolo campo:

- `size`: numero di eventi che possono essere memorizzati nel buffer; è modificabile attraverso l'opzione `event_buff_sz`;
- `opt_flag`: una bitmap di 8 bit che permette di configurare i tipi di eventi che devono essere riportati;
- `frames`: puntatore all'area di memoria che contiene tutti frame che compongono l'intero buffer (compreso il frame extra); indirizza quindi l'inizio dell'intera area di memoria del buffer;
- `table`: array di puntatori che definisce l'ordine con cui utilizzare i frame; il buffer sarà quindi composto dai frame indirizzati da questa tabella;
- `first_frame`: indice nella tabella `table` del frame che contiene il primo evento da leggere; serve quindi ad indicare l'attuale inizio del buffer;
- `read_frame`: puntatore al frame extra da utilizzare in fase di lettura;
- `written`: dimensione corrente del buffer degli eventi; contiene quindi il numero di eventi registrati;
- `overwritten_frames`: numero di frame sovrascritti;
- `lock_pt`: spinlock per sincronizzare le modifiche ai puntatori del buffer; la scelta di uno spinlock, piuttosto che di un semaforo è forzata dal fatto che, come vedremo, il buffer viene scritto anche da una funzione eseguita in un tasklet;[1]

[1] Vale la pena ricordare che i tasklet sono eseguiti nel contesto delle interruzioni software, quindi non possono bloccare.

- `read_wait`: wait queue per realizzare la chiamata di sistema `read` in modo bloccante;
- `cd_attr_size`: permette di esportare il campo `size` nel sysfs filesystem creando il file `event_buff_sz`.

Il campo `size` indica il numero di eventi memorizzabili. Un singolo evento è rappresentato da due byte definiti nella seguente struttura dati:

```
struct galil1800_event_data {
   unsigned char type;
   unsigned char data;
};
```

Il campo `type` permette di specificare un massimo di 256 (2^8) diversi tipi di eventi; al momento il driver ne implementa solo 6. Il campo `data` contiene l'informazione relativa all'evento indicato. La tabella 6.1 riassume gli eventi attualmente supportati nel driver.

Descrizione	`type`	**Significato di** `data`
Interruzione	0	Byte di stato dell'interruzione
Occupazione del buffer dei comandi:		
$< 10\%$	1	% effettiva
$< 20\%$	2	% effettiva
$< 50\%$	3	% effettiva
$> 90\%$	4	% effettiva
Buffer sovrascritto	8	Numero di eventi sovrascritti oppure 0 se più di 255

Tabella 6.1: Possibili tipi di eventi riportati dal device file event: `type` e `data` si riferiscono ai campi della struttura `galil1800_event_data`.

Come vedremo in seguito, la funzione `read` del device file event per ogni evento esporta nello spazio utente due byte, ovvero una struttura di tipo `galil1800_event_data`. Per questo tale struttura è stata definita nel file

`galil1800.h`, così da poter essere utilizzata anche nelle applicazioni *user-mode*.

Il campo `frames` contiene tutta l'area di memoria necessaria a gestire il buffer. Fissato un certo valore di `size`, vedremo ora come calcolare la quantità di byte da allocare per questo puntatore, partendo dalle seguenti considerazioni:

- un evento è rappresentato dal tipo di dato `galil1800_event_data`, ovvero due byte;
- il buffer deve essere diviso in frame ed è necessario un frame extra da utilizzare nelle operazioni di lettura;
- la dimensione dei frame è fissata staticamente attraverso la macro `GALIL1800_EVENT_BUFF_FRAME_SZ` ed è pari a 64 eventi.

Il numero di byte da allocare in `frames` è quindi:

```
(size + GALIL1800_EVENT_BUFF_FRAME_SZ) *
                              sizeof(galil1800_event_data)
```

ovvero:

```
(size + 64) * 2
```

Il campo `table` è un array la cui dimensione corrisponde al numero di frame in cui è diviso il buffer (escluso quello extra). Quindi tale valore sarà pari a:

```
size / GALIL1800_EVENT_BUFF_FRAME_SZ
```

Questi puntatori, durante l'inizializzazione del buffer, vengono impostati in modo sequenziale partendo dal valore contenuto in `frames` e quindi procedono ad incrementi di `GALIL1800_EVENT_BUFF_FRAME_SZ`.

Il puntatore `read_frame` viene inizialmente impostato sull'ultimo frame contenuto nell'area di memoria `frames`.

Nella file operation della chiamata di sistema `open` viene utilizzata la funzione `init_galil1800_event_buffer_open_dev()`, che realizza la fase di inizializzazione appena descritta.

Tipi di eventi

Il campo `opt_flag` è una mappa di 8 bit che permette di selezionare alcuni tipi di eventi che devono essere riportati nel buffer. Infatti, gli eventi con codice numerico (valore del campo `type` indicato in tabella 6.1) compreso tra 0 e 7 devono essere esplicitamente abilitati impostando ad 1 il corrispondente bit del campo `opt_flag`. In particolare il primo bit è relativo all'evento 0 (interruzione), il secondo all'evento 1 (buffer dei comandi occupato per meno del 10%), etc. . . Per default sono abilitati solo gli eventi 0 e 2, ovvero le interruzioni ed una notifica quando l'occupazione del buffer dei comandi scende sotto il 20%.

È possibile modificare lo stato di attivazione dei vari tipi di eventi attraverso comandi della chiamata di sistema `ioctl`, come vedremo nel paragrafo 6.5.

Gli eventi con un codice numerico maggiore o uguale ad 8 non sono disabilitabili: questi ultimi sono quindi sempre riportati nel buffer eventi. Al momento è implementata solo la notifica del numero di eventi persi a causa della necessità di sovrascrivere il buffer.

Maggiore è la quantità di eventi che devono essere riportati, maggiore può essere l'impatto sulle prestazioni del driver. È consigliato quindi che l'utente scelga attentamente i tipi di eventi a cui è interessato.

Aggiungere un evento nel buffer

Nel file `galil1800_event.h` sono definite alcune macro che permettono di inserire nuovi eventi nel buffer. Vediamone un esempio:

```
#define galil1800_event_buff_add_wbuff_less_10p(_buff_p, _info)  \
                  __galil1800_event_buff_lock_and_add(_buff_p,  \
                         _info, GALIL1800_EVENT_WBUFF_LESS_10P)
```

Come si può notare la macro è già predisposta per un particolare tipo di evento, ovvero il tipo 2 della tabella 6.1; in particolare esiste una macro per ogni valore, fatta eccezione per l'evento "interruzione" che viene trattato

in modo diverso e sarà dettagliatamente discusso nel paragrafo 6.6. Queste macro sono distribuite all'interno del codice, nei vari punti dove è possibile controllare l'eventuale verificarsi dell'evento. In ogni caso, quando l'intera gestione eventi oppure il singolo evento sono disabilitati tali funzioni vengono ugualmente chiamate, ma come definito nelle specifiche non devono introdurre degradi di prestazioni. Tale caratteristica viene rispettata poiché, grazie alla mappa di bit `opt_flag` verificare lo stato di attivazione dei vari eventi è molto veloce. La macro precedentemente mostrata (e così anche le sue analoghe) ne richiama un'altra comune ad ogni tipo di evento:

```
#define                                                                \
__galil1800_event_buff_lock_and_add(_buff_p, _info, _event_type)      \
({                                                                     \
   int __ret;                                                          \
   if ((__ret=galil1800_event_is_active(_buff_p, _event_type))) {      \
      spin_lock_bh(&(_buff_p)->lock_pt);                               \
      /* I must retest because the event device release */            \
      /* function can free the buffer! */                             \
      if ((__ret=galil1800_event_is_active(_buff_p, _event_type)))    \
         __galil1800_event_buff_add(_buff_p,_info,_event_type);       \
      spin_unlock_bh(&(_buff_p)->lock_pt);                             \
      wake_up_interruptible_sync(&(_buff_p)->read_wait);               \
   }                                                                   \
   __ret;                                                              \
})
```

Questa macro accetta i seguenti argomenti:

- `_buff_p` è il puntatore alla struttura `galil1800_event_buffer`;
- `_info` e `_event_type` sono rispettivamente i valori da porre nei campi `data` e `type` della struttura `galil1800_event_data`.

