Driver Linux per la scheda di acquisizione NI PCI-DIO32HS versione 2

E.Giani[1], A.Checcucci [1], C.Baffa[1]

[1]Osservatorio Astrofisico di Arcetri

Arcetri Technical Report N° 3/02
Firenze 2002

Abstract

Il progetto della nuova elettronica di controllo per la strumentazione infrarossa FASTI, prevede l'utilizzo di una scheda di I/O digitale parallela per acquisizione veloce.
Il sistema FASTI è basato sul SO Linux, da cui la necessità di un driver specifico per la scheda di acquisizione scelta: la PCI DIO32HS della National Instruments.
Il presente rapporto descrive le modifiche effettuate alla precedente versione del driver ([1]).

1 Introduzione

Il progetto della nuova elettronica di controllo per la strumentazione infrarossa FASTI, prevede l'utilizzo di una scheda di I/O digitale parallela per acquisizione veloce.
Le specifiche del progetto prevedono come requisiti una velocita' minima di trasferimento di 3-4 Mbytes/s. La nostra scelta e' ricaduta sulla scheda PCI-DIO32HS della National Instruments che in modalita' sincrona dichiara velocita' di trasferimento fino a 70 Mbytes/sec.
La National Instruments non ha rilasciato, per il momento, alcun driver Linux specifico mentre il `LinuxLab` Project ne comprende uno (*comedi*) per il supporto di diverse schede di acquisizione NI, compreso quella da noi utilizzata. Purtroppo la versione non è ancora completa.
Per questo motivo abbiamo deciso di sviluppare un driver Linux che includesse esclusivamente il supporto per il funzionamento necessario al nostro lavoro.
Vogliamo sottolineare come molte delle nostre scelte operative siano state vincolate dalle limitate informazioni tecniche che abbiamo su questa scheda e come tutto ciò influisca anche sull' efficienza del driver Linux da noi sviluppato.

2 Descrizione tecnica

La scheda PCI-DIO-32HS appartiene alla famiglia dei devices DIO 6533 della National Instruments.
I devices 6533 costituiscono delle interfaccie parallele di I/O digitale per PC compatibili, PXI o CompactPCI. In particolare la scheda PCI-DIO32HS è un device DAQ per bus PCI.
Ciascun device 6533 contiene il chip DAQ-DIO dell NI, un' interfaccia a 32-bit per I/O digitale. Tale chip abilita il device ad eseguire una vasta serie di operazioni di input ed output: *single-point*, acquisizione digitale, generazione di forme d'onde digitale, trasferimenti di dati ad alta velocità.
La scheda PCI-DIO32HS include l'interfaccia PCI MITE sviluppata dalla National Instruments. Il MITE offre la possibilità di operazioni bus-master, trasferimenti burst e controllers DMA ad alta velocità .

2.1 Caratteristiche del chip DAQ-DIO

Il chip DAQ-DIO può eseguire operazioni di I/O sia *unstrobed* sia *strobed* usando 4 porte parallele di I/O ad 8 bit ciascuna, etichettate con le lettere A,B,C e D.
L'I/O *unstrobed* comprende operazioni che non coinvolgono l'uso di temporazzazioni hardware o di segnali di handshaking. Semplicemente si può scrivere e leggere direttamente su ciascuna porta. Ogni linea di I/O può essere programmata ed indirizzata attraverso il controllo dei singoli pins.
Viceversa, l'I/O *strobed* comprende trasferimenti di dati in cui l'hardware del chip DAQ-DIO regola la temporizzazione o esegue le funzioni di handshaking. In particolare, i device 6533 supportano una vasta gamma di protocolli (vedi § 2.4).
Il chip DAQ-DIO supporta due categorie distinte di I/O *strobed*: il *protocollo a due vie* ed il *pattern generation* o *protocollo ad una via*.
Nel primo tipo di *handshaking* le informazioni di controllo passano da e verso la periferica [1], con uno scambio di segnali di strobe del tipo ACK (acknoledgment) REQ (request) attraverso i quali il device e la periferica comunicano la disponibilità dei dati in entrata o uscita.
Nel secondo caso (*pattern generation*) i dati vengono letti o scritti ad una frequenza prederminata il cui valore può essere generato internamente dalla scheda oppure fornito da una sorgente esterna.
Non tutte le funzionalità fornite dalla scheda PCI-DIO32HS sono utili al nostro tipo di lavoro, per cui ci limitiamo a descrivere solo le specifiche a noi necessarie.
Trascureremo perciò tutta la parte di I/O digitale che non coinvolge l'uso di segnali di handshaking (cioè l'I/O *unstrobed* ed il *pattern generation*).

[1] con questo termine si intende rifersi ad un device esterno che il DAQ-DIO controlla, monitora o con cui comunica

2.2 Protocollo a due vie

Per le operazioni di handshaking, il DAQ-DIO include le seguenti caratteristiche:

- due canali di *handshaking*, a cui ci si riferisce con il termine di **gruppo** con logica di temporizzazione e registri di controllo separati
- un insieme di linee di controllo di *handshaking* ed altre opzioni separate per ciascun gruppo
- controllo software del protocollo di handshaking e di altre opzioni per ogni gruppo
- 4 FIFO interne, ciascuna associata ad una porta
- *funneling* che consente al software di modificare l'instradamento (routing) delle FIFO verso le porte
- contatori per controllare il numero di trasferimenti eseguiti.

2.3 FIFO interne e gruppi di acquisizione

Il DAQ-DIO fornisce una FIFO interna per ogni porta. Ogni FIFO è costituita da un buffer di memoria bidirezionale, del tipo *first-in first-out*, ad 8 bit con una profontità di 16 parole (words). In totale vi sono 4 FIFO ciascuna associata ad una porta di I/O.
Nell'I/O *strobed* le porte e le relative FIFO devono essere allocate ad uno dei gruppi di handshaking.
Un gruppo consiste di un controller per le temporizzazioni e di un set di 4 linee di controllo associate: REQ, ACK (STARTRIG), STOPTRIG e PCLK. Come già accennato al paragrafo precedente, l'hardware della scheda PCI-DIO32HS supporta fino a 2 gruppi indipendenti, da cui segue che il chip DAQ-DIO può eseguire fino a due operazioni simultanee di handshaking.

2.3.1 Accesso alle FIFO

In modalità *strobed* I/O, devono essere usate le FIFOs, piuttosto che i registri delle porte di I/O, per leggere o scrivere i dati. In particolare si può leggere solo da un gruppo di input e scrivere solo su un gruppo di output. Le operazioni di lettura/scrittura possono essere ad 8-bit, 16-bit o 32-bit.

2.4 I protocolli

Quando vengono eseguite operazioni completamente gestite dai segnali di handshaking (*protocollo a due vie*), possono essere selezionati diversi protocolli offerti dai devices della famiglia 6533.
Il protocollo scelto determina la temporizzazione dei segnali di ACK che il device 6533 manda alla periferica e dei segnali di REQ attesi dalla periferica stessa. Uno dei protocolli, il *burst mode* differisce dagli altri in quanto è l'unico protocollo sincrono.
I protocolli che la scheda può generare sono i seguenti:

- protocollo di emulazione 8255. A causa dei tempi di risposta più veloci ed al buffering offerto dalle FIFO, questo modo di emulazione 8255 offre velocità di trasferimento superiori di quelle fornite dal chip 8255.
- level ACK. Dopo ciascun trasferimento, il device afferma il segnale ACK verso la periferca. Il mantenimento della linea di ACK fa si' che il device non inizi un nuovo trasferiment fino a che non avviene una transizione *false-true* sulla linea di REQ.
- leading edge pulse. Dopo ciascun trasferimento, il device 6533 manda alla periferica un impulso sulla linea di ACK. Il device 6533 prima di iniziare un nuovo trasferimento, aspetta una transizione *false-true* sulla linea di REQ e l'inizio di un impulso REQ. Il software consente di specificare un ritardo dell'impulso ACK
- long pulse. In questa modalità il funzionamento è analogo al precedente, con la sola differenza che si può specificare un'ampiezza minima dell'impulso.

- trailing-edge pulse. Dopo ciascun trasferimento il device 6533 manda alla periferica un impulso sulla linea di ACK. Prima di un nuovo trasferimento, il device aspetta una transizione *true-false* sulla linea di REQ e la fine dell'impulso REQ.
- burst mode. Il device 6533 manda o riceve un segnale di clock sulla linea PCLK. Durante ogni ciclo il device 6533 afferma un segnale ACK se è pronto per il trasferimento e la periferica, in modo analogo, afferma un segnale di REQ se è pronta per il trasferimento.

3 Caratteristiche tecniche per la programmazione della scheda

Per programmare un device DIO 6533 devono essere scritti e letti una serie di registri implementati sul device stesso. Questi registri, rientrano in due distinte categorie:

- i registri di interfaccia al bus
- i registri di I/O implementati dal chip DAQ-DIO

3.1 Inizializzazione del bus

Il device PCI-DIO32HS usa il chip MITE come interfaccia al bus PCI. Questo circuito integrato è stato progettato appositamente dalla National Instuments per l'acquisizione dati e deve essere propriamente configurato. Quando non viene usato il software specifico NI-DAQ della NI, come nel nostro caso, la configurazione del chip MITE deve essere eseguita nel seguente modo:

1. con le chiamate delle funzioni PCI deve essere individuata la presenza della scheda facendo ricorso ai numeri identificativi della casa produttrice e della scheda [2]
2. mappatura della memoria di I/O del device. In generale la memoria di I/O di devices connessi al bus PCI è mappata ad indirizzi fisici alti, molto al di là del temine degli indirizzi fisici della RAM. I drivers del kernel devono tradurre l'indirizzo fisico della memoria di I/O del device PCI in un indirizzo lineare nello spazio del kernel [3]. Gli indirizzi base dei chip DAQ-DIO e MITE [4] devono essere rimappati nello spazio della memoria virtuale del kernel. Questa operazione viene eseguita utilizzando la funzione *ioremap*.
3. scrittura del valore 0x80 all'offset 0xc0 a partire dall'indirizzo rimappato del MITE.

Eseguita la fase di inizializzazione del bus, l'accesso in lettura/scrittura dei registri mappati in memoria viene effettuata ricorrendo alle seguenti funzioni:

- readb/writeb
- readw/writew
- readl/writel.

Queste macro sono usate per leggere o scrivere nella memoria di I/O dati a 8-bit, 16-bit e 32-bit.
L'uso di tali macro è preferibile all'accesso diretto dell'indirizzo di memoria dei registri, in quanto tengono conto delle diverse architetture. L'indirizzo di ciascun registro è ottenuto sommando all'indirizzo rimappato del chip DAQ-DIO o del MITE [5] il relativo offset. In Appendice A possiamo trovare l'elenco completo dei registri del

[2] Nel caso specifico il codice numerico della NI è 0x1093, quello della scheda PCI-DIO32HS 0x1150

[3] Si deve tenere in considerazione che gli indirizzi lineari del kernel partono dall'indirizzo 0xc0000000. Durante la fase di inizializzazione, il kernel mappa gli indirizzi fisici della RAM disponibile nella regione iniziale del quarto GB dello spazio degli indirizzi lineari

[4] L'indirizzo base_0 corrisponde all'indirizzo base del MITE, mentre l'indirizzo base_1 all'indirizzo base del chip DAQ-DIO. La dimensione di ciascun spazio di memoria è di 4 kB.

[5] La programmazione dei registri del MITE è necessaria solo nel caso in cui si voglia ricorrere alle funzionalità fornite da questo chip, come ad es. i trasferimenti DMA.

chip DAQ-DIO ed i loro offset.
Per maggiori informazioni rimandiamo al manuale *DAQ-DIO Technical Reference Manual* che può essere tovato nel sito `web` della National Instruments:

http://www.ni.com

Non è invece disponibile alcuna informazione ufficiale relativa ai registri del MITE.

