Capitolo 83

# Primi passi verso un sistema per hardware x86-32

`CLI` 12 `INB` 11 `IRET` 24 `LGDT` 20 `LIDT` 23 `OUTB` 11 `STI` 12

Questo capitolo raccoglie le informazioni basilari per la realizzazione di un sistema autonomo, privo però di funzionalità utili. Per chiarire i concetti raccolti in questo capitolo è molto importante affiancare la lettura di *Intel Architectures Software Developer's Manual, System Programming Guide* (ottenibile presso *http://developer.intel.com/products/processor/manuals/index.htm* ).

Per affrontare il capitolo è opportuno prendere prima confidenza con la sezione 65.5, nella quale si guida a realizzare un programma, avviato attraverso GRUB o SYSLINUX, in grado semplicemente di visualizzare un messaggio sullo schermo.

Per rendere agevoli gli esperimenti descritti in questo capitolo, è necessario utilizzare Bochs o QEMU, ovvero di un emulatore di architettura x86-32. Supponendo di avere predisposto un file-immagine di un dischetto, in cui si avvia il proprio kernel sperimentale attraverso GRUB 1 o di SYSLINUX, conviene predisporre uno script per l'avvio di Bochs o di QEMU senza doversi preoccupare di altro:

```
#!/bin/sh
bochs -q 'boot:a' 'floppya: 1_44=floppy.img, status=inserted'
```

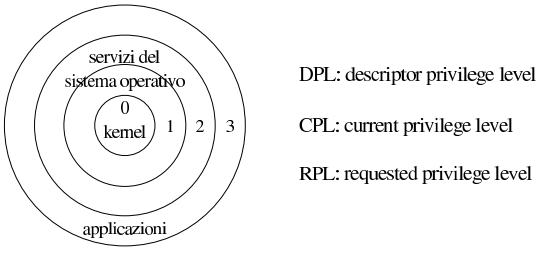
```
#!/bin/sh
qemu -fda floppy.img -boot a
```

Come si comprende intuitivamente, il file-immagine del dischetto deve chiamarsi '`floppy.img`'.

## 83.1 Privilegi dei segmenti

La gestione dei microprocessori x86-32 in modalità protetta, prevede che i dati e i processi elaborativi siano generalmente classificati in base a dei privilegi, secondo un modello ad anelli.

Figura 83.3. Modello di rappresentazione dei privilegi ad anelli.





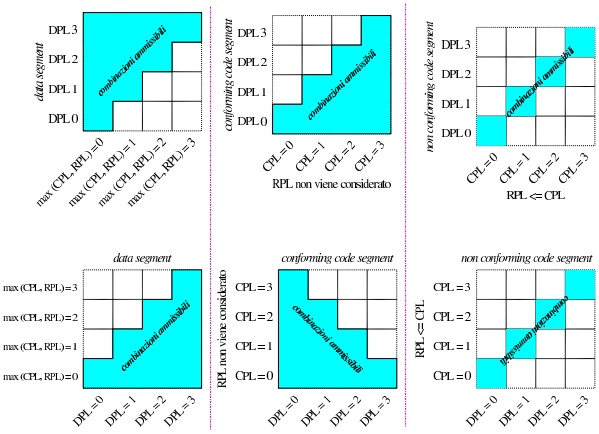

I microprocessori x86-32 definiscono precisamente quattro anelli, numerati da zero a tre e vi si attribuiscono convenzionalmente delle competenze: al livello zero, corrispondente all'anello centrale, competono i privilegi più importanti, ovvero quelli del kernel; al livello tre, corrispondente all'anello più esterno, competono i privilegi meno importanti, ovvero quelli delle applicazioni. In altri termini, gli anelli più interni, a cui corrisponde un valore numericamente minore, sono dati privilegi maggiori rispetto a quelli più esterni.

### 83.1.1 DPL, CPL e RPL

Nei microprocessori x86-32 si usano delle definizioni per rappresentare tre contesti diversi in cui sono considerati i privilegi ad anelli: DPL, ovvero *descriptor privilege level*; CPL, ovvero *current privilege level*; RPL, ovvero *requested privilege level*. La sigla DPL rappresenta un privilegio attribuito a un «oggetto»; pertanto, il descrittore del tale oggetto porta con sé l'indicazione del privilegio a cui questo fa riferimento. La sigla CPL rappresenta il privilegio attivo per il processo elaborativo in corso di esecuzione. La sigla RPL rappresenta il privilegio richiesto per accedere a un certo oggetto e potrebbe essere diverso dal privilegio del processo elaborativo attuale (CPL).

Le situazioni in cui si applica il controllo dei privilegi sono varie, ma semplificando in modo un po' approssimativo si presentano tre possibilità fondamentali: codice che deve raggiungere dati; codice che deve raggiungere altro codice di tipo «conforme»; codice che deve raggiungere altro codice di tipo «non conforme». L'aggettivo «conforme» associato al codice serve solo a distinguere due comportamenti alternativi e non ha molta importanza individuare il significato originale dato al termine usato.

Figura 83.4. Combinazioni tra CPL, RPL e DPL, nelle tre situazioni più comuni, secondo due prospettive alternative.




**Codice che deve raggiungere dei dati**

Quando si deve accedere a dei dati, in un'area di memoria a cui ci si riferisce attraverso un descrittore, il livello di privilegio di tale descrittore (DPL) deve essere numericamente maggiore, sia di CPL, sia di RPL. Pertanto il processo elaborativo ha accesso a dati meno importanti del proprio livello; se però si vuole limitare ulteriormente l'importanza dei dati a cui si può accedere, si può utilizzare un valore RPL numericamente più alto del proprio livello effettivo.

**Codice che deve raggiungere altro codice conforme**

Quando il codice in corso di esecuzione deve saltare verso un'altra posizione, qualificata come «conforme», il descrittore che si riferisce alla memoria che contiene tale nuovo codice deve avere un livello di privilegio numericamente minore o uguale a quello effettivo del codice di origine. Pertanto il processo elaborativo può spostarsi a utilizzare codice con lo stesso livello di privilegio o a codice con un privilegio più importante. In tal caso, il valore di RPL non viene considerato.

Per «codice conforme» vanno intese quindi delle procedure che sono «sicure» per tutti i processi con importanza inferiore o al massimo uguale a quella delle procedure stesse. La conformità si può riferire al concetto di standardizzazione delle procedure, come nel caso di librerie di funzioni.

**Codice che deve raggiungere altro codice non conforme**

Quando il codice in corso di esecuzione deve saltare verso un'altra posizione, qualificata come «non conforme», il descrittore che si riferisce alla memoria che contiene tale nuovo codice deve avere un livello di privilegio identico a quello effettivo del codice di origine. In questo caso, il valore di RPL è importante solo in quanto deve essere numericamente inferiore o uguale a quello di CPL.

Il codice «non conforme» è quello che non ha requisiti di standardizzazione e di sicurezza tali da consentire una condivisione con i processi elaborativi con un privilegio meno importante; d'altra parte, per motivi diversi, non è nemmeno abbastanza sicuro da poter essere riutilizzato da processi elaborativi più importanti.

### 83.1.2 Pila dei dati

Per ogni livello di privilegio che può assumere un processo elaborativo, deve essere disponibile una pila dei dati differente. Pertanto, il processo elaborativo che sta funzionando con un livello di privilegio attuale (CPL) pari a zero, deve utilizzare una pila che si colloca in un'area di memoria qualificata da un livello di privilegio del descrittore (DPL) pari a zero. Lo stesso vale per gli altri livelli di privilegio. Ciò che qui non viene spiegato è il modo in cui un processo può modificare il proprio livello di privilegio attuale (CPL) e acquisire, di conseguenza, un'altra pila dei dati.