Vediamo quali sono le operazioni che svolge:

- controlla se il tipo di evento `_event_type` deve essere riportato nel buffer, ovvero verifica che il bit di `opt_flag` corrispondente a tale evento

sia impostato ad uno; vale la pena notare che qualora il buffer sia interamente disattivato tutti i bit di `opt_flag` sono posti a zero; il test viene velocemente effettuato con la seguente macro:

```
#define galil1800_event_is_active(_buff_p, _event_type)    \
      test_bit((_event_type), (void*)&(_buff_p)->opt_flag)
```

se il valore di ritorno è 1:

  – si acquisisce lo spinlock;[2]
  – si effettua nuovamente il test per l'attivazione dell'evento e se ancora positivo si aggiunge l'evento;
  – si rilascia lo spinlock;
  – si riattivano eventuali processi bloccati sulla chiamata di sistema `read` del device file event;

- la macro assume valore 1 se l'evento viene aggiunto, 0 altrimenti.

Il primo test sullo stato di attivazione dell'evento viene effettuato prima di richiedere il lock: in questo modo, quando il buffer o l'evento sono disattivati l'impatto sulle prestazioni si riduce al test di un singolo bit. Comunque non avendo acquisito il lock, lo stato di attivazione può cambiare, oppure addirittura il device file event potrebbe essere chiuso e quindi l'intero buffer potrebbe venire deallocato. Tutto ciò obbliga ad un secondo test, prima dell'effettiva scrittura nel buffer. Il risultato finale è che nei casi in cui l'aggiunta dell'evento va a buon fine si esegue un istruzione in più (il primo test), mentre quando l'evento è disattivato se ne esegue una in meno (l'acquisizione del lock). Si è scelto di privilegiare questo secondo caso fondamentalmente per due motivi:

[2]Le funzioni che agiscono sullo spinlock hanno il suffisso `_bh`: in questo modo, quando viene acquisito lo spinlock sono anche disabilitate le interruzioni software. Ciò è necessario in quanto il buffer è acceduto anche attraverso un tasklet.

- nella maggior parte delle situazioni non tutti i tipi di eventi dovrebbero essere attivi, quindi i test negativi dovrebbero essere più frequenti;
- richiedendo che un evento sia segnalato si è consapevoli che sarà introdotto un piccolo degrado in prestazioni; al contrario disattivandolo si deve esser certi che il degrado non ci sia.

L'effettivo inserimento di un evento nel buffer avviene attraverso la macro `__galil1800_event_buff_add()`. Vediamo come è stata implementata e quindi quali sono le operazioni che svolge:

```
#define __galil1800_event_buff_add(_buff_p, _info, _info_type)  \
do {                                                            \
   struct galil1800_event_data *next_to_write;                  \
   if ((_buff_p)->written == (_buff_p)->size) {                 \
      next_to_write = (_buff_p)->table[(_buff_p)->first_frame]; \
      galil1800_event_buff_first_frame_inc(_buff_p);            \
      (_buff_p)->written -= GALIL1800_EVENT_BUFF_FRAME_SZ;      \
      (_buff_p)->overwritten_frames++;                          \
   }                                                            \
   else {                                                       \
      next_to_write = (_buff_p)->table [                        \
         ((_buff_p)->first_frame +                              \
         ((_buff_p)->written/GALIL1800_EVENT_BUFF_FRAME_SZ)) %  \
         GALIL1800_EVENT_BUFF_TABLE_SZ(_buff_p) ]               \
         + ((_buff_p)->written%GALIL1800_EVENT_BUFF_FRAME_SZ);  \
   }                                                            \
   next_to_write->type = (_info_type);                          \
   next_to_write->data = (_info);                               \
   barrier();                                                   \
   (_buff_p)->written++;                                        \
}while(0);
```

1. se il buffer è pieno (`written==size`) è necessario sovrascrivere gli eventi meno recenti; a questo scopo si è scelto di liberare un intero frame

in quanto, a causa del sistema di lettura, sarebbe troppo complicato mantenere un frame misto contenente sia gli ultimi dati che i primi da leggere; per la sovrascrittura si svolgono le seguenti operazioni:

- si imposta il puntatore `next_to_write`, all'inizio del frame con i primi dati da leggere, ovvero i meno recenti;
- viene incrementato l'indice del primo frame, ovvero `first_frame`;
- si decrementa il numero di eventi presenti nel buffer esattamente della dimensione di un frame;
- si incrementa il contatore di frame sovrascritti cosi da poter notificare all'utente il numero di eventi persi;

se non è necessario sovrascrivere (`written < size`):

- viene calcolato il puntatore al prossimo evento da scrivere calcolando prima l'indice del frame e quindi l'offset al suo interno;

2. si aggiunge l'evento;

3. si incrementa il numero di eventi scritti; prima di questa operazione l'istruzione `barrier()` assicura che tutte le precedenti operazioni siano state eseguite e quindi che il nuovo evento sia effettivamente pronto per essere prelevato dal buffer.

Inserimento degli eventi relativi al buffer dei comandi

In questa versione del driver quattro eventi forniscono informazioni sullo stato di occupazione del buffer dei comandi. La macro mostrata di seguito è inserita all'interno delle funzioni che gestiscono il buffer dei comandi:

```
#define galil1800_add_wbuff_event(_s) do {              \
   unsigned char _wbuff_flag =                          \
      galil1800_event_get_wbuff_flag(&(_s)->event_buff); \
   if(_wbuff_flag) /* one or more event report enabled */ \
      __galil1800_add_wbuff_event(_s, _wbuff_flag);     \
}while(0)
```