3.2 Interrupts

Il DAQ-DIO presenta le seguenti caratteristiche per gli interrupts:

- una linea di interrupt per l'interfaccia diretta con il bus controllabile tramite software
- un bit principale per l'abilitazione globale della linea di interrupt
- un bit di abilitazione per ogni possibile sorgente di interrupt.

La scheda genera un interrupt solo quando entrambi i bit sono configurati.
Le cause che possono generare un interrupt, sono le seguenti:

- trasferimento pronto
- numero dei trasferimenti terminato
- la logica di I/O deve fermarsi ed aspettare per la CPU o il controller DMA.
- si è verificato un DMA terminal counter sul primo gruppo del DMA
- si è verificato un DMA terminal counter sul secondo gruppo del DMA

Entrambi i gruppi di linee di handshaking possono essere programmati per generare i segnali di interrupt nel caso si verifichi uno (o piú) degli eventi elencati sopra.

3.3 Funzionamento del DMA nella scheda PCI DIO32-HS

Per eseguire trasferimenti DMA, il chip DAQ-DIO deve lavorare con un controller DMA. La scheda PCI-DIO32HS possiede il DMA controller sulla scheda stessa, implementato nel chipset PCI MITE ASIC.
È importante sottolineare che per il funzionamento dei trasferimenti DMA, devono essere programmati sia il chip DAQ-DIO, sia il controller DMA.
Il chip DAQ-DIO include le seguenti caratteristiche per il DMA:

- supporta due canali DMA. Questi possono essere allocati entrambi ad un gruppo, oppure uno a ciascun gruppo.
- l'ampiezza del trasferimento DMA (che può essere a 8-bit, 16-bit o 32-bit) è controllata tramite software.

3.3.1 Caratteristiche del controller MITE DMA

Il MITE contiene i canali DMA che possono essere usati per trasferire i dati tra le porte di I/O e la memoria di sistema. Il MITE fornisce due canali separati per i due gruppi di I/O [6]. Ciascun canale supporta diversi modi operativi.
Uno di questi è il *link-chaining* [7] che rende possibile il trasferimento di blocchi di dati in aree di memoria non contigue.
Questo tipo di implementazione *bus-master* è una combinazione di hardware e software dove il controller è

[6] Deve essere usato il controller DMA 1, la linea DRQ 1 e la linea DACK 1 per il gruppo 1; il controller DMA 2, la linea DRQ 2 e la linea DACQ 2 per il gruppo 2.

[7] Questa modalità sarà l'unica presa in considerazione, perchè è la sola per la quale siamo in grado di trovare esempi della programmazione del chipset MITE

progettato in modo da riprogrammare se stesso senza l'uso di interrupt e l'intervento da parte della CPU.
In modalità *link-chaining* il controller DMA non viene riprogrammato per le nuove locazioni di memoria ma si autoconfigura leggendo una serie di record di dati concatenati tra loro, immagazzinati in un buffer di memoria. Ciascun record contiene l'indirizzo di destinazione dei dati, il numero di byte da trasferire e l'indirizzo del successivo record di dati. Il controller DMA legge ed esegue ciascun nodo contenuto nella lista. Il microprocessore deve solamente programmare il controller con l' indirizzo del primo nodo della catena, il resto del lavoro viene svolto direttamente dal controller DMA.
La modalità *link-chaining* richiede perciò la costruzione di una catena di strutture in cui sono immagazzinate le informazioni relative a ciascun buffer di memoria usato nel trasferimento. Per ogni nodo della catena devono essere specificati i valori dei seguenti registri del controller DMA:

- Transfer Count Register (TCR): contiene il numero di byte da trasferire
- Memory Address Register (MAR): contiene l'indirizzo fisico del buffer di memoria
- Device Address Register (DAR): non viene usato dal chip DAQ-DIO
- Link Address Register (LKAR): contiene l'indirizzo fisico del nodo successivo della catena di strutture.

Prima di iniziare il trasferimento DMA, deve essere caricato il registro LKAR con l'indirizzo fisico del primo nodo perchè, come accennato in precedenza, il MITE necessita un punto iniziale di accesso alla catena.
Dopo che il DMA è stato attivato per il trasferimento, il MITE scorre l'intera catena di strutture e carica gli indirizzi fisici dei buffer di memoria all'interno del MAR. In seguito il MITE trasferisce i dati dalla scheda al buffer (input) oppure dal buffer alla scheda (output).
Nel caso di un trasferimento finito di dati, il processo continua finchè il MITE non raggiunge un nodo vuoto, cioè un nodo i cui valori sono posti tutti a zero. Per un trasferimento continuo, invece, l'ultimo nodo della catena punta nuovamente al primo.

4 Configurazione della scheda per operazioni di acquisizione

In questo capitolo intendiamo fornire le informazioni utili per configurare ed eseguire le operazioni di I/O in modalità handshaking. Indipendentemente dal metodo scelto per acquisire i dati (polling, interrupt o trasferimenti DMA) le seguenti operazioni dovranno essere comunque eseguite nell'ordine indicato:

- configurazione del gruppo (o dei gruppi) di I/O
- programmazione del protocollo di handshaking
- reset delle flags
- esecuzione della modalità handshaking
- reset delle FIFO

Rimandiamo alle App. A e B per l'elenco dei registri della scheda PCI-DIO32HS e la locazione dei bit.

4.0.2 Configurazione del gruppo di I/O

Per configurare un gruppo di acquisizione, devono essere eseguiti i passi successivi:

1- assegnare le FIFO e le porte a ciascun gruppo

2- fissare la direzione di acquisizione del gruppo (input o output)

3- configurare ogni porta assegnata al gruppo

4- selezionare l'ampiezza dell'operazione di lettura/ scrittura/ trasferimento DMA (8-bit, 16-bit oppure 32-bit)

1) Per assegnare una o più FIFO ad un gruppo di I/O devono essere settati i bit relativi nel registro `Data_Path`. **Non è possibile assegnare la stessa FIFO ad entrambi i gruppi**.
2) Lo stesso registro consente di configurare il *funneling*[8] e la direzione di acquisizione. I valori da attribuire ai bit sono i seguenti:

funneling = **0** (no funneling) o **2** (8-bit funneling)

GroupDirection = **0** (input) o **1** (output)

3) La configurazione delle porte consiste nel porre la direzione di acquisizione uguale a quella del gruppo a cui appartengono. Ad ogni porta è associato un registro `Port_Pin_Directions`. I singoli bit del registro vanno posti a 0 oppure ad 1 a seconda che il gruppo a cui appartiene la porta sia un gruppo di input o di output.
4) L'ampiezza delle operazioni di lettura, scrittura o trasferimento DMA (che di seguito indicheremo con il termine `TransferWidth`) viene determinata dai primi due bit del registro `Transfer_Size_Control`.
`TransferWidth` può assumere i seguenti valori:

- 0,1 : per operazioni effettuate a 32-bit
- 2 : per operazioni a 8-bit
- 3: per operazioni a 16-bit

Il valore configurato per `TransferWidth` deve essere uguale o minore alla dimensione in bit del gruppo di acquisizione. Ad es. ad un gruppo composto da tutte e quattro le FIFO (32-bit) possono essere associate operazioni di acquisizione a 16-bit o 32-bit.

4.0.3 Programmazione del protocollo di handshaking

La configurazione del protocollo di handshaking comporta la programmazione dei quindici registri
`Protocol_Register_1`, ... ,`Protocol_Register_15`
facendo riferimento ai valori riportati in App.C e alla direzione di acquisizione del gruppo.

4.0.4 Reset delle flags

Prima dell'avvio di ogni operazione di I/O è indispensabile eseguire il reset delle flags associate al gruppo. Queste sono:

- Clear Waited
- Clear PrimaryTC
- Clear SecondaryTC
- DMAReset
- FIFOReset
- Clear Expired

Ogni flag è associata ad un bit del registro `GROUP_FIRST_CLEAR` o `GROUP_SECOND_CLEAR`: per resettare una flag deve essere posto ad 1 il corrispondente bit del registro a cui appartiene.

[8] Il funneling consente di configurare la dimensione dell' I/O più piccola della dimensione del gruppo, cioè di usare un numero minore di porte rispetto al numero di FIFO. Ad es. il funneling del tipo 8:16 consente di eseguire operazioni di I/O ad 8-bit per un gruppo di 16-bit, assumendo che vengano trasferiti un numero pari di dati.

4.0.5 Esecuzione delle operazioni di I/O in modalità handshaking

L'avvio del processo di handshaking necessita della:

1- definizione di acquisizione definita (*numbered*) o continua

2- abilitazione dei segnali di handshaking

1) Il quarto bit (`Numbered`) del registro `Protocol_Register_1` consente di specificare se si tratta di un' acquisizione definita, cioè con una quantità determinata di dati da trasferire, oppure no. Nel primo caso il bit `Numbered` deve essere posto uguale ad 1, nell'altro caso a 0.
Per operazioni di I/O finite deve essere programmato anche il numero di conteggi. Tale valore deve essere scritto nel registro `Transfer_Count` relativo al gruppo di I/O.
2) I primi tre bit (`RunMode`) del registro `Protocol_Register_1` consentono invece di fermare o attivare la generazione dei segnali di handshaking. In particolare:

RunMode=0x00 : la generazione dei segnali di handshaking viene fermata. È importante sottolineare che la flag `RunMode` deve essere posta uguale a zero **prima di eseguire la configurazione di un gruppo di I/O**.

RunMode=0x03 : la generazione dei segnali di handshaking viene abilitata.

Una volta programmata la scheda fissando i gruppi di I/O, il protocollo, etc. come descritto nei paragrafi precedenti, le operazioni di handshaking iniziano quando la flag `RunMode` viene posta uguale a 0x03. Questa deve essere l'ultima operazione eseguita prima che abbia inizio il trasferimento dei dati da o verso la periferica.

4.0.6 Reset delle FIFO

Il reset delle FIFO risulta necessario quando vengono eseguiti trasferimenti continui in lettura.
In modalità di acquisizione definita è noto a priori il numero di dati da leggere, per cui non dovrebbero rimanere dati nelle FIFO alla fine del trasferimento. In modalità continua, invece, questa eventualità si può presentare in seguito all' improvvisa interruzione dell'acquisizione.
La presenza di dati in una FIFO è controllata dal bit `DataLeft` del registro `GROUP_STATUS`. Tale flag è uguale ad 1 finche' una (o più) FIFO contiene dati.
Per svuotare una FIFO è necessario leggere il registro `GROUP_FIFO` fino a quando il bit `DataLeft` non risulta uguale a 0.

4.1 Programmazione aggiuntiva per operazioni di I/O con IRQ

Il DAQ-DIO può accrescere l'efficienza del bus usando il canale di interrupt. Si può usare un canale di interrupt per la notifica di un evento senza dover ricorrere alla tecniche di polling.
Entrambi i gruppi di I/O possono essere programmati per generare un segnale di interrupt quando le condizioni elencate al § 3.2 sono soddisfatte.
Per abilitare gli interrupts devono essere eseguiti i seguenti passi:

1- configurare i gruppi e le porte prima di abilitare qualunque interrupt

2- selezionare le sorgenti di interrupt

3- abilitare la linea di interrupt

Per il passo **1)** rimandiamo al §4.0.2.
2) La selezione delle sorgenti di interrupt viene effettuata configurando, nel registro `Interrupt_Control` del gruppo di I/O, i bit corrispondenti.
3) L'abilitazione globale degli interrupt è realizzata dai primi due bit (`InteruptLine`) del registro `Master_DMA_and_Interrupt_Control`. Questi bit determinao l'uso della linea di interrupt. In particolare:

- 0 : pone la linea di interrupt in tristate (stato al momento dell'accensione)
- 1 : guida la linea dell'interrupt nello stato basso (disabilita gli interrupts)
- 3 : abilita gli interrupt per le normali operazioni

4.2 Programmazione aggiuntiva per operazioni di I/O con DMA

Descriviamo ora la programmazione con accesso diretto alla memoria (DMA) del DAQ-DIO e del contoller DMA[9].
Il DMA abilita il DAQ-DIO a muovere i dati direttamente da e verso la memoria senza l'uso della CPU. Il DMA può ridurre drasticamente il carico della CPU crescendo contemporaneamente la velocità di trasferimento dei dati.
Nel caso in cui si voglia implementare questa caratteristica all'interno del software di gestione del device, è necessario eseguire una serie ulteriore di operazioni rispetto a quelle descritte nei §4.0.2-§4.0.6.
Per configurare ed eseguire i trasferimenti DMA, devono essere compiute le seguenti operazioni:

1- configurazione del gruppo e delle porte associate al gruppo

2- programmazione del protocollo

3- configurazione del canale DMA usato nel trasferimento

4- scrittura di una routine che gestisca la fine del singolo trasferimento.