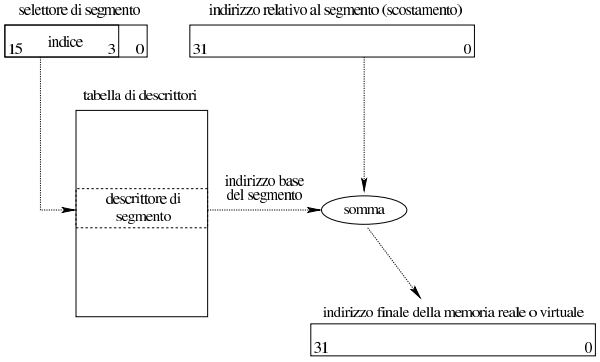
Quando il processo elaborativo raggiunge del codice «conforme» a partire da un livello di privilegio attuale meno importante di quello del codice in questione, il valore di CPL non cambia, quindi non cambia nemmeno la pila dei dati relativa al processo elaborativo.

### 83.1.3 Segmenti, selettori e registri

La gestione della memoria di un microprocessore x86-32, funzionante in modalità protetta, richiede che la memoria sia organizzata in segmenti, i quali, eventualmente possono essere suddivisi in pagine di memoria virtuale. A ogni modo, i segmenti rappresentano sempre il punto di riferimento principale e vengono specificati attraverso l'aiuto di registri di segmento.

Per individuare un indirizzo di memoria (reale o virtuale), si parte da un ***selettore***, contenuto in un registro di segmento appropriato al contesto, dal quale si ottiene un indice per selezionare una voce da una tabella di descrittori. Attraverso l'indice si individua il descrittore di un segmento, del quale si ottiene l'indirizzo iniziale nella memoria (reale o virtuale). A questo indirizzo iniziale va poi aggiunto uno scostamento che rappresenta l'indirizzo relativo all'interno del segmento.

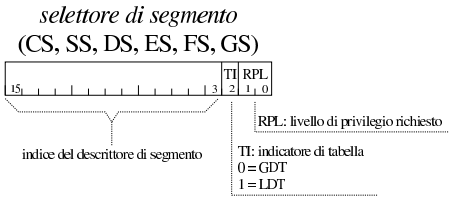
Figura 83.5. Determinazione dell'indirizzo di memoria, attraverso l'indicazione di un indirizzo relativo a un segmento.






I registri all'interno dei quali vanno inseriti i selettori di segmento possono essere ***CS*** (*code segment*), ***SS*** (*stack segment*), ***DS*** (*data segment*), ***ES***, ***FS*** e ***GS*** (sono tutti registri a 16 bit). In particolare, il registro ***CS*** serve a individuare il segmento in cui è in corso di esecuzione il codice attuale; il registro ***SS*** individua il segmento in cui si trova la pila dei dati utilizzata dal processo elaborativo attuale; il registro ***DS*** e gli altri individuano dei segmenti contenenti altri tipi di dati, a cui il processo elaborativo in corso deve accedere. Il valore che si scrive in questi registri è il selettore di segmento, il quale va interpretato secondo lo schema della figura successiva.

Figura 83.6. Interpretazione del selettore di segmento che si attribuisce ai registri relativi.




Nella figura va osservato che i primi due bit del valore che costituisce il selettore di segmento, rappresentano i privilegi richiesti (RPL), mentre il terzo bit precisa il tipo di tabella nella quale cercare il descrittore di segmento.

Per quanto riguarda invece i privilegi attuali (CPL) di cui dispone un processo elaborativo, questi sono il valore corrispondente ai primi due bit del segmento ***CS*** e ***SS***; pertanto si ottengono **leggendo** tali registri. Quando si assegna un valore al registro ***CS***, in pratica si utilizza un'istruzione di salto o una chiamata di procedura, per la quale si specifica sia il segmento, sia l'indirizzo relativo al segmento. È in questa fase che viene indicato il livello di privilegio richiesto (RPL), in quanto il valore del segmento, ma più precisamente si tratta del selettore di segmento, contiene tale indicazione. Ammesso che l'operazione sia valida, i privilegi effettivi sono quelli che rimangono poi nel registro, dopo la sua esecuzione.

Figura 83.7. Interpretazione dello stato dei registri ***CS*** e ***SS***.


Per quanto riguarda specificatamente i segmenti, i privilegi individuati dalla sigla DPL sono quelli annotati nel descrittore di segmento (della tabella relativa) al quale si vuole accedere. Esistono comunque altri contesti in cui compaiono dei privilegi di oggetti a cui si fa riferimento con un descrittore.

## 83.2 Funzioni di utilità generale

Nella realizzazione di un sistema indipendente, per architettura x86-32, sono necessarie delle piccole funzioni, attraverso le quali si richiamano delle istruzioni in linguaggio assemblatore. Ciò che viene usato o che può essere usato nelle sezioni successive, viene riassunto qui.

### 83.2.1 Porte I/O

Per comunicare con i dispositivi è necessario poter leggere e scrivere attraverso delle porte di comunicazione interne. Per fare questo si usano frequentemente le istruzioni '**`INB`**' e '**`OUTB`**' del linguaggio assemblatore. Quelli che seguono sono i listati di due funzioni con lo stesso nome, per consentire di usare queste istruzioni attraverso il linguaggio C:

```
.globl  inb
#
inb:
    enter $4, $0
    pusha
    .equ inb_port,  8          # Primo parametro.
    .equ inb_data, -4          # Variabile locale.
    mov  inb_port(%ebp), %edx # Successivamente viene usato
                               # solo DX.
    inb %dx, %al
    mov  %eax, inb_data(%ebp) # Salva EAX nella variabile
                               # locale.
    popa
    mov  inb_data(%ebp), %eax # Recupera EAX che rappresenta
    leave                      # il valore restituito dalla
                               # funzione.
    ret
```

```
.globl  outb
#
outb:
    enter $0, $0
    pusha
    .equ outb_port,  8         # Primo parametro.
    .equ outb_data, 12         # Secondo parametro.
    mov  outb_port(%ebp), %edx # Successivamente viene usato
                               # solo DX.
    mov  outb_data(%ebp), %eax # Successivamente viene usato
                               # solo AL.
    outb %al, %dx
    popa
    leave
    ret
```

I prototipi delle due funzioni, da usare nel linguaggio C sono i seguenti:

```
unsigned int inb (unsigned int port);
```

```
void outb (unsigned int port, unsigned int data);
```

Il significato della sintassi è molto semplice: la funzione ***inb()*** riceve come argomento il numero di una porta e restituisce il valore, costituito da un solo byte, che da quella si può leggere; la funzione ***outb()*** riceve come argomenti il numero di una porta e il valore, rappresentato sempre solo da un byte, che a quella porta va scritto, senza restituire alcunché.

Nei prototipi si usano interi normali, invece di byte, ma poi viene considerata solo la porzione del byte meno significativo.