`_wbuff_flag` viene estratto da `opt_flag`, azzerando tutti i bit che non riguardano eventi relativi al buffer dei comandi. Se nessuno dei quattro eventi è attivo questa macro non svolge alcuna operazione, altrimenti chiama la funzione `__galil1800_add_wbuff_event()` che calcola la percentuale di occupazione del buffer e, qualora necessario, aggiungere il nuovo evento. Questa funzione fa uso dei campi `last_event_opt_flag` e `last_size_p10` della struttura `galil1800_card_wbuffer` per evitare di aggiungere più volte consecutivamente lo stesso tipo di evento: ciò significa, ad esempio, che se il buffer dei comandi scende sotto il 10%, l'evento viene segnalato solo quando questa soglia viene oltrepassata. Di seguito viene mostrata l'implementazione della funzione `__galil1800_add_wbuff_event()`:

```
static inline
void __galil1800_add_wbuff_event(struct galil1800_state *s,
                                    unsigned char wbuff_flag)
{
   unsigned char curr_size_p10; /*current size in per 10 value*/
   unsigned char curr_size_p100;/*current size in per 100 value*/
   curr_size_p100 =
     (GALIL1800_CARD_WB_CURRENT_SIZE(&s->card_wbuff)*100) /
                                          (s->card_wbuff.size);
   curr_size_p10 = curr_size_p100/10;
   if(s->card_wbuff.last_event_opt_flag==wbuff_flag &&
                  s->card_wbuff.last_size_p10==curr_size_p10)
      return; /* Already reported */
   s->card_wbuff.last_size_p10 = curr_size_p10;
   s->card_wbuff.last_event_opt_flag=wbuff_flag;
   switch(curr_size_p10) {
      case 9:
      case 10:
         galil1800_event_buff_add_wbuff_over_90p(
                                &s->event_buff, curr_size_p100);
         break;
      case 0:
```

```
            if(galil1800_event_buff_add_wbuff_less_10p(
                                    &s->event_buff, curr_size_p100))
               break;
         case 1:
            if(galil1800_event_buff_add_wbuff_less_20p(
                                    &s->event_buff, curr_size_p100))
               break;
         case 2:
         case 3:
         case 4:
            galil1800_event_buff_add_wbuff_less_50p(
                                    &s->event_buff, curr_size_p100);
      };
   }
```

6.4 Lettura degli eventi

L'implementazione della file operation per la chiamata di sistema `read` nel device file event segue lo stesso modello utilizzato per il device file card (si veda il paragrafo 5.4) ed è realizzata dalla funzione:

```
static ssize_t galil1800_event_read(struct file * filp,
                   char __user * buf, size_t size, loff_t *poff)
```

La prima parte di questa funzione calcola il numero massimo di byte da leggere, quindi alloca un buffer temporaneo (`_buf`) che verrà poi copiato nello spazio utente. Successivamente se il buffer degli eventi è vuoto si mette in attesa nella wait queue `read_wait`, definita nella struttura `galil1800_event_buffer`, e vi resta finché non viene segnalato almeno un evento.

Quando vi sono eventi da leggere il processo si sblocca e viene eseguita la seguente porzione di codice, dove `buff_p` è un puntatore al buffer degli eventi e `request` è la dimensione del buffer temporaneo `_buf`:

```
...
do {
   galil1800_event_buff_prepare_to_read(buff_p,
                        &lost_events, &written_events);
   if(written_events > 0) {
      if(lost_events > 0) {
         lost_events *= GALIL1800_EVENT_BUFF_FRAME_SZ;
         _buf[rc++] = GALIL1800_EVENT_BUFF_OVERWRITTEN;
         _buf[rc++] = (unsigned char)(lost_events < 256 ?
                                   lost_events : 0);
      }
      bytes = written_events < GALIL1800_EVENT_BUFF_FRAME_SZ ?
              written_events % GALIL1800_EVENT_BUFF_FRAME_SZ :
                              GALIL1800_EVENT_BUFF_FRAME_SZ;
      bytes *= sizeof(struct galil1800_event_data);
      memcpy(_buf + rc, buff_p->read_frame, bytes);
      rc += bytes;
   }
   else
      break;
}while( (rc + (GALIL1800_EVENT_BUFF_FRAME_SZ + 1) *
         sizeof(struct galil1800_event_data)) <= request );
if(rc > 0) {
   if(copy_to_user(buf, _buf, rc))
      rc = -EFAULT;
}
...
```

La funzione `galil1800_event_buff_prepare_to_read()` è così definita:

```
static void galil1800_event_buff_prepare_to_read(
         struct galil1800_event_buffer *buff_p,
         unsigned int *temp_overw_f, unsigned int *temp_written)
{
   struct galil1800_event_data *temp;
```

```
    spin_lock_bh(&buff_p->lock_pt);
    *temp_overw_f = buff_p->overwritten_frames;
    *temp_written = buff_p->written;
    if(buff_p->written > 0) {
       buff_p->overwritten_frames = 0;
       temp = buff_p->read_frame;
       barrier();
       buff_p->read_frame = buff_p->table[buff_p->first_frame];
       barrier();
       buff_p->table[buff_p->first_frame] = temp;
       if(buff_p->written >= GALIL1800_EVENT_BUFF_FRAME_SZ) {
          buff_p->written -= GALIL1800_EVENT_BUFF_FRAME_SZ;
          galil1800_event_buff_first_frame_inc(buff_p);
       }
       else
          buff_p->written = 0;
    }
    spin_unlock_bh(&buff_p->lock_pt);
}
```

Segue una descrizione delle operazioni appena mostrate:

- si acquisisce il lock del buffer degli eventi;
- vengono salvate in variabili temporanee i valori dei campi `written` e `overwritten_frames` della struttura `galil1800_event_buffer`;
- se `written` è maggiore di zero, vi sono dati da leggere quindi:
  - si azzera `overwritten_frames` (il suo valore è già stato salvato in una variabile temporanea ed il numero di eventi persi verrà notificato all'utente);
  - il frame extra (puntato `read_frame`) viene scambiato con il primo frame del buffer (`table[first_frame]`);
  - se il buffer aveva il primo frame interamente occupato:

  - `written` viene decrementato del numero di eventi contenuto in un frame;
  - `first_index` viene incrementato;

  altrimenti:

  - `written` viene azzerato;

- si rilascia il lock; dopo queste poche istruzioni si può leggere l'intero frame permettendo ad altre parti del driver di continuare ad aggiungere eventi nel buffer;
- se `overwritten_frames` era positivo si aggiungono in `_buf` due byte per segnalare la sovrascrittura: il primo contiene il numero 8 che identifica l'evento "sovrascrittura", il secondo è il numero di eventi sovrascritti, oppure 0 se quest'ultimo supera 255;
- si calcola il numero di byte occupati nel frame puntato da `read_frame`, quindi quest'ultimi vengono copiati in `_buf`;

Queste operazioni vengono ripetute finché vi sono eventi da leggere oppure non viene riempito il buffer temporaneo `_buf`. Il contenuto di `_buf` viene successivamente copiato nello spazio utente, così come avveniva nel device file card.

È importante notare che poiché un evento è descritto con due byte, una chiamata `read` sul device file event restituirà sempre multipli di due byte che andranno interpretati come se fossero una sequenza di strutture `galil1800_event_data`.

6.4.1 File operation `poll`

La lettura nel device file event è bloccante e ciò è realizzato attraverso una wait queue. Questo ha reso molto semplice l'implementazione della file operation `poll` che permette di applicare al device file event le chiamate di sistema `poll`, `epoll` e `select`. Di seguito ne vediamo l'implementazione:

```
static unsigned int
       galil1800_event_poll(struct file *filp, poll_table *wait)
{
   struct galil1800_state *s =
      (struct galil1800_state*) filp->private_data;
   unsigned int mask = 0;
   poll_wait(filp, &(s->event_buff.read_wait), wait);
   if(galil1800_event_buff_not_empty(&(s->event_buff)))
      mask = POLLIN | POLLRDNORM;
   return mask;
}
```

Quando viene usata una delle chiamate di sistema appena citate, viene lanciata questa funzione e controllato il valore di `mask`. Se quest'ultimo è zero il kernel blocca il processo nella wait queue `read_wait` finché non viene eseguita una funzione `wake_up` su questa stessa coda; si può notare nel paragrafo 6.3.2 come ciò avvenga per ogni evento aggiunto nel buffer.