5- programmazione del controller DMA

6- inizio del trasferimento DMA, in modalità continua o definita (*numbered*)

I passi **1)** e **2)** sono già stati descritti al §4.0.2.
3) La selezione del canale usato da un gruppo di I/O viene eseguita configurando i primi quattro bit (`DMAChannel`) del registro `DMA_Line_Control`.
In particolare:

- <0..1> canale DMA primario 1,2 o 0 per nessuno
- <2..3> canale DMA secondariorio 1,2 o 0 per nessuno

Per usare un solo canale deve essere programmato lo stesso valore per entrambi i canali DMA, altrimenti il gruppo di I/O utilizza in modo alterno i due canali.
Oltre alla configurazione dei bit `ReadyLevel` del registro `FIFO_Control` e `TransferWidth` del registro `Transfer_Size_Control` (vedi §4.0.2) che sono necessari per le normali operazioni di handshaking, devono essere configurati anche i bit `TransferLength` e `RequireRLevel` nel registro `Transfer_Size_Control`.
I bit `TransferLength` determinano la massima durata del segnale *DRQ*. Il DAQ-DIO fornisce un meccanismo di time-out che è richiesto da alcuni controller DMA per prevenire il monopolio del bus da parte del processo DMA. I valori selezionabili sono:

- **0**: Nessun limite. Questo valore è raccomandato per il MITE (PCI) DMA
- **1**: massimo di 4 trasferimenti
- **2**: massimo di 8 trasferimenti
- **3**: massimo di 16 trasferimenti

[9] Per la scheda PCI-DIO32HS deve essere programmato il chip MITE ASIC che implementa il controller DMA

Il bit `RequireLevel` determina quanto a lungo il segnale *DMAReq* rimane affermato. Questo bit deve essere attivato per trasferimenti DMA in output, per trasferimenti non di input e trasferimenti non DMA.
4)Quando il controller MITE DMA è programmato per generare un interrupt al termine di un trasferimento, la routine di gestione dell'interrupt deve leggere il valore del registro di stato CHSR.
Di questo registro a 32 bit conosciamo il significato di alcuni bit:

- bit 31 (INT): si è verificato un interrupt
- bit 25 (DONE): il trasferimento completo è terminato
- bit 19 (LINKC): il singolo trasferimento (link) è terminato
- bit 15 (ERROR): si è verificato un errore.

5) Per quanto riguarda questa parte, la documentazione è completamente assente, per cui non possono essere forniti dettagli. Possiamo solo rimandare al sorgente scritto da Tomaz Motylewski che ha trasportato in ambiente Linux un file originariamente scritto dalla National Instruments Corporation per il funzionamento del MITE DMA in modalità *link-chaining*.
Ulteriori informazioni si possono ricavare anche dai sorgenti del driver *comedi*.
La programmazione del controller DMA in modalità *link chaining* consiste nella programmazione dei seguenti registri:

- `DMA_CHOR`: configura le operazioni eseguite dal controller
- `DMA_CHCR` configura il controllo del canale
- `DMA_MCR`, `DMA_DCR`: configurano i trasferimenti da e verso la memoria e da e verso il device
- `DMA_LKCR`: configura il link
- `DMA_LKAR`: contiene l'indirizzo fisico del primo nodo della catena (vedi §3.3.1).

6) Il trasferimento DMA viene avviato dopo che è stato abilitato il bit `DMA_CHOR_START` nel registro `DMA_CHOR` del MITE, configurato il trasferimento come continuo o finito ed abilitati i segnali di handshaking (vedi §4.0.5). Come già detto, per trasferimenti finiti deve essere programmato anche il valore del registro `Transfer_Count`.

5 Descrizione del driver Linux per la scheda PCI-DIO32HS

In questo capitolo non intendiamo fornire una descrizione dettagliata su come scrivere un device driver per Linux, in quanto esistono pubblicazioni e libri che svolgono egregiamente questo compito ([2]).
Diciamo solamente che la scrittura di un driver Linux comporta in generale la implementazione delle seguenti operazioni:

1- registrazione del driver e delle sue funzionalità
2- inizializzazione hardware del device fisico
3- definizione delle operazioni eseguibili sul device
4- rimozione del driver

5.1 Registrazione del driver pcidio32

La registrazione del driver viene effettuata dalla funzione *init_module()* che viene chiamata quando il modulo viene inserito attraverso i comandi *insmod* o *modprobe*.
La prima funzione chiamata (*register_chrdev*) ci permette di assegnare al driver, identificato dal nome `pcidio32`, il numero maggiore e di registrare in corrispondenza di esso una serie di operazioni dette *metodi*. Questi vengono invocati dal kernel tutte le volte che un'applicazione utente esegue un'operazione su un device file con numero maggiore uguale a quello associato al driver.
I metodi del driver sono definiti nella struttura `dio32_fops` il cui indirizzo viene passato come argomento alla funzione *register_chrdev*.
Il numero maggiore del device file associato al driver, non viene assegnato dinamicamente ma viene prefissato al valore 60. Questo valore appartiene ad uno degli intervalli di numeri maggiori allocati per uso sperimentale.
Dopo la registrazione del driver, la funzione *init_module()* invoca la funzione *find_NI_cards()* che rileva il numero di schede del tipo PCI-DIO32HS installate nel PC.
Questa funzione inizializza per ogni scheda individuata una struttura del tipo `mite_struct` (vedi § 6.5) e restituisce il numero delle schede individuate.
Come ultimo passo, la routine *init_module()* alloca la memoria necessaria per contenere le strutture dati associate ad ogni device fisico (struttura del tipo `ni_device` vedi § 6.1). Se si verifica una condizione di errore durante l'esecuzione della funzione, il driver viene rimosso e viene rilasciata l'eventuale memoria allocata.

5.2 Inizializzazione hardware del device

L'inizializzazione hardware del device fisico viene generalmente eseguita all' interno dalla funzione *init_module()*.
Nel caso specifico abbiamo deciso di procedere in modo diverso, facendo eseguire la funzione di inizializzazione da un apposito comando *ioctl* chiamato dal processo utente (vedi §5.3.4). Durante questa operazione, viene fissato anche il numero di gruppi di I/O con cui il driver lavorerà. Abbiamo detto in precedenza che questo tipo di scheda supporta due gruppi di I/O indipendenti, che possono lavorare simultaneamente: noi lavoreremo sempre con un solo gruppo.
Dal punto di vista del driver, ciacun gruppo di I/O è considerato come un *subdevice* separato, descritto da una struttura dati del tipo `ni_subdevice` (vedi §6.2).

5.3 Metodi associati al driver

Il driver sviluppato per la scheda PCI-DIO32HS della National Instruments non supporta tutte le funzionalità offerte da questo device, ma solo quelle che abbiamo ritenuto indispensabili al funzionamento del nostro sistema di acquisizione. In particolare noi lavoreremo esclusivamente con:

- **acquisizioni definite** di **solo lettura** e **controllate da segnali di handshaking**
- useremo **un solo gruppo di I/O**: il gruppo 1.
- le linee PCLK,ACK,REQ e STOPTRIG relative alla logica del gruppo 2, saranno programmate come linee extra di I/O per comunicare con l'elettronica esterna.

L'elenco dei metodi associati al driver include le seguenti funzioni:

- `dio32_open()` e `dio32_close()` per l'apertura e la chiusura del device file
- `dio32_read()` per la lettura dei dati di acquisizione
- `dio32_ioctl()` per la configurazione del device
- `dio32_mmap()` per la mappatura del buffer di acquisizione DMA nello spazio degli indirizzi dell'applicazione utente.

5.3.1 dio32_open()

Il metodo *dio32_open()* viene eseguito quando l'applicazione utente apre il device file per le operazioni di I/O.
La funzione *dio32_open()* ricava dalla struttura *inode* informazioni sul numero minore del device file attraverso il quale, chiamando la funzione *get_device_by_minor()*, si ricava la struttura dati associata al device fisico in uso.
La funzione *dio32_open()* si limita successivamente ad incrementare il contatore del driver con una chiamata a `MODULE_INC_USE_COUNT` ed il contatore privato `use_count` (vedi § 6.1).

5.3.2 dio32_close()

Quando il programma utente rilascia il *file handle* che ha usato per interagire con il device, viene chiamata la funzione *dio32_close()*.
Dal numero minore del device file viene ricavata la struttura associata al device e per ogni subdevice viene eseguito il reset dei registri, viene ristabilita la configurazione iniziale delle linee extra di I/O della seconda logica,viene rilasciata la memoria virtuale allocata ed inoltre viene azzerata la struttura associata al subdevice.
Infine viene decrementato sia il contatore del driver (`MODULE_DEC_USE_COUNT`) sia quello privato.

5.3.3 dio32_read()

La funzione *dio32_read()* viene chiamata tutte le volte che il programma utente esegue una operazione di lettura sul device file.
Questa funzione implementa, per il momento, l'acquisizione dei dati in *polling mode* senza l'utilizzo di interrupt o del DMA.
Quando i dati sono presenti nelle FIFO associate al gruppo di I/O su cui viene eseguita l'operazione di lettura, il bit `TransferReady` del registro `GROUP_FLAGS` relativo al gruppo di I/O ha valore 1. Controllando periodicamente lo stato di tale bit, è possile determinare quando i dati sono disponibili.
Per l'operazione di lettura viene usato un buffer interno di dimensione fissa (4096 byte) che, quando completo, viene copiato nello spazio di indirizzamento del processo utente.
La funzione implementa inoltre un timeout in modo che il driver non congeli il sistema in caso di errato funzionamento.

5.3.4 dio32_ioctl()

Una funzionalità complementare alla necessità di leggere e scrivere il device è il controllo dell'hardware ed il modo più comune di eseguire questa operazione attraverso il device driver, consiste nell'implementare il metodo *ioctl*. La chiamata di sistema *ioctl* offre un metodo per eseguire comandi specifici del device e consente alla applicazione utente di accedere a caratteristiche uniche dell' hardware che deve essere pilotato dal device driver.
Gli argomenti della chiamata di sistema sono passati al metodo `ioctl` del driver in accordo alla dichiarazione del metodo stesso:

int (*ioctl)(struct inode *inode,struct file *filp,
unsigned int cmd,unsinged long arg);

I puntatori `inode` e `flip` sono valori che corrispondono al descrittore di file `fd` passato dall'applicazione.
L'argomento `cmd` è passato invariato e l'argomento opzionale `arg` è passato sotto forma `unsigned long` indipendentemente dal fatto che sia stato fornito dalla chiamata di sistema come intero o puntatore. Se il programma utente non passa il terzo argomento, il valore `arg` ricevuto dal driver è privo di significato.
La maggior parte delle implementazioni *ioctl* consiste in un comando `switch` che seleziona il corretto comportamento secondo l'argomento `cmd`. I differenti comandi sono identificati da valori numerici che devono essere unici. Per semplicità, ad ogni comando numerico viene associato un nome simbolico, assegnato attraverso una direttiva di preprocessore.
Di seguito forniamo l'elenco dei comandi e la loro descrizione.