### 83.2.2 Sospensione e ripristino delle interruzioni hardware

Attraverso le istruzioni '**CLI**' e '**STI**' è possibile, rispettivamente, sospendere il riconoscimento delle interruzioni hardware (IRQ) e ripristinarlo. Le due funzioni seguenti si limitano a tradurre queste due istruzioni in funzioni utilizzabili con il linguaggio C:

```
.globl cli
#
cli:
    cli
    ret
```

```
.globl sti
#
sti:
    sti
    ret
```

Evidentemente, i prototipi per il linguaggio C sono semplicemente così:

```
void cli (void);
```

```
void sti (void);
```



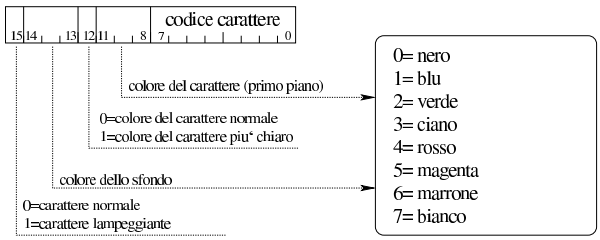
## 83.3 Utilizzo dello schermo VGA a caratteri

Per poter scrivere un programma che utilizzi autonomamente le risorse hardware, senza avvalersi di un sistema operativo, la prima cosa di cui ci si deve prendere cura è la visualizzazione di messaggi sullo schermo. Di norma si parte dal presupposto che un elaboratore x86-32 disponga di uno schermo controllato da un adattatore VGA, sul quale è possibile visualizzare del testo puro e semplice, senza dover affrontare troppe complicazioni.

### 83.3.1 Memoria dello schermo a caratteri

Per visualizzare un messaggio su uno schermo VGA, quando non è possibile usare le funzioni del BIOS, perché si sta lavorando in modalità protetta, è necessario scrivere in una porzione di memoria che parte dall'indirizzo $B8000_{16}$, utilizzando una sequenza a 16 bit, dove gli otto bit più significativi costituiscono un codice che descrive i colori da usare per il carattere e il suo sfondo, mentre il secondo contiene il carattere da visualizzare.

Figura 83.12. Organizzazione dei 16 bit con i quali si rappresenta un carattere sullo schermo.




La figura mostra come va costruito il carattere da visualizzare sullo schermo. Per esempio, un colore indicato come $28_{16}$ genera un testo di colore bianco, a intensità normale, su sfondo verde, mentre $A0_{16}$ genera un testo lampeggiante nero su sfondo verde.

### 83.3.2 Cursore

Per visualizzare del testo sullo schermo, è sufficiente assemblare i caratteri nel modo descritto sopra, collocandoli im memoria a partire dall'indirizzo $B8000_{16}$, sapendo che presumibilmente lo schermo è organizzato a righe da 80 colonne (pertanto ogni riga utilizza 160 byte e una schermata normale da 25 righe occupa complessivamente 4000 byte). Ma la visualizzazione del testo è indipendente dalla gestione del cursore e per collocarlo da qualche parte sullo schermo, occorre comunicare con l'adattatore VGA attraverso dei registri specifici.

Prima di comunicare con l'adattatore VGA per collocare il cursore, occorre definire le coordinate del cursore. Per questo occorre contare i caratteri, contando da zero. Per esempio, ammesso di voler collocare il cursore in corrispondenza della seconda colonna della ventesima riga, su uno schermo da 80 colonne per 25 righe, la posizione che si vuole raggiungere è $19 \times 80 + 2 - 1 = 1521$. Questo numero corrisponde a $05F1_{16}$.

Con l'ausilio della funzione ***outb()*** descritta in un altro capitolo, si comunica con l'adattatore VGA la posizione del cursore nel modo seguente:

```
...
    outb (0x3D4, 14);   // Prima parte.
    outb (0x3D5, 0x05);

    outb (0x3D4, 15);   // Seconda parte.
    outb (0x3D5, 0xF1);
...
```

Come si vede, l'indirizzo del cursore va dato in due fasi, dividendolo in due byte.

## 83.4 GDT

Nei microprocessori x86-32, per poter accedere alla memoria quando si sta operando in modalità protetta,[1] è indispensabile dichiarare la mappa dei segmenti di memoria attraverso una o più tabelle di descrizione. Tra queste è indispensabile la dichiarazione della tabella GDT, ovvero *global description table*, collocata nella stessa memoria centrale.[2]

La tabella GDT deve essere predisposta dal sistema operativo, prima di ogni altra cosa; di norma ciò avviene prima di far passare il microprocessore in modalità protetta, in quanto tale passaggio richiede che la tabella sia già presente per consentire al microprocessore di conoscere i permessi di accesso. Tuttavia, se si utilizza un programma per l'avvio del sistema operativo, si potrebbe trovare il microprocessore già in modalità protetta, con una tabella GDT provvisoria, predisposta in modo tale da consentire l'accesso alla memoria senza limitazioni e in modo lineare.[3] Per esempio, questo è ciò che avviene con un sistema si avvio aderente alle specifiche *multiboot*, come nel caso di GRUB 1. Ma anche così, il sistema operativo deve comunque predisporre la propria tabella GDT, rimpiazzando la precedente.

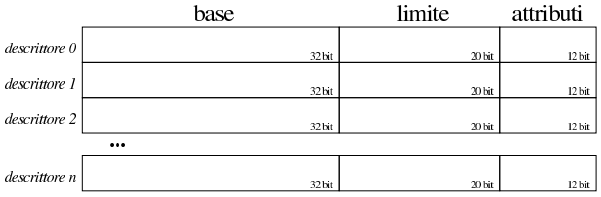
Negli esempi che appaiono nelle sezioni successive, si fa riferimento alla predisposizione di una tabella GDT, a partire da un sistema che è già in modalità protetta, essendo consentito di accedere a tutta la memoria, linearmente, senza alcuna limitazione.

### 83.4.1 Privilegi

Negli esempi che vengono mostrati, i privilegi corrispondono sempre all'anello zero, onde evitare qualunque tipo di complicazione. Tuttavia, è evidente che un sistema operativo comune deve invece gestire in modo più consapevole questo problema.

### 83.4.2 Organizzazione e contenuti della tabella GDT

Inizialmente conviene considerare la tabella GDT in modo semplificato, organizzata a righe e colonne, come si vede nello schema successivo, dal momento che nella realtà i dati sono spezzettati e sparpagliati nello spazio a disposizione. Le righe di questa tabella possono essere al massimo 8192 e ogni riga costituisce il ***descrittore*** di un segmento di memoria, il quale, eventualmente, può sovrapporsi ad altre aree.[4]

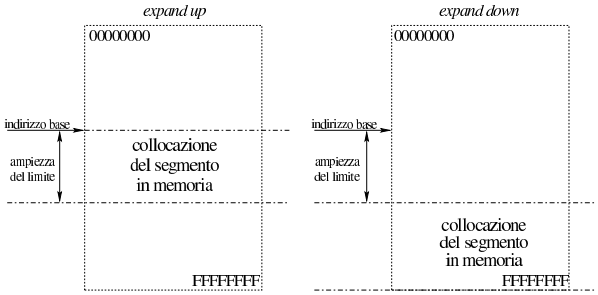



Come si vede, gli elementi dominanti delle voci che costituiscono la tabella, ovvero dei descrittori di segmento, sono la «base» e il «limite». Il primo rappresenta l'indirizzo iniziale del segmento di memoria a cui si riferisce il descrittore; il secondo rappresenta in linea di massima l'estensione di questo segmento.

Il modo corretto di interpretare il valore che rappresenta il limite dipende da due attributi: la granularità e la direzione di espansione. Come si può vedere il valore attribuibile al limite è condizionato dalla disponibilità di soli 20 bit, con i quali si può rappresentare al massimo il valore $FFFFF_{16}$, pari a $1048575_{10}$. L'attributo di granularità consente di specificare se il valore del limite riguarda byte singoli o se rappresenta multipli di 4096 byte (ovvero $1000_{16}$ byte). Evidentemente, con una granularità da 4096 byte è possibile rappresentare valori da $00000000_{16}$ a $FFFFF000_{16}$.