6.5 Comandi della chiamata di sistema `ioctl`

Attraverso la chiamata di sistema `ioctl` è possibile selezionare i tipi di eventi che devono essere riportati dal device file event. A tale scopo sono disponibili sei comandi nei quali l'ultimo argomento della chiamata `ioctl` deve essere un intero che vale 0 per disattivare l'evento, 1 per attivarlo. Segue l'elenco dei comandi disponibili affiancato dalla descrizione dell'evento attivato o disattivato:

- `GALIL1800_IOCTL_EVENT_ALL_ACTIVE`: tutti gli eventi;
- `GALIL1800_IOCTL_EVENT_IRQ`: interruzioni;
- `GALIL1800_IOCTL_EVENT_WBUFF_LESS_10P`: occupazione del buffer dei comandi inferiore al 10%;

- `GALIL1800_IOCTL_EVENT_WBUFF_LESS_20P`: occupazione del buffer dei comandi inferiore al 20%;
- `GALIL1800_IOCTL_EVENT_WBUFF_LESS_50P`: occupazione del buffer dei comandi inferiore al 50%;
- `GALIL1800_IOCTL_EVENT_WBUFF_OVER_90P`: occupazione del buffer dei comandi superiore al 90%.

La file operation è implementata con uno `switch`, tipico per le chiamate `ioctl`. Di seguito, a titolo di esempio, ne viene mostrato un caso:

```
...
case GALIL1800_IOCTL_EVENT_IRQ:
{
   if((void*)arg==NULL) /* disable */
      galil1800_event_disactive(buff_p, GALIL1800_EVENT_IRQ);
   else /* enable */
      galil1800_event_active(buff_p, GALIL1800_EVENT_IRQ);
}
break;
...
```

Per completezza si riporta anche la definizione delle macro che attivano e disattivano il report dell'evento:

```
#define galil1800_event_active(_buff_p, _event_type)        \
      set_bit((_event_type), (void*)&(_buff_p)->opt_flag)

#define galil1800_event_disactive(_buff_p, _event_type)     \
      clear_bit((_event_type), (void*)&(_buff_p)->opt_flag)
```

6.6 Gestione delle interruzioni

Nel paragrafo 1.4.1 è stato descritto il modo in cui i dispositivi Galil DMC 1800 usano le interruzioni. In questo paragrafo vedremo quali operazioni è

necessario svolgere ogni volta che il controller segnala un'interruzione e come i *byte di stato* elencati in tabella 1.2 di pagina 20 vengano inseriti nel buffer degli eventi.

6.6.1 Inizializzazione

Nelle file operation delle chiamate di sistema `open` di tutti e tre i device file viene chiamata la funzione `galil1800_init_controller_open_step()`, già descritta nel paragrafo 3.6.6, che qualora si stia aprendo il primo device file esegue la funzione `__galil1800_init_controller_open_step()`. Quest'ultima si occupa di registrare una linea di interruzione per il controller Galil DMC 1800. Rivediamo la parte di codice che svolge questa registrazione:

```
request_irq(s->irq, galil1800_interrupt, SA_SHIRQ,
                            GALIL1800_MODULE_NAME, s);
```

`s` è un puntatore alla struttura `galil1800_state`, quindi `s->irq` è la linea di interruzione assegnata al dispositivo. `galil1800_interrupt()` è la funzione che implementa il gestore di interruzione. In particolare questa funzione realizza ciò che tipicamente viene chiamato *top half handler*, ovvero quella parte del gestore di interruzione che deve essere eseguita il più rapidamente possibile; la gestione dell'interruzione viene poi completata nel *bottom half handler*, implementato dalla funzione `galil1800_bottom_half_interrupt()` ed eseguito attraverso un tasklet. `SA_SHIRQ` sta ad indicare che la linea di interruzione può essere condivisa (*shared interrupt*) con altri dispositivi hardware.

Altre operazioni svolte immediatamente dopo aver registrato un gestore di interruzione sono le seguenti:

```
init_galil1800_irq_stack(&s->irq_stack);
tasklet_enable(&s->irq_tasklet);
galil1800_enable_IRQ(s);
```

Ovvero:

1. viene inizializzato l'`irq_stack`, cioè un piccolo buffer circolare dove vengono temporaneamente salvati i byte di stato;

2. si abilita il tasklet responsabile dell'esecuzione del bottom half handler;

3. vengono attivate le interruzioni nel dispositivo Galil DMC 1800.

Per completezza viene mostrato cosa avviene quando l'ultimo device file viene chiuso. In tal caso viene eseguita la seguente funzione:

```
static void
__galil1800_clean_controller_open_step(struct galil1800_state *s)
{
   galil1800_disable_IRQ(s);
   synchronize_irq(s->irq);
   free_irq(s->irq, s);
   tasklet_disable(&s->irq_tasklet);
}
```

Le quattro funzioni utilizzate eseguono le seguenti operazioni:

- sono disabilitate le interruzioni nel controller Galil DMC 1800;
- `synchronize_irq()` attende che eventuali gestori di interruzione per la linea `s->irq` in attesa su altre CPU vengano eseguiti;
- si deregistra il gestore di interruzione;
- viene disabilitato il tasklet per il bottom half handler.

6.6.2 Stack delle interruzioni

L'obiettivo finale della gestione dell'interruzione in questo driver è ottenere il byte di stato ed esportarlo alle applicazioni utente. Abbiamo visto che il buffer degli eventi dispone di un apposito tipo di evento, quindi in realtà l'operazione che bisogna svolgere è aggiungere l'evento "interruzione" (`type` uguale a 0) nel buffer.

Sebbene, nella maggior parte delle situazioni, le operazioni per aggiungere un evento nel buffer possono essere considerate abbastanza veloci, non sono sufficientemente rapide per essere eseguite all'interno di un gestore di

interruzione, o meglio nel top half handler. Innanzi tutto per aggiungere un evento è necessario acquisire un spinlock: se quest'ultimo è occupato il gestore viene ritardato. Inoltre il numero di istruzioni per l'aggiunta di un evento nel buffer è eccessivo per essere svolto all'interno di un top half handler; infine la funzione `wake_up_interruptible_sync()` è anch'essa troppo complessa.

La soluzione adottata è stata quella di definire un buffer provvisorio per i byte di stato, l'*irq stack*. Quest'ultimo è infatti un semplice buffer circolare, non sovrascrivibile, che risulta quindi molto veloce: il top half interrupt handler vi aggiunge i byte di stato e successivamente attiva il tasklet per il bottom half handler che si occuperà di trasferire il contenuto dello stack nel buffer degli eventi.

Di seguito viene mostrata la definizione dell'irq stack, seguita da una descrizione dei suoi campi:

```
#define GALIL1800_IRQ_STACK_SZ 2048

struct galil1800_irq_stack {
   unsigned char data[GALIL1800_IRQ_STACK_SZ];
   unsigned char *head; /* next to write */
   unsigned char *tail; /* next to read */
};
```

- `data`: il buffer per i byte di stato;
- `head`: puntatore al primo byte libero;
- `tail`: puntatore al primo byte da leggere.