- DIO32_IOC_BOARD_CONFIG: questo comando esegue la funzione *do_board_config()* che:

- inizializza ad 1 il numero dei subdevice (`n_subdevices`) in uso perchè come specificato al §5.3 noi lavoreremo con un solo gruppo di I/O;
- chiama la funzione *nidio_device_attach()*. Questa funzione effettua l' inizializzazione hardware del device, eseguendo le seguenti operazioni:
  - chiama la funzione *nidio_find_device()*. Questa funzione ricava dal numero di scheda la struttura del tipo `mite_struct` ad essa associata che, come descritto al §6.5, contiene le informazioni sulle risorse hardware del device ;
  - chiama la funzione *setup_mite()* che esegue l'inzializzazione dell'hardware come descritto al paragrafo §3.1;
  - salva nei campi relativi della struttura `ni_device` gli indirizzi rimappati del chip DAQ-DIO e MITE;
  - installa il gestore di interrupt in corrispondenza dell'IRQ del device;
  - alloca la memoria per le strutture del tipo `ni_subdevice`;
  - installa le routine *ni_dio_mode0, ni_dio_mode1,..,ni_dio_mode4* per la programmazione dei protocolli;
  - configura la flag `attached` ad 1, indicando la corretta inizializzazione della scheda;

- DIO32_IOC_CONFIG_GROUP: la chiamata di sistema *ioctl* relativa a questo comando deve specificare anche un parametro aggiuntivo che rappresenta il puntatore ad una struttura del tipo `dio32_config`. Questa (vedi §7.5) raccoglie tutte le informazioni per la configurazione del gruppo di I/O.
  Dal punto di vista del driver, questo comando `ioctl` attiva la funzione *do_config_ioctl()* che svolge i seguenti compiti:
  - copia la struttura `dio32_config` passata dall'utente, nello spazio di indirizzamento del kernel. I dati in essa contenuta configurano i parametri del gruppo di I/O;
  - chiama la funzione *check_user_value()* che esegue un controllo dei parametri passati dall'utente, in modo da stabilirne la validità[10];
  - ricava il numero del subdevice a cui si riferiscono i dati e seleziona la relativa struttura del tipo `ni_subdevice`;
  - chiama la funzione *handshaking_config()* che esegue le operazioni descritte al §4.0.2.
  - chiama la funzione *extra_lines_config()* che configura le
  - pone la variabile `configured` (vedi §6.2) ad 1 per indicare che il gruppo è stato configurato correttamente.

- DIO32_IOC_RESET_GROUP: questo comando esegue un reset della configurazione dei gruppi.
  Questa operazione risulta indispensabile quando il programma utente deve modificare l'assegnazione delle porte (e quindi della FIFO) ai gruppi. Dalla parte del driver, il comando `ioctl` esegue la funzione *do_group_reset()* che effettua le seguenti operazioni:
  - seleziona la struttura `ni_subdevice` relativa al gruppo da resettare;
  - interrompe i segnali di handshaking chiamando la funzione *stop_handshaking()*;
  - chiama la funzione *ni_daq_reset()* che esegue il reset dei registri del gruppo: ad es. i registri delle FIFO assegnate,i registri del protocollo, del DMA etc.;
  - dealloca l'eventuale buffer di memoria virtuale chiamando la funzione *mbuff_dealloc()*;
  - azzera la struttura `ni_subdevice` relativa al gruppo salvando solo il numero del gruppo.

[10] Questo passaggio potrebbe essere evitato se il controllo sui parametri da passare al driver venisse eseguito dalla applicazione utente. Riteniamo opportuno che tale verifica venga comunque eseguita.

- DIO32_IOC_MITE_CONFIG_GROUP: questo comando consente al processo utente di configurare i registri del controller MITE DMA relativi al gruppo di I/O selezionato.
  A livello del kernel il comando seleziona la routine *do_mite_config()* che:
  - chiama la funzione *DMA_Program_Minimite()* che esegue la programmazione del controller DMA per il gruppo specificato;
  - chiama la funzione *DMA_Arm_Minimite()* che inizia i trasferimenti DMA tutte le volte che vi sono dati nella FIFO del gruppo;
  - setta il bit SF_MITE della flag `iobits` del gruppo (vedi §7.1), per segnalare che il controller DMA è stato programmato.
- DIO32_IOC_MITE_RELEASE_GROUP: questo comando consente al processo utente di fermare i trasferimenti DMA per il gruppo di I/O. Il comando `ioctl` attiva la funzione *do_mite_release()* che:
  - chiama la funzione *DMA_Disarm_Minimite()* per fermare i trasferimenti DMA;
  - resetta il bit SF_MITE della flag `iobits` del gruppo per segnalare che il controller DMA non è più programmato.
- DIO32_IOC_START_GROUP: il comando avvia la procedura di acquisizione per il corrispondente gruppo di I/O mandando in esecuzione la funzione *start_acquisition()*.
  La funzione *start_acquisition()* esegue i seguenti passi:
  - se la flag `USE_DMA` è configurata (vedi § 7.4), verifica che il controller DMA sia stato programmato (bit SF_MITE abilitato) e che il buffer virtuale sia stato allocato e mappato nello spazio di indirizzamento dell'applicazione utente (bit SF_MMAP abilitato);
  - chiama la routine *run_handshaking()* che esegue le operazioni descritte al §4.0.5.
- DIO32_IOC_STOP_GROUP: il comando ferma la procedura di acquisizione per il gruppo di I/O. La funzione corrispondente a questo comando `ioctl` è *stop_acquisition()*, che compie le seguenti operazioni:
  - chiama la routine *stop_handshaking()* per fermare i segnali di handshaking;
  - chiama la funzione *empty_fifo()* che svuota le FIFO (vedi § 4.0.6)[11].
  - chiama *clear_flags()* che esegue il reset delle flag (vedi §4.0.4).
- DIO32_IOC_WAIT_INTERRUPT_GROUP: gli argomenti di questa richiesta sono il numero del subdevice, il canale di interrupt ed il tempo di timeout specificato in millisecondi.
  Questo comando esegue a livello del driver la funzione *wait_for_interrupt()*.
  Questa routine sospende il processo durante l'attesa di un interrupt su uno dei sei diversi canali di cui cinque sono relativi al chip DAQ-DIO (vedi §3.2) ed uno al chip MITE DMA. Ad ogni canale di interrupt è associato:
  - una coda di attesa **pcidaq_wait[channel]** che viene inizializzata durante la fase di configurazione del gruppo,
  - un contatore **interrupt_ctr[channel]** che viene incrementato ogni volta che viene generato un interrupt relativo all'evento specifico
  - un contatore **sleeping_process[channel]** che viene incrementato ogni volta che un processo è posto in attesa dell'evento

  Se un interrupt è già stato ricevuto sul canale di interesse la funzione decrementa il contatore dell'interrupt e restituisce 0. Se invece l'interrupt deve essere ancora ricevuto, il processo è posto in attesa sotto il controllo di un timeout. La funzione in questo caso ritorna il valore 1 se il timeout viene raggiunto prima dell'arrivo dell' interrupt, oppure 0 in caso contrario.

[11] Come detto al § 4.0.6 questa operazione non è necessaria per acquisizioni finite ma il suo utilizzo è risultato indispensabile in fase di debug. Tale chiamata può essere condizionata al solo debug.

- DIO32_IOC_CHECK_TIMEOUT_GROUP: questo comando può essere eseguito quando un processo viene sospeso in attesa di un interrupt. L'applicazione utente passa come argomenti di input della funzione il canale sul quale viene atteso l'interrupt. In corrispondenza di questo comando `ioctl` viene chiamata la funzione *check_timeout_status()* che restituisce all'applicazione utente il valore del contatore *sleeping_process[channel]*. Se tale valore risulta positivo, allora si è verificato un timeout prima della ricezione dell'interrupt.

- DIO32_IOC_RECONF_GROUP: questo comando consente all'applicazione utente di modificare solo quei parametri di configurazione di un gruppo di I/O che non riguardano l'assegnazione delle porte (e delle FIFO) al gruppo. Ad esempio: modifica del protocollo di handshaking, della modalità di acquisizione, della direzione delle porte (input o output) oppure della sorgente di interrupt.
  Se l'applicazione utente deve eseguire una riprogrammazione del gruppo che prevede anche una nuova assegnazione delle porte di I/O (e quindi della ampiezza di trasferimento), deve essere prima eseguita un'operazione di reset delle FIFO.

- DIO32_IOC_WRITE_PCLK2: questo comando esegue un'operazione di scrittura sulla linea PCLK2.

- DIO32_IOC_WRITE_ACK2: questo comando esegue un'operazione di scrittura sulla linea ACK2.

- DIO32_IOC_READ_REQ2: questo comando legge un dato dalla linea REQ2.

- DIO32_IOC_READ_STOPTRIG2: questo comando legge un dato dalla linea STOPTRIG2.

- DIO32_IOC_MEM_ALLOC: questo comando consente di allocare un buffer di memoria virtuale in corrispondenza del gruppo di I/O associato al device file.
  La chiamata di sistema *ioctl()* deve specificare la dimensione in byte del buffer di memoria, dimensione che deve esssere uguale a quella dei dati da acquisire.
  Il buffer di memoria virtuale viene utilizzato **solo** per acquisizioni che ricorrono all'uso del controller DMA.
  Questo comando `ioctl` attiva a livello del kernel la funzione *do_allocate()* che riceve come argomenti il numero del subdevice e la dimensione della memoria da allocare.
  La funzione *do_allocate()* invoca la routine *mbuff_alloc()*. Questa funzione procede ad allocare la memoria per la struttura del tipo `mbuff` (che racchiude le informazioni relative al buffer DMA vedi §6.3) e successivamente esegue la funzione *rvmalloc()* che:

  - chiama la funzione *vmalloc()* la quale alloca una regione di memoria contigua nello spazio degli indirizzi virtuali della dimensione specificata;
  - riserva le pagine di tale regione in modo che non siano sottoposte al processo di swap.

  Sebbene le pagine che costituiscono il buffer non siamo necessariamente consecutive nella memoria fisica, il kernel le vede come un insieme di indirizzi adiacenti.
  Una delle caratteristiche dei trasferimenti DMA è legata però al fatto che quando il buffer DMA è più grande della dimensione di una pagina fisica, deve occupare pagine contigue nella memoria fisica. Se il buffer ha dimensioni grandi (in Linux esiste un limite superiore di 128 KB) la sua allocazione diventa difficile per un sistema operativo che usa memoria virtuale.
  Un buffer virtuale, cioè allocato con la funzione *vmalloc()*, permette di allocare ampie regioni di memoria, ma non risulta adatto ad essere un buffer DMA in quanto non è rappresentato da un intervallo di indirizzi contigui in memoria fisica.
  La modalità di funzionamento *link-chaining* del controller DMA ci viene incontro per superare questo problema in quanto, come visto al §3.3.1 consente di trasferire dati in regioni di memoria non contigue.
  L' uso combinato della memoria virtuale e della modalità di funzionamento *link-chaining* ci permette di trasferire un buffer di memoria delle dimensioni adatte a contenere tutti i dati dell'acquisizione.
  Un metodo alternativo consiste nel riservare una porzione della memoria fisica durante il boot del sistema. Sebbene quest'ultimo metodo consenta di ottene un blocco di memoria fisica utilizzabile per trasfermenti DMA, la conseguente riduzione della memoria non risulta una soluzione "amichevole" per i SO basati sulla memoria virtuale.