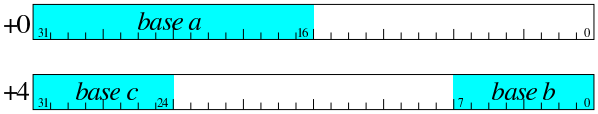
Figura 83.15. Confronto tra un limite da interpretare in modalità normale (a sinistra), rispetto a un limite da interpretare secondo un attributo di espansione in basso.



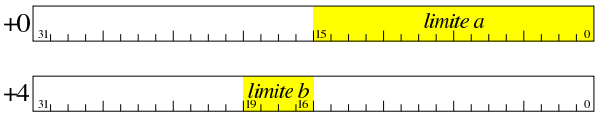

La direzione di espansione serve a determinare come si colloca l'area del segmento; si distinguono due casi: da ***base***, fino a ***base***+(***limite***×***granularità***) incluso; oppure da ***base***+(***limite***×***granularità***)+1 a $FFFFFFFF_{16}$. Il concetto è illustrato dalla figura già apparsa.

### 83.4.3 Struttura effettiva della tabella GDT

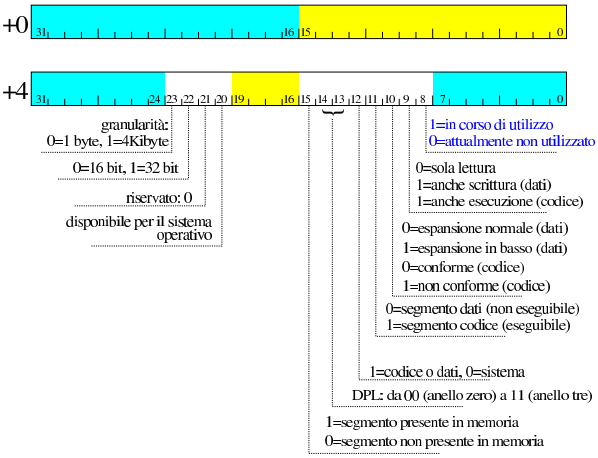
Nella realtà, la tabella GDT è formata da un array di descrittori, ognuno dei quali è composto da 8 byte, rappresentati qui in due blocchi da 32 bit, come nello schema successivo, dove viene evidenziata la porzione che riguarda l'indicazione dell'indirizzo iniziale del segmento di memoria:




L'indirizzo iniziale del segmento di memoria va ricomposto, utilizzando i bit 16-31 del primo blocco a 32 bit; quindi aggiungendo a sinistra i bit 0-7 del secondo blocco a 32 bit; infine aggiungendo a sinistra i bit 24-31 del secondo blocco a 32 bit. Anche il valore del limite del segmento di memoria risulta frammentato:




Il limite del segmento di memoria va ricomposto, utilizzando i bit 0-15 del primo blocco a 32 bit, aggiungendo a sinistra i bit 16-19 del secondo blocco a 32 bit. Nel disegno successivo si illustrano gli altri attributi, considerando che si tratti di un descrittore di memoria per codice o dati; in altri termini, il bit 12 (il tredicesimo) del secondo blocco a 32 bit **deve essere impostato a uno**:

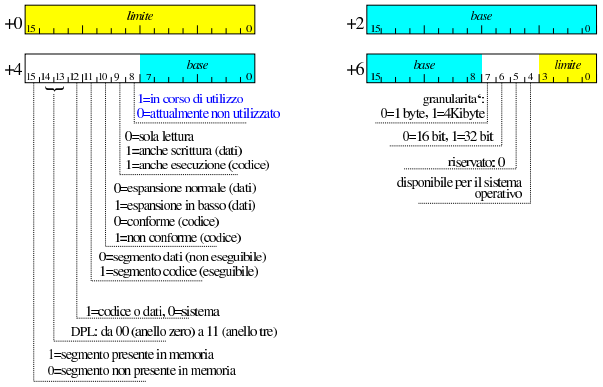



A proposito del bit che rappresenta il tipo di espansione o la conformità, in generale va usato con il valore zero, a indicare che il limite rappresenta l'espansione del segmento, a partire dalla sua ori-

gine, oppure che l'interpretazione del codice è da intendere in modo «conforme» per ciò che riguarda i privilegi. Il bit di accesso in corso (il bit numero 8, nel secondo blocco da 32 bit) viene aggiornato dal microprocessore, ma normalmente solo se all'inizio appare azzerato.

Va ricordato che i microprocessori x86-32 scambiano l'ordine dei byte in memoria. Pertanto, gli schemi mostrati sono validi solo se l'accesso alla memoria avviene a blocchi da 32 bit, perché diversamente occorrerebbe tenere conto di tali scambi. Per questa stessa ragione, il descrittore di un segmento di memoria è stato mostrato diviso in due blocchi da 32 bit, invece che in uno solo da 64, dato che l'accesso non può avvenire simultaneamente per modificare o leggere un descrittore intero.

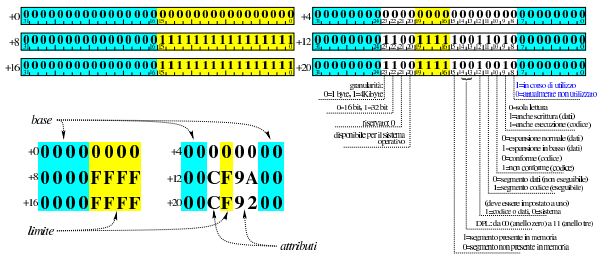
Quando si deve predisporre una tabella GDT prima di essere passati al funzionamento in modalità protetta, ovvero quando non ci si può avvalere di un sistema di avvio che offre una modalità protetta provvisoria, occorre ragionare a blocchi da 16 bit, non essendoci la possibilità di usare istruzioni a 32. Pertanto, ognuno dei blocchi descritti va invertito, come si può vedere nel disegno successivo:




### 83.4.4 Tabella GDT elementare

Una tabella GDT elementare, con la quale si voglia dichiarare tutta la memoria centrale (al massimo fino a 4 Gibyte), in modo lineare e senza distinzione di privilegi tra il codice e i dati, richiede almeno tre descrittori: un descrittore nullo iniziale, obbligatorio; un descrittore per il segmento codice che si estende su tutta la superficie della memoria; un altro descrittore identico, ma riferito ai dati. In pratica, a parte il descrittore nullo iniziale, servono almeno due descrittori, uno per il codice e l'altro per i dati, sovrapposti, entrambi attribuiti all'anello zero (quello principale). Questa è di norma la situazione che viene proposta negli esempi in cui si dimostra il funzionamento di un kernel elementare; nel disegno della figura 83.20 si può vedere sia in binario, sia in esadecimale.

Figura 83.20. Esempio di tabella GDT elementare.