Anche se l'array `data` è lungo 2048 byte in realtà possono essere memorizzati solo 2047 byte di stato. Lo spreco di un byte è necessario per implementare correttamente il buffer circolare: infatti quando `head==tail` il buffer è considerato vuoto, mentre viene considerato pieno se `head+1==tail`.

Poiché solo il top half handler può aggiungere dati nello stack e solo il bottom half handler può leggerne, allora non è necessario l'uso di lock; in realtà

lo stesso top half handler può essere eseguito contemporaneamente su diverse CPU, ma vedremo che questo problema viene facilmente risolto. Invece per quanto riguarda il bottom half handler, poiché le funzioni assegnate ad uno stesso tasklet sono serializzate non vi è alcuna possibilità di concorrenza.

Nei prossimi paragrafi vedremo l'implementazione dei gestori di interruzione e contestualmente sarà mostrato anche l'uso dello stack.

6.6.3 Top half handler

Una volta registrato un gestore su una data linea di IRQ, ogni qualvolta viene segnalata un'interruzione su quella linea la funzione registrata come gestore viene eseguita. Vediamo l'implementazione del top half handler in questo driver:

```
irqreturn_t
galil1800_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   struct galil1800_state *s = dev_id;
   unsigned long flags;
   unsigned char status, t;

   t = inb(GALIL1800_CONTROL_REG(s));
   if(!(t & GALIL1800_IRQ_ENABLE_FLAG) ||
                                 !(t & GALIL1800_IRQ_STATUS_FLAG))
      return IRQ_NONE; /* shared interrupt raised by someone else */

   status = inb(GALIL1800_IRQ_REG(s));

   /* Here add in stack don't need to acquire the spinlock... */
   galil1800_irq_stack_add(&s->irq_stack, status);

   spin_lock_irqsave(&s->hwlock, flags);
   t = inb(GALIL1800_CONTROL_REG(s)); /* need to re-read!! */
   outb(t | GALIL1800_IRQ_STATUS_FLAG, GALIL1800_CONTROL_REG(s));
```

```
    spin_unlock_irqrestore(&s->hwlock, flags);

    /* Schedule request for bottom half handling */
    tasklet_schedule(&s->irq_tasklet);

    return IRQ_HANDLED;
}
```

La maggior parte delle istruzioni eseguite da questa funzione seguono le indicazioni fornite dalle specifiche tecniche delle schede Galil DMC 1800 [24].

La prima operazione legge il contenuto del registro di controllo e testa il valore di due bit: con questa operazione ci si accerta che il controller abbia le interruzioni abilitate e che sia stato lui ad emettere questa interruzione. Infatti poiché il controller supporta la condivisione della linea di IRQ questo gestore viene chiamato anche quando sono altri dispositivi ad attivare la stessa linea. In questo caso deve semplicemente ritornare `IRQ_NONE` e terminare il gestore.

Se l'interruzione è stata generata dal dispositivo controllato dal driver, allora viene letto il byte di stato dal registro di IRQ e salvato nella variabile `status`. Anche dopo questa operazione, finché non viene azzerato il bit `GALIL1800_IRQ_STATUS_FLAG` nel registro di controllo, il dispositivo non è in grado di emettere altre interruzioni: ciò permette di inserire il valore contenuto in `status` nell'irq stack, senza possibilità di concorrenza da parte di un altro gestore.

Vediamo come è implementata la macro `galil1800_irq_stack_add()`:

```
#define galil1800_irq_stack_add(_pt, _status_irq) do {        \
    unsigned char *new_head =                                 \
    (_pt)->head + 1 < (_pt)->data + GALIL1800_IRQ_STACK_SZ ?  \
                             (_pt)->head + 1 : (_pt)->data;   \
    if(new_head != (_pt)->tail) {                             \
        *((_pt)->head)=(_status_irq);                         \
        (_pt)->head = new_head;                               \
    }                                                         \
```

```
}while(0)
```

Come possiamo vedere se lo stack è pieno il byte di stato viene perso. In ogni caso con uno stack di 2048 byte, la possibilità di perdere un byte di stato è trascurabile.

A questo punto, seguendo le specifiche, si avvisa la scheda che l'interruzione è stata gestita e che quindi ne possono essere emesse di nuove.

L'ultima istruzione richiede l'esecuzione del tasklet, ovvero del bottom half handler, che sarà descritto nel prossimo paragrafo.

6.6.4 Bottom half handler

Compito del bottom half handler è quello di leggere l'irq stack e conseguentemente inserire i nuovi dati nel buffer degli eventi. Vediamo come è stato implementato:

```
void galil1800_bottom_half_interrupt(unsigned long data) {
   struct galil1800_state *s = (struct galil1800_state *)data;
   struct galil1800_event_buffer *buff_p = &s->event_buff;

   /* Now add irq event from irq_stack to event buffer */
   if (galil1800_event_is_active(buff_p,GALIL1800_EVENT_IRQ)) {
      spin_lock_bh(&buff_p->lock_pt);
      if (galil1800_event_is_active(buff_p,GALIL1800_EVENT_IRQ)) {
         for_each_galil1800_irq_stack(&s->irq_stack) {
               __galil1800_event_buff_add(buff_p,
                              *(s->irq_stack.tail),
                              GALIL1800_EVENT_IRQ);
         }
      } else
         s->irq_stack.tail = s->irq_stack.head;
      spin_unlock_bh(&buff_p->lock_pt);
      wake_up_interruptible_sync(&buff_p->read_wait);
   } else
```

```
        s->irq_stack.tail = s->irq_stack.head;
}
```

Praticamente le operazioni preliminari e finali per l'aggiunta di un nuovo evento sono le stesse già descritte nel paragrafo 6.3.2. L'unica eccezione è che qualora la segnalazione dell'evento "interruzione" non sia attiva lo stack viene svuotato, ovvero si assegna `tail=head`.

Per inserisce gli eventi nel buffer si usa la seguente macro che scorre l'intero stack:

```
#define for_each_galil1800_irq_stack(_pt)                     \
   for( ; (_pt)->head != (_pt)->tail; (_pt)->tail =           \
      ((_pt)->tail + 1 < (_pt)->data + GALIL1800_IRQ_STACK_SZ ? \
                             (_pt)->tail + 1 : (_pt)->data) )  \
```

Con questa istruzione viene incrementato `tail` finché il buffer non è vuoto.

Capitolo 7

Lettura della FIFO secondaria

Questo capitolo descriverà le caratteristiche del device file *info*. Come vedremo tale device file è molto più semplice degli altri due già discussi: ciò è dovuto al fatto che il suo unico scopo è quello di permettere la lettura della FIFO secondaria dei dispositivi Galil DMC 1800.

7.1 Introduzione al device file info

I dispositivi Galil DMC 1800 dispongono, oltre alle FIFO primarie, di un terzo buffer hardware di sola lettura, la FIFO secondaria, già discussa nei paragrafi 1.4 e 4.4. Questa FIFO permette all'utente di ottenere molte informazioni sullo stato di tutti gli assi controllati.

All'avvio del dispositivo la FIFO secondaria è disabilitata. Attraverso il comando Galil `DUn` l'utente può richiedere che la FIFO venga aggiornata ogni `n` secondi; `DU0` interrompe l'aggiornamento [25].

Ogni lettura nella FIFO secondaria restituisce sempre lo stesso numero di byte, ovvero l'intera FIFO; nelle specifiche tecniche dei controller DMC 1800 [24] è disponibile una mappa che indica il significato di ciascun byte restituito.