Inoltre questa soluzione risulta poco flessibile, non permettendo il ridimensionamento della memoria durante il processo di acquisizione. Per questi motivi non abbiamo preso in considerazione tale soluzione

- DIO32_IOC_MEM_DEALLOC: questo comando rilascia il buffer di memoria virtuale allocato per l'acquisizione con il DMA per il gruppo di I/O specificato dal device file usato nella chiamata *ioctl()*.
  A livello di driver, viene chiamata la routine *do_deallocate()* che chiama la funzione *mbuff_dealloc()*. Questa routine rilascia la memoria associata al buffer DMA, preoccupandosi di sbloccare le pagine in modo da renderle nuovamente disponibili per il processo di *swap*.

5.3.5 dio32_mmap()

Il metodo `mmap` consente di mappare una regione di memoria del device nello spazio di indirizzamento dell'applicazione utente. In questo modo le operazioni di lettura e scrittura eseguite sul device file corrispondono ad operazioni analoghe compiute sulla regione di memoria mappata.
Qualora l'utente intenda eseguire un trasferimento DMA, deve allocare un buffer di memoria eseguendo il comando *ioctl* `DIO32_IOC_MEM_ALLOC` (vedi § 5.3.4). Questo buffer di memoria viene rimappato nello spazio degli indirizzi dell'applicazione utente quando il programma invoca la chiamata di sistema *mmap()* per il device file associato al gruppo di I/O.
In corrispondenza della chiamata di sistema *mmap()* viene attivata a livello di kernel il metodo `mmap`. Il metodo `mmap` è dichiarato come segue:

int (*mmap) (struct file *filp, struct vm_area_struct *vma);

L'argomento `filp` è un puntatore al file mentre `vma` contiene le informazioni sull' intervallo di indirizzi virtuali usati per accedere al device. Per implementare il metodo `mmap` il driver deve:

- costruire le *page table* per il range di indirizzi;

- ridefinire eventualmente i metodi `open` e `close` associati alla regioni di memoria virtuale `vma` con un nuovo set di funzioni.

Una limitazione del metodo `mmap` è che l'operazione di *mapping* ha una granularità pari a `PAGE_SIZE`[12]. Il kernel può disporre di indirizzi virtuali solo a livello di *page table* perciò la regione mappata deve essere un multiplo di `PAGE_SIZE` e deve risiedere nella memoria fisica ad un indirizzo che sia multiplo di `PAGE_SIZE`, cioè deve essere *paged aligned*.
La funzione *dio32_mmap()* invocata dal metodo `mmap` esegue le operazioni descritte nel seguente modo:

- chiama la routine *rvmmap()*. Questa funzione esegue le verifiche necessarie sulla dimensione e l'indirizzo della regione di memoria da mappare. Inoltre costruisce le *page table* chiamando la funzione *remap_page_range()* per ogni pagina costituente il buffer di memoria virtuale.

- installa il nuovo set di metodi *mb_open()* e *mb_close()* che tengono traccia del contatore d'uso del driver. Questi due metodi incrementato il contatore del modulo tutte le volte che una VMA è aperta, e lo decrementano quando viene chiusa [13].
  Nei kernel più recenti questo lavoro non è necessario perchè il kernel non chiama il metodo *release* se un'area VMA rimane aperta. Il kernel 2.0, comunque, non tiene traccia di ciò, così per avere un codice portabile è conveniente tenere traccia del contarore del modulo.

- incrementa il contatore d'uso del driver (MOD_INC_USE_COUNT). Questo è necessario perchè in questo caso la VMA è creata dalla chiamata di sistema *mmap()*, per cui è invocato il metodo `mmap` e non il metodo `open` della VMA.

- viene attivato il bit SF_MMAP della flag `iobits` per segnalare che il buffer di memoria è stato mappato;

[12] Per un PC IBM compatibile, PAGE_SIZE equivale a 4KB
[13] Il metodo close viene invocato quando l'intera area viene rilasciata

La chiamata di sistema *mmap()* restituisce, se portata a termine correttamente, l'indirizzo nello spazio dell' applicazione utente a cui è stata mappata la regione di memoria.
I dati trasferiti con DMA, sono accessibili andando a leggere direattamente tale regione di memoria. In questo caso non è necessario implementare nel driver il metodo di lettura.

5.4 Rimozione del driver

Il driver viene rimosso dal kernel con il comando *rmmod* che invoca la funzione *cleanup_module()*. Il compito svolto da questa funzione è quello di rilasciare tutte le risorse allocate dal driver prima di liberare il numero maggiore del device file di controllo attraverso la chiamata della funzione *unregister_chrdev()*.
La funzione *cleanup_module()* chiama in successione le funzioni *dio32_free()* e *mite_cleanup()*. La prima rilascia la memoria e le risorse hardware associate a ciascun device chiamando la funzione *nidio_device_detach()*. La seconda libera invece la memoria allocata in fase di inizializzazione per la catena di strutture `mite_struct`.

6 Descrizione delle strutture dati interne

Prendiamo in esame le varie strutture del driver che definiscono i parametri per descrivere in modo completo il funzionamento della scheda.
Alcune di queste strutture sono interne, cioè accessibili solo dal kernel, altre invece sono utilizzate anche dall' applicazione utente per consentire la configurazione del device.

6.1 Struttura ni_device

La struttura `ni_device` riassume le informazioni che caratterizzano il singolo device.

```
typedef struct ni_device_struct ni_device;
struct ni_device_struct{
       int use_count;
       int n_subdevices;
       unsigned char attached;
       int iobase;
       int mite_addr;
       int irq;
       ni_subdevice *subdevice;
       ni_subdevice *read_subdev;
       struct mite_struct *mite;
       void(*trig[5])(ni_device*,dio32_device *);
};
```

Prendiamo in esame i campi della struttura:

use_count : è il contatore che tiene traccia manualmente dell' uso del modulo. Il sistema necessita questa informazione perchè il device file /dev/nipci00 viene aperto in modalità singola, cioè vi può accedere un solo processo utente.

n_subdevices : specifica il numero dei subdevices usati. Ognuno dei due gruppi di I/O è considerato come un subdevice separato. Poichè ci limitiamo a lavorare con un solo gruppo, questo valore viene fissato uguale ad 1 in fase di inizializzazione.

attached : questa flag vale 0 od 1 a seconda che la scheda sia stata inizializzata oppure no. L' inizializzazione della scheda comporta la configurazione del bus come descritto al §3.1.

iobase : questo parametro contiene l'indirizzo base del chipset DAQ-DIO rimappato nello spazio del kernel;

mite_addr : questo campo rappresenta l'indirizzo del chip MITE rimappato nello spazio del kernel;

irq : numero di interrupt;

***subdevice** : puntatore alla struttura del tipo `ni_subdevice` che descrive il singolo gruppo di I/O (vedi §6.2);

***read_subdev** : puntatore alla struttura del subdevice usato durante l'acquisizione. Questa struttura viene inizializzata durante la configurazione hardware eseguita dalla funzione nidio_device_attach().

***mite** : puntatore alla struttura del tipo `mite_struct` (vedi §6.5);

***trig**[5] : puntatori alle funzioni di generazione di protocollo.

6.2 Struttura ni_subdevice

La struttura `ni_subdevice` riassume le informazioni che caratterizzano il singolo gruppo di I/O.

```
struct ni_subdevice_struct {
       unsigned char ngroup;
       unsigned char configured;
       unsigned char group_fifo;
       unsigned char flags;
       int mmap_count;
       short dmaChannel;
       short drqnum;
       unsigned char Data_Path_Copy;
       unsigned char Interrupt_Control_Copy;
       unsigned char DMA_and_Int_Control_Copy;
       unsigned char OpMode_Copy;
       unsigned char DAQ_Options_Copy;
       unsigned char Transfer_Size_Control_Copy;
       int interrupt_ctr[INTERRUPT_CHANNELS];
       int sleeping_process[INTERRUPT_CHANNELS];
       struct wait_queue *pcidaq_wait[INTERRUPT_CHANNELS];
       struct mbuff *mbuff;
       dio32_group cur_group;
       dio32_mode cur_mode;
       dio32_trig cur_trig;
       dio32_interrupt cur_int;
};
```

ngroup : questo parametro rappresenta il numero del gruppo di I/O (1 o 2);

configured : questa flag consente di stabilire se il gruppo di I/O è stato configurato (1) oppure no (0). Il valore di questa flag viene posto uguale ad 1 se la funzione *do_config_ioctl()* viene eseguita correttamente;

group_fifo : il parametro tiene conto delle FIFO assegnate al gruppo;

flags : questa flag tiene traccia delle configurazioni eseguite sul gruppo. In particolare: configurazione del controller MITE (bit SF_MITE selezionato), mappatura del buffer virtuale (bit SF_MMAP selezionato);

mmap_count : questa variabile tiene conto del mapping della memoria virtuale allocata;

dmaChannel : canale DMA del gruppo di I/O;

drqnum : canale DRQ;

Data_Path_Copy,...,Transfer_Size_Control_Copy : questi campi sono la copia del contenuto dei registri corrispondenti;

interrupt_ctr[INTERRUPT_CHANNELS] : contatore degli interrupt che possono verificarsi sui diversi canali;

sleeping_process[INTERRUPT_CHANNELS] : parametro che controlla il numero dei processi in sospeso su un canale di interrupt;

***pcidaq_wait[INTERRUPT_CHANNELS]** : coda di attesa per il processo sospeso in attesa di interrupt;

***mbuff** : puntatore alla struttura del tipo mbuff (vedi §6.3)

cur_group : struttura del tipo dio32_group (vedi §7.1);

cur_mode : struttura del tipo dio32_mode (vedi §7.2);

cur_trig : struttura del tipo dio32_trig (vedi §7.3);

cur_int : struttura del tipo dio32_interrupt (vedi §7.4);

6.3 Struttura mbuff

La struttura `mbuff` descrive il buffer di memoria che viene allocato quando vengono programmati trasferimenti DMA.

```
struct mbuff{
       unsigned char *vbuf;
       unsigned short  mmap_buf_num;
       unsigned long size;
       struct vm_area_struct *vm_area;
       BufferinfoNode *BufferNodePtr;
       BufferinfoNode *BufferHeadNodePtr;
};
```

***vbuf** : puntatore al buffer di memoria virtuale in cui vengono trasferiti i dati dal DMA;

mmap_buf_num : numero di nodi nella catena DMA. Tale valore rappresenta anche in numero di trasferimenti DMA che devono essere effettuati;

size : dimensione in byte del buffer di memoria;

***vm_area** : puntatore alla struttura `vm_area_struct` contenente le informazioni della regione di memoria virtuale (VMA) del processo generata dal *mmapping* del device;

***BufferNodePtr,*BufferHeadNodePtr** : puntatori alla struttura del tipo `BufferinfoNode` (vedi §6.4);

6.4 Struttura BufferinfoNode e MITELinkStruct

Queste due strutture, definite nel file include `pcidio32-dma.h`, vengono usate nella programmazione del controller MITE DMA in modalità *link-chaining*. La struttura `BufferinfoNode` consente di registrare le informazioni relative ad ogni singolo blocco di dati che deve essere trasferito in memoria dal DMA.

```
typedef struct BufferinfoNode {
        struct BufferinfoNode  *NextNode;
        unsigned  long BufferPhysicalAddress;
        unsigned  long NumberOfBytesTransfer;
} BufferinfoNode;
```

***NextNode** : puntatore al nodo successivo;

BufferPhysicalAddress : indirizzo *fisico* del buffer di memoria in cui vengono immagazzinati i dati del trasferimento DMA;

NumberOfBytesTransfer : numero di byte che costituiscono il blocco da trasferire in memoria;

La struttura `MITELinkStruct` raccoglie le informazioni di ogni record di dati usati durante l'autoconfigurazione del controller DMA.

```
typedef struct {
        unsigned long  TCRlink;
        unsigned long  MARlink;
        unsigned long  DARlink;
        unsigned long  LKARlink;
} MITELinkStruct;
```

TCRlink : è il valore che deve essere scritto nel registro TCR. Rappresenta il numero di byte di ogni blocco da trasferire;

MARlink : contiene l'indirizzo fisico del buffer di memoria in cui vengono trasferiti i dati mentre è in funzione il DMA;

DARlink : questo parametro viene posto uguale a 0x1c.