### 83.4.5 Costruzione di una tabella GDT

Per costruire una tabella GDT è complicato usare una struttura per tentare di riprodurre la suddivisione degli elementi di un descrittore di segmento; pertanto, qui viene proposta una soluzione con una suddivisione che si riduce a due blocchi da 32 bit:



```
#include <stdint.h>
typedef struct {
    uint32_t  w0;
    uint32_t  w1;
} gdt_descriptor_t;
```

La funzione successiva riceve come argomento un array di descrittori di segmento, con l'indicazione dell'indice a cui si vuole fare riferimento e degli attributi che gli si vogliono associare. Va però osservato che i nomi dei parametri ***access*** e ***granularity*** rappresentano una semplificazione, nel senso che ***access*** si riferisce agli attributi che vanno dal segmento presente in memoria fino al segmento in corso di utilizzo, mentre ***granularity*** va dalla granularità fino ai bit che rappresentano un segmento riservato e disponibile:

```
#include <stdint.h>
static void
gdt_descriptor_set (gdt_descriptor_t *gdt,
                    int              descr,
                    uint32_t         base,
                    uint32_t         limit,
                    uint32_t         access,
                    uint32_t         granularity)
{
  //
  // Azzera la voce selezionata.
  //
  gdt[descr].w0 = 0;
  gdt[descr].w1 = 0;
  //
  // Trasferisce l'ampiezza del segmento (limit).
  //
  gdt[descr].w0 = gdt[descr].w0 | (limit & 0x0000FFFF);
  gdt[descr].w1 = gdt[descr].w1 | (limit & 0x000F0000);
  //
  // Trasferisce l'indirizzo iniziale del segmento (base).
  //
  gdt[descr].w0
   = gdt[descr].w0 | ((base << 16) & 0xFFFF0000);
  gdt[descr].w1
   = gdt[descr].w1 | ((base >> 16) & 0x000000FF);
  gdt[descr].w1
   = gdt[descr].w1 | (        base & 0xFF000000);
  //
  // Trasferisce gli attributi di accesso e altri attributi
  // vicini.
  //
  gdt[descr].w1
   = gdt[descr].w1 | ((access << 8) & 0x0000FF00);
  //
  // Trasferisce la granularità e altri attributi vicini.
  //
  gdt[descr].w1
   = gdt[descr].w1 | ((granularity << 20) & 0x00F00000);
}
```

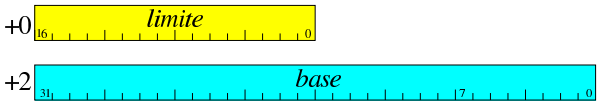
Per usare questa funzione occorre prima dichiarare l'array di descrittori di segmento. L'esempio seguente serve a riprodurre la tabella elementare della figura 83.20:

```
...
static gdt_descriptor_t gdt[3]; // Tabella GDT con tre voci.
...
gdt_descriptor_set (gdt, 0, 0,       0,    0,   0); // [1]
gdt_descriptor_set (gdt, 1, 0, 0xFFFFF, 0x9A, 0xC); // [2]
gdt_descriptor_set (gdt, 2, 0, 0xFFFFF, 0x92, 0xC); // [3]
// [1] Obbligatorio.
// [2] Codice conforme.
// [3] Dati.
...
```

Nell'esempio, l'array ***gdt[]*** viene creato specificando l'uso di memoria «statica», nell'ipotesi che ciò avvenga dentro una funzione; diversamente, può trattarsi di una variabile globale senza vincoli particolari.

### 83.4.6 Attivazione della tabella GDT

La tabella GDT può essere collocata in memoria dove si vuole (o dove si può), ma perché il microprocessore la prenda in considerazione, occorre utilizzare un'istruzione specifica con la quale si carica il registro ***GDTR*** (*GDT register*) a 48 bit. Questo registro non è visibile e si carica con l'istruzione '**LGDT**', la quale richiede l'indicazione dell'indirizzo di memoria dove si articola una struttura contenente le informazioni necessarie. Si tratta precisamente di quanto si vede nel disegno successivo:




In pratica, vengono usati i primi 16 bit per specificare la grandezza complessiva della tabella GDT e altri 32 bit per indicare l'indirizzo in cui inizia la tabella stessa. Tale indirizzo, sommato al valore specificato nel primo campo, deve dare l'indirizzo dell'ultimo byte della tabella stessa.

Dal momento che la dimensione di un descrittore della tabella GDT è di 8 byte, il valore del limite corrisponde sempre a $8\times\boldsymbol{n}-1$, dove ***n*** è la quantità di descrittori della tabella. Così facendo, si può osservare che gli ultimi tre bit del limite sono sempre impostati a uno.

Nel disegno è stato mostrato chiaramente che il primo campo da 16 bit va considerato in modo separato. Infatti, si intende che l'accesso in lettura o in scrittura vada fatto lì esattamente a 16 bit, perché diversamente i dati risulterebbero organizzati in un altro modo. Pertanto, nel disegno viene chiarito che il campo contenente l'indirizzo della tabella, inizia esattamente dopo due byte. In questo caso, con l'aiuto del linguaggio C è facile dichiarare una struttura che riproduce esattamente ciò che serve per identificare una tabella GDT:

```
#include <stdint.h>
typedef struct {
    uint16_t  limit;
    uint32_t  base;
} __attribute__ ((packed)) gdtr_t;
```

L'esempio mostrato si riferisce all'uso del compilatore GNU C, con il quale è necessario specificare l'attributo *packet*, per fare in modo che i vari componenti risultino abbinati senza spazi ulteriori di allineamento. Fortunatamente, il compilatore GNU C fa anche la cosa giusta per quanto riguarda l'accesso alla porzione di memoria a cui si riferisce la struttura.

Avendo definito la struttura, si può creare una variabile che la utilizza, tenendo conto che è sufficiente rimanga in essere solo fino a quando viene acquisita la tabella GDT relativa dal microprocessore:

```
…
    gdtr_t gdtr;
…
```

Per calcolare il valore che rappresenta la dimensione della tabella (il limite), occorre moltiplicare la dimensione di ogni voce (8 byte) per la quantità di voci, sottraendo dal risultato una unità. L'esempio presuppone che si tratti di tre voci in tutto:

```
…
    gdtr.limit = ((sizeof (gdt_descriptor_t)) * 3) - 1;
…
```

L'indirizzo in cui si trova la tabella GDT, può essere assegnato in modo intuitivo:

```
…
    gdtr.base  = (uint32_t) &gdt[0];
…
```



### 83.4.7 Verifica della tabella GDT

Le prime volte che si fanno esperimenti per ottenere l'attivazione di una tabella GDT, sarebbe il caso di verificare il contenuto di questa, prima di chiedere al microprocessore di attivarla. Infatti, un piccolo errore nel contenuto della tabella o in quello della struttura che contiene le sue coordinate, comporta generalmente un errore irreversibile. D'altra parte, proprio la complessità dell'articolazione delle voci nella tabella rende frequente il verificarsi di errori, anche multipli.

Ammesso di poter lavorare in una condizione tale da poter visualizzare qualcosa con una funzione ***printf()***, la funzione seguente consente di vedere il contenuto di una tabella GDT, partendo dall'indirizzo della struttura che rappresenta il registro ***GDTR*** da caricare, ovvero dallo stesso indirizzo che dovrebbe ricevere il microprocessore, con l'istruzione '**LGDT**':