Considerate le caratteristiche della FIFO secondaria si possono ora dedurre quelle richieste al device file info:

- **sola lettura**: essendo la FIFO secondaria di sola lettura, non è necessario implementare la chiamata di sistema `write`;
- **lettura non bloccante**: considerato quanto descritto nel paragrafo 4.4 è evidente come ogni tentativo di lettura della FIFO secondaria vada sempre a buon fine; non ha quindi senso implementare la lettura bloccante;
- **utente unico**: la chiamata di sistema `open` dovrà essere implementata in modo simile a quella del device file `event` e permettere un solo utente alla volta.

Di seguito viene mostrata la struttura `file_operations` utilizzata per registrare il device file info:

```
struct file_operations galil1800_info_fops = {
    .owner = THIS_MODULE,
    .read = galil1800_info_read,
    .write = galil1800_info_write,
    .open = galil1800_info_open,
    .release = galil1800_info_release,
    .unlocked_ioctl = galil1800_info_unlocked_ioctl,
    .llseek = no_llseek
};
```

Come per il device file event, la funzione `galil1800_info_write()` restituisce soltanto l'errore `EPERM`, ad indicare che questo device file non permette la scrittura.

7.1.1 Opzioni

Il device file info dispone di un unico parametro configurabile, ovvero il timeout `fifo2_timeout`, il cui utilizzo è descritto nel paragrafo 4.4.

Il suo valore di default è 20 ms, ma è modificabile attraverso:

- opzioni di caricamento del modulo;

- scrittura sui file `/sys/class/galil/info*/fifo2_timeout` del *sysfs* filesystem;
- il comando `GALIL1800_IOCTL_SET_FIFO2_TIMEOUT` della chiamata di sistema `ioctl` (l'argomento deve indicare il timeout espresso in millisecondi).

Quest'ultimo comando è anche l'unico disponibile per la chiamata di sistema `ioctl` del device file info.

7.2 Apertura e chiusura del device file

Il driver deve permettere l'apertura del device file info ad un solo utente alla volta, esattamente come per il device file event. Quindi l'implementazione delle file operation `open` e `release` segue esattamente lo stesso schema di quelle realizzate per il device file event.

La funzione `galil1800_info_open()` definita nel file `galil1800_info.c` implementa la file operation `open`. Di seguito sono riassunte le operazioni svolte:

1. ottiene un puntatore alla struttura `galil1800_state` che descrive il dispositivo e lo rende disponibile per tutte le altre file operation;
2. controlla il modo di apertura del device file;
3. se è il primo device file ad essere aperto, completa l'inizializzazione del driver (si veda paragrafo 3.6.6).

Tutti punti del precedente elenco sono trattati in modo identico al device file event e sono quindi già stati discussi. Come negli altri casi, a meno che non venga specificato il flag `O_NONBLOCK`, la funzione `open` è bloccante.

A differenza degli altri device file questa funzione `open` non deve effettuare alcuna inizializzazione specifica, come invece avviene per il buffer dei comandi e quello degli eventi.

La funzione `galil1800_info_release()` implementa la file operation per la chiamata di sistema `close` sul device file info. Le operazioni svolte sono:

- invoca la funzione `galil1800_clean_controller_open_step()` (si veda paragrafo 3.6.6);
- corregge lo stato delle variabili per il controllo sul modo di apertura del device file.

7.3 Lettura della FIFO secondaria

La FIFO secondaria è un buffer hardware che, quando attivo, viene continuamente aggiornato dai controller dei dispositivi Galil DMC 1800. È quindi evidente come ogni tentativo di lettura possa solo andare a buon fine: questo ha richiesto l'implementazione della chiamata di sistema `read` solo in modo non bloccante, ignorando il valore del flag `O_NONBLOCK`.

Di seguito viene mostrata l'implementazione della file operation per la chiamata di sistema `read`:

```
static ssize_t galil1800_info_read(struct file * filp,
               char __user * buf, size_t size, loff_t *poff)
{
   struct galil1800_state *s =
      (struct galil1800_state *) filp->private_data;
   char _buf[GALIL1800_MAX_FIFO2_SIZE];
   ssize_t count, request = (size < GALIL1800_MAX_FIFO2_SIZE ?
                             size : GALIL1800_MAX_FIFO2_SIZE);
   int rc;
   count = galil1800_read_FIFO2(s, _buf, request);
   if (!count)
      return 0;
   rc = copy_to_user(buf, _buf, count);
   return (rc > 0 ? -EFAULT : count);
}
```

La quantità di byte letti dipende solo dal modello del dispositivo Galil DMC 1800 e cresce al crescere del numero massimo di assi che il dispositivo è in grado di controllare. Per permettere all'utente di allocare facilmente un buffer sufficiente a contenere l'immagine della FIFO secondaria è stata definita una macro nel file `galil1800.h`:

```
#define GALIL1800_MAX_FIFO2_SIZE  264
```

Questa macro è definita con la dimensione della FIFO secondaria presente nei modelli DMC 1880, in grado di controllare fino ad 8 assi di movimento, ovvero il massimo disponibile per questa serie di controllori. Come si può notare dall'utilizzo della variabile `request` è comunque possibile che l'utente richieda di leggere solo parte della FIFO.

La funzione `galil1800_read_FIFO2()` effettua la lettura della FIFO attraverso l'apposita porta di I/O; la sua implementazione è stata discussa in dettaglio nel paragrafo 4.4. Qualora l'utente non abbia attivato l'aggiornamento della FIFO attraverso il comando Galil `DU`, il controller restituisce una sequenza di zeri.

Infine la funzione `copy_to_user()` copia i byte letti nello spazio di memoria dell'utente e vengono restituiti il numero di byte letti oppure l'errore `EFAULT` in caso quest'ultima operazione fallisca.

Capitolo 8

Test

Al termine dello sviluppo di un driver è necessaria un'approfondita fase di test che verifichi:

- la correttezza dell'implementazione;
- il carico aggiunto al resto del sistema operativo dal driver;
- le prestazioni fornite;
- la corrispondenza fra le caratteristiche del driver e le esigenze dei programmatori che ne faranno uso; questa fase permette anche di determinare i valori più adeguati per i parametri del driver.

Per il lavoro svolto in questa tesi, al fine di realizzare in modo esaustivo tutte le fasi appena elencate, sarebbe stato necessario avere a disposizione un sistema di controllo digitale completo; ovvero, oltre ad un sistema Linux ed ad una scheda Galil DMC 1800, almeno un'amplificatore di potenza ed un meccanismo controllabile con tali schede.

Purtroppo, nei laboratori dell'Università di Tor Vergata un sistema completo non è attualmente disponibile: i test sono stati effettuati su una scheda

Galil DMC 1860[1] collegata ad un computer con 2 processori Intel Pentium III a 500 Mhz, 256 MB di RAM ed il sistema operativo Linux 2.6.15.

Ricordando che la scheda viene utilizzata principalmente attraverso comandi di uno specifico linguaggio definito dalla Galil [25], l'assenza di un meccanismo collegato ad essa non ha permesso di utilizzare alcuni comandi fra cui, ad esempio, quelli per effettuare movimenti. Non tutte le fasi di test hanno però risentito di questa mancanza.