LKARLink : è l'indirizzo fisico del nodo successivo della catena usato dal chip MITE per riprogrammarsi.

6.5 Struttura mite_struct

La struttura `mite_struct` raccoglie le informazioni sulle risorse hardware del device.

```
struct mite_struct{
        struct mite_struct *next;
        int used;
        unsigned char board;
        struct pci_dev *pcidev;
        unsigned long mite_phys_addr;
        unsigned long daq_phys_addr;
        unsigned long mite_io_addr;
        unsigned long daq_io_addr;
};
```

I campi della struttura hanno il seguente significato:

***next** : puntatore ad una struttura dello stesso tipo. In questo modo viene costruita una lista concatenata di strutture, ciascuna relativa ad un diverso device;

used : la variabile assume valore 1 o 0 a seconda che la struttura a cui appartiene corrisponda ad un device già in uso oppure no;

board : numero della scheda a cui è associata la struttura. Assume valore 0 per la prima scheda individuata, 1 per la seconda etc.;

***pcidev** : puntatore alla struttura del tipo `pci_dev`. Questa struttura rappresenta un device PCI ed è usata in ogni operazione che coinvolge device PCI ;

mite_phys_addr : indirizzo fisico del chipset MITE;

daq_phys_addr : indirizzo fisico del chipset DAQ-DIO;

mite_io_addr : indirizzo del chipset MITE rimappato nello spazio di indirizzamento del kernel;

daq_io_addr : indirizzo fisico del chipset DAQ-DIO rimappato nello spazio di indirizzamento del kernel;

7 Descrizione delle strutture dati esterne

In questo capitolo diamo una descrizione delle strutture di dati accessibili dal processo utente. I valori dei campi delle varie strutture vengono definiti dall' applicazione e sono passati nello spazio del kernel attraverso i comandi `ioctl`.

7.1 Struttura dio32_group

La struttura `dio32_group` specifica le caratteristiche di ciascun gruppo ed è definita come segue:

```
struct dio32_group_struct{
       unsigned char groupSize;
       unsigned char dir;
       unsigned char port;
       unsigned char funneling;
       unsigned char pin_mask;
       unsigned char transfer_width;
};
```

Di seguito descriviamo il significato di ciascun elemento.

groupSize : specifica la dimensione del gruppo. In particolare questa variabile può assumere solo 3 valori: **1**,**2** o **4**. A seconda del valore assegnato a `groupSize`, ad ogni gruppo di acquisizone vengono assegnate rispettivamente **1**, **2** o **4** porte di I/O.

dir : specifica la direzione di acquisizione delle porte assegnate al gruppo. Il valore 0 implica che le porte sono configurate per l'input, il valore 1 per l'output.

port : specifica le porte di I/O assegnate al gruppo. L'assegnazione dipende dalla dimensione del gruppo. In particolare se `groupSize` vale **1**, la porta può assumere i valori **0**,**1**,**2**,**3** (corrispondenti alle porte A,B,C,D). Se `groupSize` vale **2**, `port` può assumere il valore **0** (porte A e B) e **2** (porte C e D). Infine se `groupSize` vale **4**, allora `port` può assumere solo valore **0** (porte A,B,C e D).
Tutto ciò può essere schematizzato nel seguente modo:

groupSize	**port**
1	0,1,2,3
2	0,2
4	0

È importante sottolineare che per la famiglia DIO 6533, si possono raggruppare le porte **0** ed **1** insieme e le porte **2** e **3** insieme, ma non la **1** e la **3** oppure la **0** e la **2**.

funneling : questa flag può assumere valore **0** o **2**. Nel primo caso il funneling è disattivato, nel secondo caso no.

pin_mask : rappresenta la maschera dei pin di ciascuna porta. Per operazioni di input questa maschera è significativa solo per il *change detection* una variazione del *pattern generation*.

transfer_width : configura l'ampiezza che il gruppo supporta nelle operazione di lettura, scrittura o accesso DMA. Questo valore deve essere minore od uguale all'ampiezza complessiva del gruppo specificata da `groupSize`. Ad esempio, per usare un trasferimento DMA a 16-bit deve essere scelto un valore di `transfer_width` corrispondente a 16-bit anche se si stanno eseguendo operazioni a 32-bit usando tutte e quattro le FIFO e tutte e quattro le porte. Questa variabile può assumere solo i valori **0/1**,**2** ,**3**. Il significato è il seguente:

- **0,1**: trasferimenti a 32-bit (DMA o FIFO)
- **2**: trasferimenti a 8-bit (DMA o FIFO). I trasferimenti hanno comunque luogo tra questi pins e ciascuna FIFO del gruppo in successione.
- **3**: trasferimenti a 16-bit (DMA o FIFO). I trasferimenti hanno luogo da o verso le FIFO dei gruppi, due alla volta, in successione. Un gruppo che esegue un trasferiemnto DMA a 16-bit deve contenere due o quattro FIFO.

Si può usare la seguente tabella per determinare le combinazioni valide di porte e gruppi che si possono usare con l'ampiezza dei dati che si vogliono trasferire.

Ampiezza Trasferimento	**Valore di groupSize**	**Possibile combinazione porte**	**Gruppi che possono essere usati**
8 bits	1	Port 0	Group 1/Group 2
		Port 1	Group 1/Group 2
		Port 2	Group 1/Group 2
		Port 3	Group1/Group 2
16 bits	2	Port 0,Port 1	Group 1/Group 2
		Port 2,Port 3	Group 1/Group 2
32 bits	4	Port 0,Port 1,Port 2,Port 3	Group 1/Group 2

7.2 Struttura dio32_mode

La struttura `dio32_mode` raggruppa i parametri che consentono di specificare le caratteristiche di un protocollo di handshaking. La struttura contiene i seguenti elementi:

```
struct dio32_mode_struct{
       unsigned char nprot;
       unsigned char edge;
       unsigned char reqPol;
       unsigned char ackPol;
       unsigned char delayTime;
};
```

nprot : specifica il numero che identifica il protocollo di handshaking. I valori che tale variabile può assumere ed i corrispondenti protocolli sono:

- **0**: level-ACK
- **1**: pulsed ACK
- **2**: long pulse ACK
- **3**: burst mode (modalità sincrona)
- **4**: emulazione 8255

edge : può assumere valore 0 o 1 e specifica se il gruppo è configurato per il *leading-edge* o *trailing-edge* e può assumere solo i seguenti valori:

- **0**: leading-edge
- **1**: trailing-edge

Questo parametro ha significato solo per i protocolli **1** e **2**.

reqPol : questo parametro indica se il segnale REQ di richiesta del gruppo deve essere attivo alto o attivo basso. Questo parametro può assumere i seguenti valori:

- **0**: REQ attivo alto
- **1**: REQ attivo basso

Se il protocollo specificato è il **4** (emulazione 8255), questo valore viene ignorato in quanto in tal caso viene utilizzato un segnale REQ attivo basso.

ackPol : indica se il segnale ACQ del gruppo deve essere attivo alto o attivo basso. Questo parametro può assumere i seguenti valori:

- **0**: ACQ attivo alto
- **1**: ACQ attivo basso

Se il protocollo specificato è il **4**, questo valore viene ignorato in quanto in emulazione 8255 viene utilizzato un segnale ACQ attivo basso.

delayTime : per tutti i protocolli, eccetto il burst, specifica il ritardo inserito nel protocollo di handshaking, in multipli di 100 ns fino ad un massimo di 700 ns. Tale parametro può perciò assumere valori compresi tra 0 e 7.
L'effetto introdotto da questo parametro varia da protocollo a protocollo. Per il protocollo *burst* il ritardo seleziona la frequenza del segnale di clock quando è generato da una sorgente esterna.

7.3 Struttura dio32_trig

Gli elementi della struttura `dio32_trig` consentono di definire alcuni dei parametri necessari per il processo acquisizione.

```
struct dio32_trig_struct{
        unsigned char numbered;
        unsigned char sizeInByte;
        unsigned char iomode;
        int transferCount;
};
```

numbered : tale parametro specifica la modalità di trasferimento. In particolare il DAQ-DIO supporta due distinti modi di trasferimento: *numerato*, in cui cui viene eseguito un trasferimento prederminato e finito di dati e *non numerato* (per il momento non supportato dal driver) in cui il trasferimento continua in modo indefinito fino a che non viene interrotto resettando al valore 0 la flag `RunMode` (vedi §4.0.5)

sizeInByte : dimensione in byte dei campioni da acquisire. Per acquisizioni con DMA, questo valore deve uguagliare la dimensione in byte del gruppo. Non viene usato invece per le operazioni in *polling mode*.

iomode : questa flag consente di scegliere all'utente la modalità di acquisizione. Per acquisizioni DMA, il suo valore consente di programmare i registri DMA_MCR e DMA_DCR in modo da supportare trasferimenti da e verso la memoria e da e verso il device a 8, 16 e 32 bit. I valori che tale variabile può assumere sono:

- `USE_DMA`, `USE_DMA_16`, `USE_DMA_32`: il trasferimento dei dati viene eseguito utilizzando il controller DMA integrato sul chip MITE ed i registri DMA_MCR e DMA_DCR vengono programmati per trasferimenti da e verso la memoria e da e verso il device rispettivamente a 8,16 e 32 bit. Nel caso in cui sia stato selezionato un trasferimento finito di dati (variabile `numbered` della struttura `dio32_trig` settata ad 1), la fine del processo di trasferimento viene segnalata dalla generazione di un interrupt da parte del MITE.
- `USE_IRQ`: il trasferimento dei dati viene realizzato appoggiandosi agli interrupt che il device è in grado di generare.
  Per il momento questa opzione non è attiva.
- `USE_READ`:il trasferimento dei dei dati viene effettuato in *polling mode*. Per stabilire se nella FIFO del gruppo di acquisizione sono presenti i dati,viene letto il valore del registro `GROUP_FLAGS` relativo al gruppo. Questa è la modalità di default nel caso in cui la flag `iomode` sia nulla.

transferCount : il numero di dati da trasferire nel caso in cui sia stato selezionata la modalità *numbered*.

7.4 Struttura dio32_interrupt

Gli elementi di questa struttura configurano le linee di interrupt da attivare in modo che in corrispondenza dell'evento, il device sia in grado di generare un interrupt.

```
 struct dio32_interrupt_struct {
        unsigned char TR_interrupt;
        unsigned char CE_interrupt;
        unsigned char Waited_interrupt;
        unsigned char P_TC_interrupt;
        unsigned char S_TC_interrupt;
};
```

TR_interrupt—,CE_interrupt,.. : queste flags possono assumere solo il valore 0 o 1.
Quando viene configurata ad 1, il rispettivo bit del registro di interrupt della scheda viene abilitato ed un interrupt può essere generato in corrispondenza del relativo evento[14].

7.5 Struttura dio32_config

La struttura `dio32_config` consente di raggruppare tutte le informazioni realative alla configurazione di un gruppo di I/O. In particolare la struttura `dio32_config` è definita come segue:

```
 struct dio32_config_struct {
        unsigned char subdev;
        unsigned char ngroup;
        dio32_group cur_group;
        dio32_mode cur_mode;
        dio32_trig cur_trig;
        dio32_interrupt cur_int;
};
```

Il primo elemento della struttura configura il subdevice in uso (0 per il primo, 1 per il secondo), mentre il secondo campo della struttura specifica il gruppo di I/O associato al subdevice.
Gli altri campi fanno riferimento a strutture già descritte nei paragrafi precedenti.