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
static void
gdt_show (gdtr_t *gdtr)
{
  gdt_descriptor_t *gdt = (gdt_descriptor_t *) gdtr->base;
  int entries = (gdtr->limit + 1)
                / (sizeof (gdt_descriptor_t));
  int  descr;
  //
  uint32_t base;
  uint32_t limit;
  uint32_t access;
  uint32_t granularity;
  //
  printf ("gdt base: 0x%08X  limit: 0x%04X\n",
          gdtr->base, gdtr->limit);
  //
  printf ("       base       limit    "
          "access granularity\n");
  //
  for (descr = 0; descr < entries; descr++)
    {
      base = limit = access = granularity = 0;
      //
      // Indirizzo del segmento di memoria.
      //
      base = base | ((gdt[descr].w0 >> 16) & 0x0000FFFF);
      base = base | ((gdt[descr].w1 << 16) & 0x00FF0000);
      base = base | ((gdt[descr].w1      ) & 0xFF000000);
      //
      // Estensione del segmento di memoria.
      //
      limit = limit | (gdt[descr].w0 & 0x0000FFFF);
      limit = limit | (gdt[descr].w1 & 0x000F0000);
      //
      // Attributi di accesso e di tipo.
      //
      access = access | ((gdt[descr].w1 >> 8) & 0x000000FF);
      //
      // Attributi di granularità.
      //
      granularity = granularity
                    | ((gdt[descr].w1 >> 20) & 0x0000000F);
      //
      // Visualizza la voce della tabella.
      //
      printf ("gdt[%i] 0x%08" PRIX32 " 0x%06" PRIX32
              " 0x%04" PRIX32 " 0x%04" PRIX32 "\n",
              descr, base, limit, access, granularity);
  }
}
```

Stando agli esempi già fatti, si dovrebbe vedere una cosa simile al testo seguente:

```
gdt base: 0x00106044  limit: 0x0017
       base       limit    access granularity
gdt[0] 0x00000000 0x000000 0x0000 0x0000
gdt[1] 0x00000000 0x0FFFFF 0x009A 0x000C
gdt[2] 0x00000000 0x0FFFFF 0x0092 0x000C
```

Il valore $17_{16}$ corrisponde a $23_{10}$, pertanto, in questo caso, la tabella inizia all'indirizzo $00106044_{16}$ e termina all'indirizzo $0010605B_{16}$ compreso; inoltre la tabella occupa complessivamente 24 byte.

### 83.4.8 Istruzioni per l'attivazione

Per rendere operativo il contenuto della tabella GDT, va indicato al microprocessore l'indirizzo della struttura che contiene le coordinate della tabella stessa, attraverso l'istruzione '**LGDT**' (*load GDT*). Negli esempi seguenti si utilizzano istruzioni del linguaggio assemblatore, secondo la sintassi di GNU AS; in quello seguente, in particolare, si suppone che il registro ***EAX*** contenga l'indirizzo in questione:

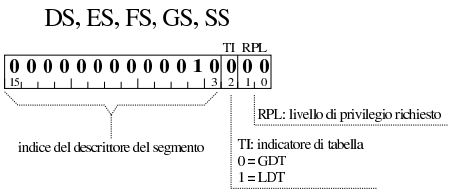
```
…
    lgdt (%eax)  # EAX contiene l'indirizzo della struttura.
…
```

A questo punto, la tabella non viene ancora utilizzata dal microprocessore e occorre sistemare il valore di alcuni registri:

```
…
    mov  $0x10, %eax    # Selettore di segmento.
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs
    mov  %ax, %gs
    mov  %ax, %ss
…
```

I registri in cui si deve intervenire sono ***DS***, ***ES***, ***FS***, ***GS*** e ***SS***, ma per assegnare loro un valore, occorre passare per la mediazione di un altro registro che in questo caso è ***AX***. Il registro ***DS*** (*data segment*) e poi tutti gli altri citati, devono avere un selettore di segmento che punti al descrittore del segmento dati attuale, con la richiesta di privilegi adeguati e la specificazione che trattasi di un riferimento a una tabella GDT. Il disegno della figura successiva mostra come va interpretato il valore dell'esempio.

Figura 83.33. Selettore del segmento dati che riguarda sia ***DS*** con gli altri registri affini per l'accesso ai dati, sia ***SS***, per la gestione della pila dei dati.



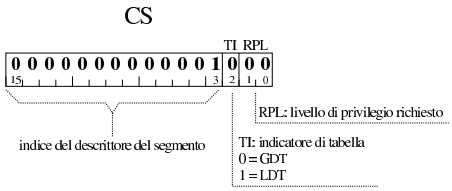

Come si può vedere nel disegno, il valore $10_{16}$ assegnato ai registri destinati ai segmenti di dati, contiene l'indice $2_{10}$ per la tabella GDT, con la richiesta di privilegi pari a zero (ovvero il valore più importante). Il descrittore con indice due della tabella GDT è esattamente quello che è stato predisposto per i dati (figura 83.20).

Subito dopo deve essere specificato il valore del registro ***CS*** (*code segment*) che in questo caso deve corrispondere a un selettore valido per il descrittore del segmento predisposto nella tabella GDT per il codice. In questo caso il valore è $08_{16}$, come si può vedere poi dalla figura successiva. Tuttavia, non è possibile assegnare il valore al registro e per ottenere il risultato, si usa un salto incondizionato a lunga distanza (*far jump*) a un simbolo rappresentato da un'etichetta che appare a poca distanza, ma con l'indicazione dell'indirizzo di segmento:

```
…
    jmp $0x08, $flush
flush:
…
```



Figura 83.35. Selettore del segmento codice.




Il listato successivo rappresenta una soluzione completa per l'attivazione della tabella GDT, a partire dall'indirizzo della struttura che ne contiene le coordinate:

```
.globl  gdt_load
#
gdt_load:
    enter $0, $0
    .equ gdtr_pointer,  8          # Primo parametro.
    mov  gdtr_pointer(%ebp), %eax  # Copia il puntatore
                                   # in EAX.
    leave
    #
    lgdt (%eax)         # Carica il registro GDTR dalla
    #                   # posizione a cui punta EAX.
    #
    mov  $0x10, %eax
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs
    mov  %ax, %gs
    mov  %ax, %ss
    jmp $0x08, $flush
flush:
    ret
```

Il codice mostrato costituisce una funzione che nel linguaggio C ha il prototipo seguente:

```
gdt_load (void *gdtr);
```

Va osservato che l'istruzione '**LEAVE**' viene usata prima di passare all'istruzione '**LGDT**'; diversamente, se si tentasse si mettere dopo l'etichetta del simbolo a cui si salta nel modo descritto (per poter impostare il registro ***CS***), l'operazione fallirebbe.

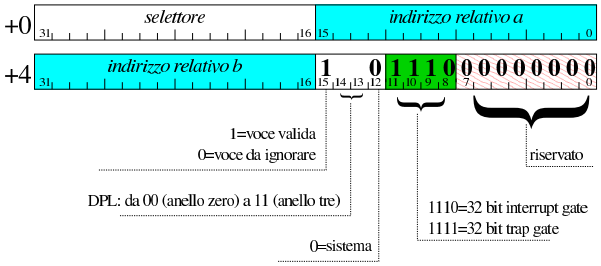
## 83.5 IDT

La tabella IDT, ovvero *interrupt descriptor table*, serve ai microprocessori x86-32 per conoscere quali procedure avviare al verificarsi delle interruzioni previste. Le interruzioni in questione possono essere dovute a eccezioni (ovvero errori rilevati dal microprocessore stesso), alla chiamata esplicita dell'istruzione che produce un'interruzione software, oppure al verificarsi di interruzioni hardware (IRQ).

Le eccezioni e gli altri tipi di interruzione, vengono associati ognuno a una propria voce nella tabella IDT. Ogni voce della tabella ha un proprio indirizzo di procedura da eseguire al verificarsi dell'interruzione di propria competenza. Tale procedura ha il nome di ISR: *interrupt service routine*.

### 83.5.1 Struttura della tabella IDT

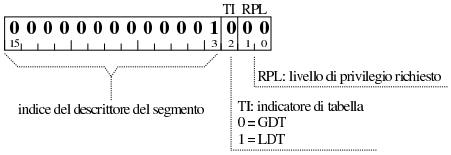
La tabella IDT è costituita da un array di descrittori di interruzione, ognuno dei quali occupa 64 bit. I descrittori possono essere al massimo 256 (da 0 a 255). Nel disegno successivo, viene mostrata la struttura di un descrittore della tabella IDT, prevedendo un accesso a blocchi da 32 bit:




La struttura contiene, in particolare, un selettore di segmento e un indirizzo relativo a tale segmento, riguardante il codice da eseguire quando si manifesta un'interruzione per cui il descrittore è competente (la procedura ISR). L'indirizzo relativo in questione è suddiviso in due parti, da ricomporre in modo abbastanza intuitivo: si prendono le due porzioni dei due blocchi a 32 bit e si uniscono senza dover fare scorrimenti.

Il selettore che si trova nei descrittori della tabella IDT ha la stessa struttura dei selettori usati direttamente con i registri per l'accesso al codice e ai dati. Per i fini degli esempi che vengono mostrati, il livello di privilegi richiesto è zero e la tabella dei descrittori di segmento a cui ci si riferisce è la GDT:

Figura 83.38. Selettore del segmento codice della procedura ISR.




In base a quanto si vede nel disegno e per gli esempi che si fanno nel capitolo, il selettore del segmento codice per le procedure ISR corrisponde a $0008_{16}$. Inoltre, negli esempi si fa riferimento esclusivamente a descrittori di tipo *interrupt gate* (a 32 bit).

### 83.5.2 Codice per la costruzione di una tabella IDT

Per costruire una tabella IDT potrebbe essere usata una struttura abbastanza ordinata; tuttavia, il tipo di descrittore e gli altri attributi non potrebbero essere suddivisi come richiederebbe il caso, pertanto qui si preferisce una struttura che si limita a riprodurre due blocchi a 32 bit, come già fatto nella sezione 83.4 a proposito della tabella GDT.