La ricerca di bug e quindi la verifica di una corretta implementazione sono stata effettuate attraverso la scrittura di speciali programmi di test che richiedono al controller della scheda di eseguire tutti quei comandi che non hanno la necessità di interagire con il dispositivo collegato. Tali test hanno coinvolto l'intero codice, dalle funzioni di inizializzazione all'utilizzo dei buffer e possono quindi ritenersi sufficientemente accurati.

Una valutazione analoga può essere fatta per l'analisi del carico che il driver aggiunge al resto del sistema. Anche in questo caso il tipo di comandi utilizzabili è ininfluente e quindi si è potuto verificare come, anche durante un'intensa attività del driver, il resto del sistema non risenta di evidenti cali di prestazioni. In particolare molti test sono stati effettuati facendo generare alla scheda un alto numero di interruzioni[2] in modo da valutare l'impatto del gestore di interruzioni; anche in questo caso i risultati sono stati completamente positivi.

Le ultime due fasi di test, analisi delle prestazioni e rispondenza delle caratteristiche, hanno invece bisogno di un sistema completo per essere effettuate in modo esaustivo. Tutti i test possibili sono stati naturalmente svolti, ma sarebbe necessaria un'ulteriore fase che valuti i risultati durante l'esecuzione di movimenti degli assi. A tale scopo, il driver è attualmente in fase di test presso alcuni laboratori e centri di ricerca, fra cui quelli della stessa Galil; inoltre, fra non molto, potrebbe essere disponibile un sistema completo

[1]Modello della serie 1800, in grado di controllare fino ad un massimo di 6 assi di movimento. Tale scheda è stata gentilmente fornita dalla Galil Motion Control, Inc.

[2]È possibile farlo attraverso il comando `UIn`.

anche nei laboratori dell'Università di Tor Vergata. L'analisi di questi test sarà la base per uno dei principali sviluppi futuri di questo lavoro, ovvero un miglioramento delle caratteristiche del driver ed la calibrazione dei valori assegnati ai suoi parametri.

Conclusioni

Questa tesi di laurea si è incentrata sullo *sviluppo di un driver per un controllore digitale di movimento*: lo svolgimento di tale lavoro ha attraversato diverse fasi, dall'acquisizione delle conoscenze di base, alla progettazione, quindi lo sviluppo ed i test.

La prima parte del lavoro è stata dedicata alla conoscenza dell'hardware, ovvero ad uno studio delle caratteristiche dei dispositivi Galil DMC 1800 e soprattutto della loro interfaccia di comunicazione. In particolare è stato richiesto anche un approfondimento di alcuni aspetti dello standard PCI, essendo questo il bus utilizzato dalla scheda Galil.

Il passo successivo è stato la conoscenza del kernel del sistema operativo Linux. Infatti, sviluppare un driver per un sistema operativo significa conoscere gli strumenti che il suo kernel mette a disposizione, ma anche i modelli e gli standard di programmazione utilizzati. Il kernel di Linux è un sistema molto avanzato, composto da una vastissima quantità di codice: senza il supporto di testi quali [2] e [1] sarebbe stato impossibile raggiungere le conoscenze necessarie in tempi ragionevoli.

Nella fase di sviluppo del driver si sono dovuti affrontare i seguenti aspetti:

1. la comunicazione con l'hardware, quindi la scrittura del driver di basso livello;

2. la progettazione dell'interfaccia verso l'utente; questo è un aspetto molto importante, poiché definisce il modo in cui le schede Galil DMC 1800 devono essere utilizzate e quindi presuppone una conoscenza delle esigenze dei programmatori di controllori digitali di movimento;

3. l'inizializzazione del driver e dei dispositivi rilevati;

4. l'implementazione delle file operation per ognuno dei tre device file progettati;

5. lo sviluppo dei programmi per la gestione dei buffer;

6. il supporto al *sysfs* filesystem, una delle novità della versione 2.6 del kernel di Linux.

I buffer sono una delle caratteristiche più rilevanti del driver. L'introduzione del buffer dei comandi permette all'utente di utilizzare i controllori digitali senza preoccuparsi di alcuni aspetti legati alle prestazioni del sistema e quindi introduce un nuovo modo di utilizzare tali schede. Il buffer degli eventi introduce anch'esso una novità nella comunicazione tra driver ed applicazioni ed, a causa del suo potenziale impatto sulle prestazioni, ha richiesto particolare attenzione in fase di progetto.

Infine la fase di test eseguita qui all'Università di Tor Vergata si è dovuta incentrare soprattutto sulla ricerca di malfunzionamenti nel codice e solo in parte sull'analisi delle prestazioni: ciò è dovuto all'assenza nei nostri laboratori di un sistema completo di controllo digitale controllabile attraverso le schede Galil DMC 1800. Altri test sono attualmente in corso presso alcuni laboratori in Italia ed all'estero, fra cui quelli della stessa Galil: l'obiettivo è ottenere dei risultati che convalidino ulteriormente le buone prestazioni del driver, ma anche quello di comprendere meglio le esigenze degli stessi programmatori di controllori digitali di movimento.

Il risultato finale sono state circa 5000 righe di codice in linguaggio C che devono essere spesso aggiornate per seguire le evoluzioni del kernel di Linux. Lo sviluppo ha avuto inizio quando era disponibile il kernel 2.6.12, mentre la versione del driver discussa in questa tesi è stata sviluppata e testata sul kernel 2.6.15.

Sviluppi futuri

Gli sviluppi futuri di questo lavoro possono essere molteplici:

- il mantenimento del codice, seguendo gli aggiornamenti al kernel di Linux di cui, a breve, verrà rilasciata la versione 2.6.17;
- un'estensiva fase di test che completi il lavoro già iniziato e che guidi verso:
  - l'inserimento di nuove caratteristiche;
  - la calibrazione dei parametri del driver;
- una fase di raffinamento delle strutture dati;
- l'estensione del driver anche alla serie di schede Galil DMC 1802, molto simili alle 1800; in realtà una versione del driver con questo supporto è già stata sviluppata, ma non avendo a disposizione una scheda per i test si è scelto di sospenderne momentaneamente lo sviluppo.

Riferimenti bibliografici

[1] Jonathan Corbet Alessandro Rubini Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, third edition, January 2005.

[2] Daniel P. Bovet Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, third edition, November 2005.

[3] Jonathan Corbet. Driver porting: Device classes. *Linux Weekly News*, (31370), May 2003. http://lwn.net.

[4] Jonathan Corbet. Driver porting: the workqueue interface. *Linux Weekly News*, (23634), February 2003. http://lwn.net.

[5] Jonathan Corbet. kobjects and sysfs. *Linux Weekly News*, (54651), October 2003. http://lwn.net.

[6] Jonathan Corbet. The zen of kobjects. *Linux Weekly News*, (51437), October 2003. http://lwn.net.

[7] Jonathan Corbet. Safe seeks. *Linux Weekly News*, (97154), August 2004. http://lwn.net.

[8] Jonathan Corbet. Api changes in the 2.6 kernel series. *Linux Weekly News*, October 2005. http://lwn.net.

[9] Jonathan Corbet. Introducing gfp_t. *Linux Weekly News*, (155344), October 2005. http://lwn.net.

[10] Jonathan Corbet. Kernel development. *Linux Weekly News*, (163269), December 2005. http://lwn.net.

[11] Jonathan Corbet. kzalloc(). *Linux Weekly News*, (147014), August 2005. http://lwn.net.