[14] In realtà questo non è sufficiente. La generazione di un interrupt da parte del device deve essere abilitata in modo globale. A tale scopo devono essere posti ad 1 i primi due bits del registro MASTER_DMA_AND_INTERRUPT_CONTROL

8 Organizzazione dei sorgenti del driver

I sorgenti che servono per la costruzione del modulo del driver, sono contenuti in due directory distinte: la directory *include* e la directory *src*. La prima comprende gli includes files, la seconda i sorgenti C ed il Makefile per la costruzione del modulo.
I sorgenti del driver e le relative versioni di sviluppo, sono controllati dal sistema CVS (version control system).

8.1 Directory src

I files sorgenti sono stati organizzati in modo da rispecchiare l'organizzazione del driver. In particolare:

pcidio32-inits.c : include le funzioni *init_module* e *cleanup_module()* per la registrazione e le rimozione del driver come pure la routine *dio32_free()* che esegue il rilascio della memoria allocata per le strutture del device. Nello stesso sorgente è definita la funzione *get_device_by_minor()* che permette di selezionare la struttura del tipo `ni_device` relativa al device fisico in uso.

pcidio32-fops.c : comprende la definizione delle operazioni eseguibili sul device file elencate al § 5.3.
Nello stesso sorgente sono definite anche le funzioni *mb_open()* e *mb_close()* che costituiscono i metodi associati alle aree di memoria virtuale create dall'operazione di mappatura della memoria.

pcidio32-defs.c : contiene le funzioni che vengono chiamate dai vari comandi *ioctl*.

pcidio32-ints.c : definisce le routine associate al gestore di interrupt.

pcidio32-daq.c : include le funzioni di più basso livello che configurano e leggono i registri del chip DAQ-DIO.

pcidio32-util.c : definisce le funzioni che eseguono il controllo sui parametri di configurazione della scheda.

mite.c : definisce le funzioni per il rilevamento ed inizializzazione delle schede PCI-DIO32HS della NI.

dma.c : contiene le funzioni necessarie per programmare in modalità *link-chaining* il controller MITE DMA, ed alcune funzioni correlate al suo corretto funzionamento.

kvmem.c comprende le funzioni per l' allocazione, il mapping ed il rilascio della memoria virtuale. Queste routine sono un parziale riadattamento di quelle scritte per i driver *bttv* e *mbuff*[15]. I sorgenti di quest'ultimo sono prelevabili dal sito internet

http://crds.chemie.unibas.ch/PCI-MIO-E/ .

8.2 Directory include

La directory `include` contiene i seguenti files:

pcidio32-kernel.h : comprende la definzione delle strutture dati `ni_device` e `ni_subdevice` associate al device driver ed i prototipi delle funzioni.

pcidio32-ioctl.h include la definzioni dei comandi *ioctl* e le strutture dati del gruppo di I/O. Queste sono state poste in un file separato, perchè devono essere accessibili alle applicazioni utenti. È preferibile una netta separazione tra le strutture accessibili esclusivamente dal kernel (definite appunto all'interne del file pcidio32-kernel.h) e quelle accessibili anche a livello di processo utente.

pcidio32-dma.h : comprende l'elenco dei registri del chipset MITE DMA e la definizione delle strutture dati utilizzate per la sua configurazione.

mite.h : definisce la struttura dati `mite_struct` e la dichiarazione delle funzioni di rilevamento ed inizializzazione dell'hardware.

[15] Il driver Linux mbuff è stato sviluppato da Tomasz Motylewski

daqdio.h : contiene l'elenco degli offset dei registri del chip DAQ-DIO.

mitereg.h : comprende l' elenco dei bitfield dei registri del chipset MITE. Questo file è fornito dalla National Instrument.

bitfields.h : include la definizione di alcune macro che consentono di accedere in modo semplice ai singoli bit dei registri del chip DAQ-DIO.

kvmem.h : comprende la definizione delle funzioni che gestiscono la memoria virtuale.

sysdep.h : questo file raggruppa le incompatibilità tra le versioni 2.0, 2.2 e 2.4 del kernel Linux.

pci-compat.h : questo header file deve essere incluso per il supporto corrente del PCI quando il driver è compilato per la versione 2.0 del kernel.

Questi due ultimi file sono stati scritti da A.Rubini e J.Corbet e sono prelevabili, insieme ai sorgenti degli esempi riportati in [1], dal sito internet:

http://examples.oreilly.com/linuxdrive2 .

9 Costruzione ed uso del driver

La compilazione del driver è eseguita automaticamente dal Makefile contenuto nella directory dei sorgenti. L' operazione di compilazione produce il file oggetto *dio32.o*. Il comando **make install** consente di installare il driver nella directory `misc` dei moduli relativi alla versione del kernel per il quale il driver è stato compilato.
Per usare il device driver il modulo *dio32* deve essere caricato nel kernel specificando il comando:

insmod dio32

Prima di eseguire l'applicazione utente, si devono creare nella directory `/dev` i device files `nipci00` per la prima scheda, `nipci10` per la seconda scheda e così via.
Questi devices files vengono creati con il comando *mknod* specificando il tipo di device (caratteri) ed il numero maggiore e minore. Il numero maggiore, come detto in precedenza è posto uguale a 60. I numeri minori dei device files sono scelti nel seguente modo:
minor_number = card_number
dove `card_number` vale 0 per la prima scheda, 1 per la seconda etc.

10 Test di funzionamento del driver

Il driver Linux per la scheda PCI-DIO32HS della National Instruments è stato compilato e testato per le versioni 2.2.19 e 2.4.17 del kernel Linux.
Un lavoro analogo **non** è stato eseguito per la versione 2.0 del kernel.
Il funzionamento del driver è stato provato utilizzando un'apparecchiatura costruita in laboratorio che ci ha consentito di generare sia i dati sia i segnali di handshaking. In particolare abbiamo scelto di lavorare con il protocollo di emulazione 8255.
Per eseguire il test del driver abbiamo scritto alcune applicazioni utenti che si differenziano solo sulla modalità con cui vengono acquisiti i dati: *polling* oppure DMA.
I test di funzionamento sono stati eseguiti:

- programmando solo il gruppo 1 di I/O (gruppo 1) ed associando ad esso le quattro FIFO e le quattro porte di I/O
- eseguendo operazioni di sola lettura con un numero finito di dati
- usando le linee PCLK2,ACK2,REQ2 e STOPTRIG2 come linee di I/O extra per comunicare con l'apparecchiatura esterna.

Nel trasferimento di quantità di dati maggiori di 1KByte abbiamo ottenuto le seguenti velocità di trasferimento:

- 5.5-5.8 MB/s in modalità DMA
- 2.5-2.9 MB/s in *polling mode*

Ci preme sottolineare che esiste un limite interno del driver per l'allocazione del buffer virtuale usato per il trasferimento, limite che si manifesta quando si prova ad acquisire 32Mbyte di dati o più.
In questo caso viene generato un errore dalla routine **DMA_Program_Minimite()** che esegue la programmazione del DMA. Una delle operazioni svolte da questa funzione consiste nel riservare uno spazio di memoria per un vettore di strutture del tipo `MITELinkStruct`. Se la memoria richiesta per il vettore è maggiore di 128 KByte, limite massimo di memoria allocabile con kmalloc(), la funzione restituisce un errore.
I 128KByte vengono superati nel caso in cui il vettore abbia più di 8192 elementi [16].
Questo incoveniente può essere rimosso se invece di allocare un vettore di strutture del tipo MITELinkStruct, si costruisce una lista concatenata di strutture dello stesso tipo.

10.1 Conclusioni

Uno dei limiti di questo driver consiste nella necessità di allocare una regione di memoria, non utilizzabile dagli altri processi, pari alla dimensione in byte dei dati da acquisire.
Questa scelta è stata una conseguenza delle limitate informazioni che avevamo sulla programmazione del controller MITE DMA.
Nella versione precedente del driver, infatti, l'unico metodo per sapere quando il trasferimento DMA di un singolo blocco di dati era terminato, consisteva nel leggere in polling il registro TCR del controller MITE che contiene il numero di bytes che devono essere ancora trasferiti in un singolo link.
Successivamente, studiando il codice del driver *comedi* siamo riusciti a capire come programmare il controller DMA in modo da generare un interrupt alla fine del trasferimento di ogni singolo blocco e/o al termine del trasferimento completo. La nuova versione ha però mantenuto la vecchia struttura del driver, sostituendo solo la funzione che controlla la fine dell'acquisizione.
Uno degli obiettivi futuri consiste nell'implementare una routine di lettura che si appoggi al DMA.

11 References

[1] E.Giani,A.Checcucci,C.Baffa "Driver Linux per la scheda di acquisizione NI PCI-DIO32HS", Arcetri Technical Report n.7/01
[2] A.Rubini,J.Corbet "Linux device drivers",O'Reilly

[16] la dimensione in byte della struttura MITELinkStruct è 16 byte

12 Appendice A

12.1 Registri del chip DAQ-DIO

La seguente tabella mostra la mappa dei registri del chip DAQ-DIO elencati in ordine alfabetico

Register Name	Offset (decimal)	Type
Chip_ID	24-27	Read
Data_Path	64 (Group 1) 96 (Group 2)	Write
DMA_Line_Control	76 (Group 1) 108 (Group 2)	Write
FIFO_A	16	Read/Write
FIFO_B	17	Read/Write
FIFO_C	18	Read/Write
FIFO_D	19	Read/Write
Group_1_FIFO	8-11	Read/Write
Group_1_First_Clear	6	Write
Group_1_Flags	6	Read
Group_1_Second_Clear	46	Write
Group_2_FIFO	12-15	Read/Write
Group_2_First_Clear	7	Write
Group_2_Flags	7	Read
Group_2_Second_Clear	47	Write
Group_Status	5	Read
Inteerupt_and_Window_Status	4	Read
Interrupt_Control	75 (Group 1) 107 (Group 2)	Write
Master_Clock_Routing	45	Write
Master_DMA_and_Interrupt_Control	5	Write
Pattern_Detection	81 (Group 1) 113 (Group 2)	Write
Port_A_Input	28	Read
Port_A_Output	28	Write
Port_A_Pattern	48	Write
Port_A_Pin_Directions	32	Write
Port_A_Pin_Mask	36	Write
Port_A_Polarities	40	Write

Port_B_Input	29	Read
Port_B_Output	29	Write
Port_B_Pattern	49	Write
Port_B_Pin_Directions	33	Write
Port_B_Pin_Mask	37	Write
Port_B_Polarities	41	Write
Port_C_Input	30	Read
Port_C_Output	30	Write
Port_C_Pattern	50	Write
Port_C_Pin_Directions	34	Write
Port_C_Pin_Mask	38	Write
Port_C_Polarities	42	Write
Port_D_Input	31	Read
Port_D_Output	31	Write
Port_D_Pattern	51	Write
Port_D_Pin_Directions	35	Write
Port_D_Pin_Mask	39	Write
Port_D_Polarities	43	Write
Protocol_Register_1	65 (Group 1) 97 (Group 2)	Write
Protocol_Register_2	66 (Group 1) 98 (Group 2)	Write
Protocol_Register_3	67 (Group 1) 99 (Group 2)	Write

Protocol_Register_4	70 (Group 1) 102 (Group 2)	Write
Protocol_Register_5	71 (Group 1) 103 (Group 2)	Write
Protocol_Register_6	73 (Group 1) 105 (Group 2)	Write
Protocol_Register_7	74 (Group 1) 106 (Group 2)	Write
Protocol_Register_8	88-91 (Group 1) 120-123 (Group 2)	Write
Protocol_Register_9	82 (Group 1) 114 (Group 2)	Write
Protocol_Register_10	83 (Group 1) 115 (Group 2)	Write
Protocol_Register_11	84 (Group 1) 116 (Group 2)	Write
Protocol_Register_12	85 (Group 1) 117 (Group 2)	Write
Protocol_Register_13	86 (Group 1) 118 (Group 2)	Write
Protocol_Register_14	68-69 (Group 1) 100-101 (Group 2)	Write
Protocol_Register_15	79 (Group 1) 111 (Group 2)	Write
Reserved for board-level circuitry (RSTI switch)	124-127	Write
Subspace_1 Transfer_Count	56-63 20-23 (Group 1) 24-27 (Group 2)	Read/Write Write
Transfer_Size_Control	77 (Group 1) 109 (Group 2)	Write
Window_Address	4	Write
Window_Data	0-3	Read/Write