[12] Jonathan Corbet. Nested class devices and the future of the device model. *Linux Weekly News*, (156281), October 2005. http://lwn.net.

[13] Jonathan Corbet. Nested classes. *Linux Weekly News*, (154557), October 2005. http://lwn.net.

[14] Jonathan Corbet. The new way of ioctl(). *Linux Weekly News*, (119652), January 2005. http://lwn.net.

[15] Jonathan Corbet. A pair of new timeout functions. *Linux Weekly News*, (149019), August 2005. http://lwn.net.

[16] Jonathan Corbet. Porting device drivers to the 2.6 kernel. *Linux Weekly News*, February 2005. http://lwn.net.

[17] Jonathan Corbet. Semaphores and mutexes. *Linux Weekly News*, (165039), December 2005. http://lwn.net.

[18] Jonathan Corbet. Some 2.6.12 api changes. *Linux Weekly News*, (126823), March 2005. http://lwn.net.

[19] Jonathan Corbet. Some power management changes for 2.6.15. *Linux Weekly News*, (157949), November 2005. http://lwn.net.

[20] Jonathan Corbet. Sysfs and a stable kernel abi. *Linux Weekly News*, (172986), February 2006. http://lwn.net.

[21] Free Software Foundation, Inc. Gnu general public license version 2. http://www.gnu.org/licenses/gpl.html, June 1991.

[22] Free Software Foundation, Inc. *The GNU Make Manual*, December 2002. http://www.gnu.org/software/make/manual/.

[23] Galil Motion Control, Inc. *DMC-18x0 and DMC-18x2 Series.* Technical report, 3750 Atherton Road Rocklin, California 95765.

[24] Galil Motion Control, Inc., 3750 Atherton Road Rocklin, California 95765. *DMC-1700/1800, User Manual*, 1.2k edition, July 2004.

[25] Galil Motion Control, Inc., 3750 Atherton Road Rocklin, California 95765. *Optima Series Bus-Based Command Reference*, 1.0o edition, November 2005.

[26] PCI Special Interest Group, 2575 N.E. Kathryn, Hillsboro, Oregon 97124. *PCI Local Bus Specification - Rev. 2.2*, December 1998. http://www.pcisig.com.

[27] Alessandro Rubini. Introduzione a sysfs. http://www.linux.it/~rubini/docs/sysfs/sysfs.html, 2004.

[28] www.kernel.org. *udev documentation.* http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html.

Ringraziamenti

Anche se spesso alcune vicende della vita mi creano grandi incertezze, mi ritengo ancora una persona fondamentalmente realista. Non credo molto nel destino, in eventi predeterminati o nella fortuna: credo invece che nella maggior parte dei casi ognuno di noi determini la propria strada ogni volta che compie una scelta. Di conseguenza penso che ciò che divieni crescendo è determinato soprattutto dalle persone che ti sono vicine.

È per questo che sento un profondo senso di gratitudine e di affetto verso tutti coloro che fino ad oggi, standomi vicino, mi hanno insegnato ed aiutato ad affrontare la vita. Se esiste una fortuna, la mia è stata quella di averli accanto.

Prima di tutti, i miei genitori. Non posso certo elencare tutto ciò che ho imparato da loro, ma posso affermare con certezza che mi hanno tenuto continuamente al centro della loro attenzione dal giorno stesso che sono nato, facendo sempre il massimo per il mio bene: questo è qualcosa che non potrò mai ricambiare. Di certo se ho ottenuto od otterrò risultati positivi sarà innanzi tutto grazie a loro ed è a loro che saranno dedicati.

Dalla nascita ho sempre sentito fortissimo l'affetto dei miei nonni: tutti, quelli che ci sono ancora e quelli che non ci sono più. Con loro ho passato moltissimo tempo ed oggi in alcuni aspetti del mio carattere rivedo parte del loro. Li ringrazio soprattutto per avermi raccontato le storie della loro giovinezza dai cui ho imparato che non tutto è sempre così facile come sembra. Credo che dalle difficoltà si impari molto ed io, che fino ad oggi ho avuto una vita facile grazie ai miei genitori, spero di non dimenticare mai l'esempio dei

miei nonni e di tenere sempre in mente i loro racconti.

Soprattutto negli ultimi anni un'altra guida importante è stata Roberto, il mio Maestro di Karate, con cui nel tempo ho anche costruito una bella amicizia. Senza troppe parole, ma soltanto con il suo carisma, mi ha insegnato a vedere la vita da un punto di vista diverso, più ampio. Lo ringrazio anche perché ha sempre capito la mia volontà nell'impegnarmi seriamente in quello che non è un semplice sport ed allo stesso tempo le mie difficoltà per non poter trascurare l'università, il lavoro e tutto il resto.

Voglio ringraziare particolarmente il mio relatore, il Prof. Marco Cesati, e con lui il Prof. Daniel P. Bovet, perché dimostrandomi fiducia mi hanno incoraggiato molto, ridandomi quello stimolo di cui avevo bisogno. Li ringrazio anche per avermi aiutato a capire la strada che voglio intraprendere, quella della *ricerca*. Sono sicuro che con loro, insieme a Roberto e Gianluca, riuscirò a lavorare mantenendo sempre viva la passione per la conoscenza.

Dal lontano 1995 ho iniziato a conoscere Francesco, Loredana, Barbara e... Giulia!! Ringrazio Francesco e Loredana perché a casa loro mi sento a mio agio, in famiglia, e perché come una famiglia mi sostengono e stimolano. A Barbara e Giulia dico grazie perché con il loro modo di esternare l'affetto ti ricordano ogni giorno che c'è qualcuno che ti vuole bene.

Gli amici, quelli veri, sono pochi e non tutti hanno la fortuna di averne. Io ringrazio prima di tutto Lele ed Alessio per i tanti momenti semplici, ma importanti passati insieme. E soprattutto li ringrazio perché con i fatti mi hanno dato la certezza che potrò sempre contare su di loro. Senza questa sicurezza sarei certamente un uomo più debole.

Oltre a loro sono ovviamente molto importanti tutti quegli amici con cui ho passato i più bei momenti di tutto il periodo universitario. Donato che, con i suoi modi quantomeno "particolari", sa essere un buon amico. Tutta la vecchia 29a: Ivanino, Antonio, Ivanone, Fabrizio, Aldo, Lele... Poi Marco C. con cui per un periodo era iniziata una bella amicizia. Ed ancora quelli con cui, nonostante ci si incontri di rado, si sente sempre un legame che viene da tanti anni di conoscenza, come Marco G., Emiliano, Benedetta, Virna,

Andrea e Giorgio. Infine Tiziana, Andrea, Flavia, Alessandro, Marta, Luca, Donatella, Sara, Flavia, Marco e tutti quelli che non ho nominato in queste pagine.

Un grazie anche ai miei zii e cugini per tutto l'affetto che mi hanno sempre donato.

Si potrebbe pensare che me l'ero dimenticata, ma la realtà è che in molti casi sono sempre le cose più dolci ad essere lasciate per ultime: anch'io voglio concludere la mia tesi dedicando queste ultime righe alla mia Mary. Dal 1995 ho avuto vicino una ragazza estremamente intelligente, generosa, altruista, che oltre a sostenermi ha saputo anche migliorarmi. Siamo cresciuti insieme e, comunque vada in futuro, oggi è la persona più importante della mia vita.