13 Appendice B

13.1 Locazione dei bit dei registri del chip DAQ-DIO

Data_Path

Group1 Offset: 64
Group2 Offset: 96
Type:Write Only

0	FIFOEnables(A)
1	FIFOEnables(B)
2	FIFOEnables(C)
3	FIFOEnables(D)
4	funneling
5	funneling
6	
7	GroupDirection

DMA_Line_Control

Group1 Offset: 76
Group2 Offset: 108
Type:Write Only

0	DMAChannel
1	DMAChannel
2	DMAChannel
3	DMAChannel
4	Reserved
5	Reserved
6	Reserved
7	Reserved

FIFO_Control

Group1 Offset: 72
Group2 Offset: 104
Type:Write Only

0	ReadyLevel
1	ReadyLevel
2	ReadyLevel
3	Reserved
4	Reserved
5	Reserved
6	Reserved
7	Reserved

Group_FIFOs

Group1 Offset: 8-11
Group2 Offset: 12-15
Type: Read/Write

0	FIFOdata
1	FIFOdata
2	FIFOdata
3	FIFOdata
4	FIFOdata
5	FIFOdata
6	FIFOdata
7	FIFOdata

Group_First_Clear

Group1 Offset: 6
Group2 Offset: 7
Type:Write Only

0	
1	
2	
3	ClearWaited
4	Clear PrimaryTC
5	Clear SecondaryTC
6	DMAReset
7	FIFOReset

Group_Flags

Group1 Offset: 6
Group2 Offset: 7
Type:Read Only

0	TransferReady
1	CountExpired
2	
3	
4	
5	Waited(overflow/underflow)
6	PrimaryTC
7	SecondaryTC

Group_Second_Clear

Group1 Offset: 46
Group2 Offset: 47
Type:Write Only

0	ClearExpired
1	
2	Reserved
3	Reserverd
4	Reserved
5	Reserved
6	Reserved
7	Reserved

Group_Status

Offset: 5

Type: Read Only

0	DataLeft(1)
1	Reserved
2	Req(1)
3	StopTrig(1)
4	DataLeft(2)
5	Reserved
6	Req(2)
7	StopTrig(2)

Interrupt_and_Window_Status

Offset: 4

Type:Read Only

0	IntStatus(1)
1	IntStatus(2)
2	WindowAddressStatus
3	WindowAddressStatus
4	WindowAddressStatus
5	WindowAddressStatus
6	WindowAddressStatus
7	Reserved

Interrupt_Control

Group1 Offset: 75

Group2 Offset: 107

Type:Write Only

0	IntEnables(0) (Transfer Ready)
1	IntEnables(1) (Count Expired)
2	
3	
4	
5	IntEnables(5) (Waited)
6	IntEnables(6) (PrimaryTC)
7	IntEnables(7) (SecondaryTC)

Master_DMA_and_Interrupt_Control

Offset: 5

Type:Write Only

0	InterruptLine
1	InterruptLine
2	OpenInt
3	
4	Reserved
5	Reserved
6	Reserved
7	Reserved

Master_Clock_Routing

Offset: 45

Type: Read Only

0	Reserved
1	Reserved
2	Reserved
3	Reserved
4	RTSIClocking
5	RTSIClocking
6	Reserved
7	Reserved

Port_Pin_Directions

Port_A	Offset: 32
Port_B	Offset: 33
Port_C	Offset: 34
Port_D	Offset: 35

Type:Write Only

0	PinDirections
1	PinDirections
2	PinDirections
3	PinDirections
4	PinDirections
5	PinDirections
6	PinDirections
7	PinDirections

Port_Pin_Mask

Port_A	Offset: 36
Port_B	Offset: 37
Port_C	Offset: 38
Port_D	Offset: 39

Type:Read Only

0	PinMask
1	PinMask
2	PinMask
3	PinMask
4	PinMask
5	PinMask
6	PinMask
7	PinMask

Protocol_Register_1

Group1	Offset: 65
Group2	Offset: 97

Type:Write Only

0	RunMode
1	RunMode
2	RunMode
3	Numbered
4	Protocol(1)
5	Protocol(1)
6	Protocol(1)
7	

Protocol_Register_2

Group1	Offset: 66
Group2	Offset: 98

Type:Write Only

0	Protocol(2)
1	Protocol(2)
2	Protocol(2)
3	Protocol(2)
4	Protocol(2)
5	Protocol(2)
6	Protocol(2)
7	InverStopTrig

Protocol_Register_3

Group1 Offset: 67
Group2 Offset: 99
Type:Write Only

0	Protocol(3)
1	Protocol(3)
2	Protocol(3)
3	Protocol(3)
4	Protocol(3)
5	Protocol(3)
6	Protocol(3)
7	Protocol(3)

Protocol_Register_4

Group1 Offset: 70
Group2 Offset: 102
Type:Write Only

0	Protocol(4)
1	Protocol(4)
2	Protocol(4)
3	Protocol(4)
4	Protocol(4)
5	Protocol(4)
6	Protocol(4)
7	Protocol(4)

Protocol_Register_5

Group1 Offset: 71
Group2 Offset: 103
Type:Write Only

0	Protocol(5)
1	Protocol(5)
2	Protocol(5)
3	Protocol(5)
4	Protocol(5)
5	Protocol(5)
6	Protocol(5)
7	Protocol(5)

Protocol_Register_6

Group1 Offset: 73
Group2 Offset: 105
Type:Write Only

0	InvertAck
1	InvertReq
2	InvertClock
3	InvertSerial
4	OpenAck
5	OpenClock
6	Reserved
7	Reserved

Protocol_Register_7

Group1 Offset: 74
Group2 Offset: 106
Type:Write Only

0	Protocol(7)
1	Protocol(7)
2	Protocol(7)
3	Protocol(7)
4	Protocol(7)
5	Protocol(7)
6	Protocol(7)
7	Protocol(7)

Protocol_Register_8

Group1 Offset: 88-91
Group2 Offset: 120-123
Type:Write Only

0	Protocol(8)
1	Protocol(8)
2	Protocol(8)
3	Protocol(8)
4	Protocol(8)
5	Protocol(8)
6	Protocol(8)
7	Protocol(8)

Protocol_Register_9

Group1 Offset: 82
Group2 Offset: 114
Type:Write Only

0	Protocol(9)
1	Protocol(9)
2	Protocol(9)
3	Protocol(9)
4	Protocol(9)
5	Protocol(9)
6	Protocol(9)
7	Protocol(9)

Protocol_Register_10

Group1 Offset: 83
Group2 Offset: 115
Type:Write Only

0	Protocol(10)
1	Protocol(10)
2	Protocol(10)
3	Protocol(10)
4	Protocol(10)
5	Protocol(10)
6	Protocol(10)
7	Protocol(10)

Protocol_Register_11

Group1 Offset: 84
Group2 Offset: 116
Type:Write Only

0	Protocol(11)
1	Protocol(11)
2	Protocol(11)
3	Protocol(11)
4	Protocol(11)
5	Protocol(11)
6	Protocol(11)
7	Protocol(11)

Protocol_Register_12

Group1 Offset: 85
Group2 Offset: 117
Type:Write Only

0	Protocol(12)
1	Protocol(12)
2	Protocol(12)
3	Protocol(12)
4	Protocol(12)
5	Protocol(12)
6	Protocol(12)
7	Protocol(12)

Protocol_Register_13

Group1 Offset: 86
Group2 Offset: 118
Type:Write Only

0	Protocol(13)
1	Protocol(13)
2	Protocol(13)
3	Protocol(13)
4	Protocol(13)
5	Protocol(13)
6	Protocol(13)
7	Protocol(13)

Protocol_Register_14 (16 bits)

Group1 Offset: 68-69
Group2 Offset: 100-101
Type:Write Only

0	Protocol(14)
1	Protocol(14)
2	Protocol(14)
3	Protocol(14)
4	Protocol(14)
5	Protocol(14)
6	Protocol(14)
7	Protocol(14)

Protocol_Register_15

Group1 Offset: 79
Group2 Offset: 111
Type:Write Only

0	StartSource
1	StartSource
2	InvertStart
3	StopSource
4	StopSource
5	Reserved
6	ReqSource
7	PreStart

Transfer_Size_Control

Group1 Offset: 77
Group2 Offset: 109
Type:Write Only

0	TransferWidth
1	TransferWidth
2	Reserved
3	TransferLength
4	TransferLength
5	RequireRLevel
6	Reserved
7	Reserved

Transfer_Count (32 bits)

Group1 Offset: 20
Group2 Offset: 24
Type:Write Only

0	TransferCount
1	TransferCount
2	TransferCount
3	TransferCount
4	TransferCount
5	TransferCount
6	TransferCount
7	TransferCount

14 Appendice C

14.1 Registri per la programmazione del protocollo

Protocol/ Protocol Reg.	1 OpMode	2 ClockReg	3 Sequence	4 ReqReg	5 BlockMode	6 LinePolarities	7 AckSer	8 StartDelay	9 ReqDelay	10 ReqNotDelay	11 AckDelay	12 AckNotDelay	13 DataDelay	14 ClockSpeed	15 DAQ Options
8255 Input Emulation	96	5	38	0	0[1]	3	96	pd x 2 +1	2	1	2	1	1	0	0
8255 Output Emulation	0	0	38	0	0[1]	3	96	pd x 2 +1	2	1	2	1	1	0	0
Level Mode Input	0 non 64 latch	1	7	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	3	pd x 2 +1	pd x2 +1	2	1	0	0
Level Mode Output	0 non 96 latch	0	7	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	3	pd x 2 +1	pd x2 +1	2	1	0	0
Leading Edge Pulse Mode Input	0 non 96 latch	0	11	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	3	pd x 2 +1	pd x2 +1	3	1	0	0
Leading Edge Pulse Mode Output	0 non 64 latch	1	7	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	3	pd x 2 +2	2	3	1	0	0
Long Pulse Mode Input	0 non 64 latch	1	7	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	3	pd x 2 +2	2	3	1	0	0
Long Pulse Mode Output	0 non 64 latch	0	7	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	3	pd x 2 +2	2	3	1	0	0
Trailing Edge Mode Input	0 non 96 db buf latch	1	2	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	pd x 2 +7[4]	3	2	pd x 2 +7[4]	1	0	0
Trailing Edge Mode Output	0 non 96 db buf[2]	0	2	0	0[1]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	pd x 2 +1	pd x 2 +7[4]	3	2	pd x 2 +7[4]	1	0	0
Burst Mode Input (drive PCLK)	0	96	0	8	4[5]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	1	1	1	1	1	1	pd	0
Burst Mode Output (receive PCLK)	0	16	0	32	4[5]	1 inv ACK[3] 2 inv REQ[3] 3 both inv[3]	96	1	1	1	1	1	1	0	0

[1] Set to 0; if performing funneling, set to 24

[2] Use the former value if you wish to disable REQ-edge latching (default mode for output). Usee the latter value if you wish to enable REQ-edge latching (default mode for input).

[3] Set to 0 for active-high ACK and REQ, 1 for active-low ACK, 2 for active-low REQ, or 3 for active-low ACK e REQ.

[4] pd= programmable delay, from 0 to 7, in hundreds of ns.

[5] Set to 4; if performing 8:16 funneling, set to 0.