```
typedef struct {
    uint32_t  w0;
    uint32_t  w1;
} idt_descriptor_t;
```

La funzione successiva riceve come argomento un array di descrittori di una tabella IDT, con l'indicazione dell'indice a cui si vuole fare riferimento e degli attributi che gli si vogliono associare:

```
#include <stdint.h>
static void
idt_descriptor_set (idt_descriptor_t *idt,
                    int              descr,
                    uint32_t         offset,
                    uint32_t         selector,
                    uint32_t         type,
                    uint32_t         attrib)
{
    //
    // Azzera inizialmente la voce.
    //
    idt[descr].w0 = 0;
    idt[descr].w1 = 0;
    //
    // Indirizzo relativo.
    //
    idt[descr].w0 = idt[descr].w0 | (offset & 0x0000FFFF);
    idt[descr].w1 = idt[descr].w1 | (offset & 0xFFFF0000);
    //
    // Selettore di segmento.
    //
    idt[descr].w0
     = idt[descr].w0 | ((selector << 16) & 0xFFFF0000);
    //
    // Tipo (gate type).
    //
    idt[descr].w1
     = idt[descr].w1 | ((type << 8) & 0x00000F00);
    //
    // Altri attributi.
    //
    idt[descr].w1
     = idt[descr].w1 | ((type << 12) & 0x0000F000);
}
```



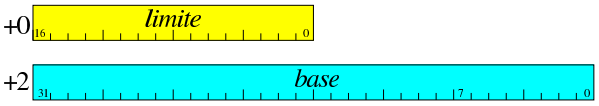
Per poter usare questa funzione occorre dichiarare prima l'array che rappresenta la tabella IDT. Di norma viene creata con tutti 256 descrittori possibili, assicurandosi che inizialmente siano azzerati effettivamente, anche se sarebbe sufficiente azzerare il bit di validità (il bit 15 del secondo blocco a 32 bit):

```
…
    static idt_descriptor_t idt[256];
…
    for (descr = 0; descr < 256; descr++)
      {
        idt_descriptor_set (idt, descr, 0, 0, 0, 0);
      }
…
```

Nell'esempio, l'array ***idt[]*** viene creato specificando l'uso di memoria «statica», nell'ipotesi che ciò avvenga dentro una funzione; diversamente, può trattarsi di una variabile globale senza vincoli particolari.

### 83.5.3 Attivazione della tabella IDT

La tabella IDT può essere collocata in memoria dove si vuole, ma perché il microprocessore la prenda in considerazione, occorre utilizzare un'istruzione specifica con la quale si carica il registro ***IDTR*** (*IDT register*) a 48 bit. Questo registro non è visibile e si carica con l'istruzione '**`LIDT`**', la quale richiede l'indicazione dell'indirizzo di memoria dove si articola una struttura contenente le informazioni necessarie. Si tratta precisamente di quanto si vede nel disegno successivo:




In pratica, vengono usati i primi 16 bit per specificare la grandezza complessiva della tabella IDT e altri 32 bit per indicare l'indirizzo in cui inizia la tabella stessa. Tale indirizzo, sommato al valore specificato nel primo campo, deve dare l'indirizzo dell'ultimo byte della tabella stessa.

Dal momento che la dimensione di un descrittore della tabella IDT è di 8 byte, il valore del limite corrisponde sempre a $8\times\boldsymbol{n}-1$, dove $\boldsymbol{n}$ è la quantità di descrittori della tabella. Così facendo, si può osservare che gli ultimi tre bit del limite sono sempre impostati a uno.

Nel disegno è stato mostrato chiaramente che il primo campo da 16 bit va considerato in modo separato. Infatti, si intende che l'accesso in lettura o in scrittura vada fatto lì esattamente a 16 bit, perché diversamente i dati risulterebbero organizzati in un altro modo. Pertanto, nel disegno viene chiarito che il campo contenente l'indirizzo della tabella, inizia esattamente dopo due byte. In questo caso, con l'aiuto del linguaggio C è facile dichiarare una struttura che riproduce esattamente ciò che serve per identificare una tabella IDT:

```
#include <stdint.h>
typedef struct {
    uint16_t  limit;
    uint32_t  base;
} __attribute__ ((packed)) idtr_t;
```

L'esempio mostrato si riferisce all'uso del compilatore GNU C, con il quale è necessario specificare l'attributo *packet*, per fare in modo che i vari componenti risultino abbinati senza spazi ulteriori di allineamento.

Avendo definito la struttura, si può creare una variabile che la utilizza, tenendo conto che è sufficiente rimanga in essere solo fino a quando viene acquisita la tabella IDT relativa dal microprocessore:

```
…
    idtr_t idtr;
…
```

Per calcolare il valore che rappresenta la dimensione della tabella, occorre moltiplicare la dimensione di ogni voce (8 byte) per la quantità di voci, sottraendo dal risultato una unità. L'esempio presuppone che si tratti di 256 voci:

```
…
    idtr.limit = ((sizeof (idt_descriptor_t)) * 256) - 1;
…
```

L'indirizzo in cui si trova la tabella IDT, può essere assegnato in modo intuitivo:

```
…
    idtr.base  = (uint32_t) &idt[0];
…
```

Per rendere operativo il contenuto della tabella IDT, quando questa è stata popolata correttamente, va indicato al microprocessore l'indirizzo della struttura che contiene le coordinate della tabella stessa, attraverso l'istruzione '**LIDT**' (*load IDT*). Negli esempi seguenti si utilizzano istruzioni del linguaggio assemblatore, secondo la sintassi di GNU AS; in quello seguente, in particolare, si suppone che il registro ***EAX*** contenga l'indirizzo in questione:

```
…
    lidt (%eax)  # EAX contiene l'indirizzo della struttura.
…
```

L'attivazione non richiede altro e non ci sono registri da modificare; pertanto, il listato seguente mostra una funzione che provvede a questo lavoro:

```
.globl  idt_load
#
idt_load:
    enter $0, $0
    .equ idtr_pointer,  8          # Primo parametro.
    mov  idtr_pointer(%ebp), %eax  # Copia il puntatore
                                   # in EAX.
    leave
    #
    lidt (%eax)   # Utilizza la tabella IDT a cui punta EAX.
    #
    ret
```

Il codice mostrato costituisce una funzione che nel linguaggio C ha il prototipo seguente:

```
idt_load (void *idtr);
```

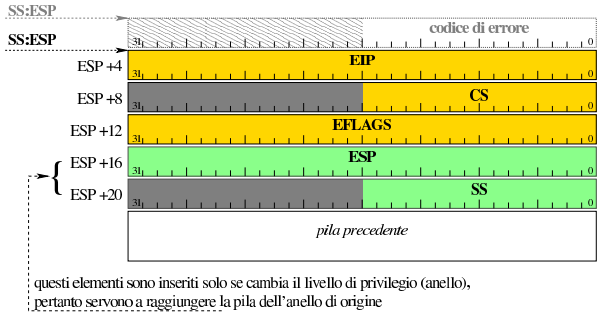
È il caso di ribadire che l'attivazione della tabella IDT va fatta solo dopo che le sue voci sono state compilate con l'indicazione delle procedure di interruzione (ISR) da eseguire.

### 83.5.4 Lo stato della pila al verificarsi di un'interruzione

Al verificarsi di un'interruzione (che coinvolge la consultazione della tabella IDT), il microprocessore accumula alcuni registri sulla pila dell'anello in cui deve essere eseguito il codice delle procedure di interruzione (ISR), come si vede nel disegno successivo, dove la pila



viene rappresentata in modo crescente dal basso verso l'alto. Va osservato che i registri ***SS*** e ***ESP*** vengono accumulati nella pila solo se i privilegi effettivi cambiano rispetto a quelli del processo da cui si proviene, perché in quel caso, al termine della procedura ISR, occorre ripristinare la pila preesistente; inoltre, quando l'interruzione è causata da un'eccezione prodotta dal microprocessore, in alcuni casi viene accumulato anche un codice di errore.




Al termine di una procedura di interruzione, per ripristinare correttamente lo stato dei registri, ovvero per riprendere l'attività sospesa, si usa l'istruzione '**IRET**'.

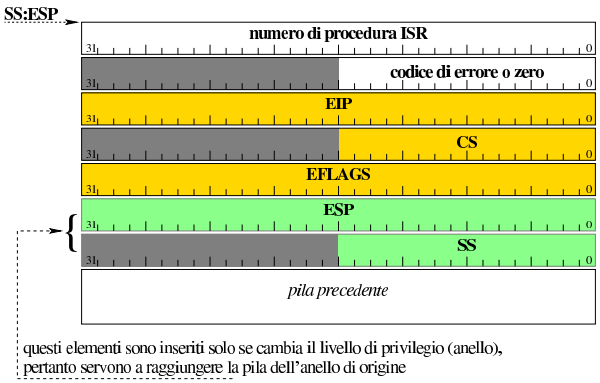
### 83.5.5 Bozza di un gestore di interruzioni

Per costruire un gestore di interruzioni è necessario predisporre un po' di codice in linguaggio assemblatore, dal quale poi è possibile chiamare altro codice scritto con un linguaggio più evoluto. Per poter gestire tutte le interruzioni in modo uniforme, occorre distinguere i casi in cui viene inserito automaticamente un codice di errore nella pila dei dati, da quelli in cui ciò non avviene; pertanto, nell'esempio viene inserito un codice nullo di errore quando non si prevede tale inserimento a cura del microprocessore, in modo da avere la stessa struttura della pila dei dati. Lo schema usato in questo listato è sostanzialmente conforme a un esempio analogo che appare nel documento *Bran's kernel development tutorial*, di Brandon Friesen, citato alla fine del capitolo.

```
.extern interrupt_handler
#
.globl isr_0
.globl isr_1
…
.globl isr_254
.globl isr_255
#
isr_0:          # division by zero exception
    cli
    push $0     # Codice di errore fittizio.
    push $0     # Numero di procedura ISR.
    jmp isr_common
#
isr_1:          # debug exception
    cli
    push $0     # Codice di errore fittizio.
    push $1     # Numero di procedura ISR.
    jmp isr_common
…
isr_8:          # double fault exception
    cli
    #
    push $8     # Numero di procedura ISR.
    jmp isr_common
…
#
isr_32:         # IRQ 0: timer
    cli
    push $0     # Codice di errore fittizio.
    push $32    # Numero di procedura ISR.
    jmp isr_common
#
isr_33:         # IRQ 1: tastiera
    cli
    push $0     # Codice di errore fittizio.
    push $1     # Numero di procedura ISR.
    jmp isr_common
```

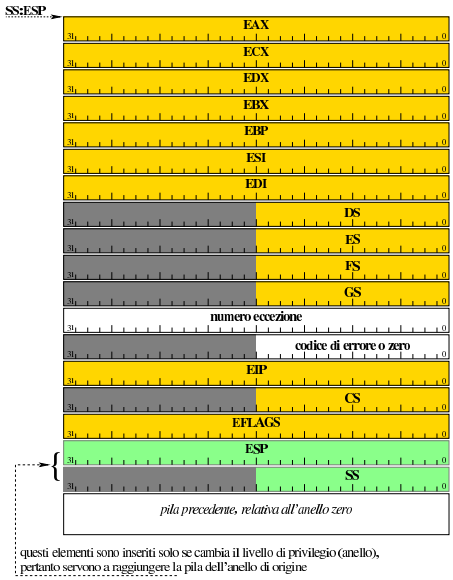
```
…
isr_47:         # IRQ 15: canale IDE secondario
    cli
    push $0     # Codice di errore fittizio.
    push $15    # Numero di procedura ISR.
    jmp isr_common
…
#
isr_common:
    pushl %gs
    pushl %fs
    pushl %es
    pushl %ds
    pushl %edi
    pushl %esi
    pushl %ebp
    pushl %ebx
    pushl %edx
    pushl %ecx
    pushl %eax
    #
    call interrupt_handler
    #
    popl %eax
    popl %ecx
    popl %edx
    popl %ebx
    popl %ebp
    popl %esi
    popl %edi
    popl %ds
    popl %es
    popl %fs
    popl %gs
    add $4, %esp     # Espelle il numero di procedura ISR.
    add $4, %esp     # Espelle il codice di errore (reale o
                     # fittizio).
    #
    iret             # ripristina EIP, CS, EFLAGS, SS
                     # e conclude la procedura.
```

Come si può vedere, quando viene chiamata una procedura che non prevede l'esistenza di un codice di errore, come nel caso di ***isr_0()***, al suo posto viene aggiunto un valore fittizio, mentre quando il codice di errore è previsto, come nel caso di ***isr_8()***, questo inserimento nella pila viene a mancare. Prima di eseguire il codice che inizia a partire da ***isr_common()***, lo stato della pila è il seguente:




Il codice che si trova a partire da ***isr_common()*** serve a preparare la chiamata di una funzione, scritta presumibilmente in C, pertanto si procede a salvare i registri; qui si includono anche quelli di segmento, per maggiore scrupolo. Al momento della chiamata, la pila ha la struttura seguente:






In base a questo contenuto della pila, una funzione scritta in C per il trattamento dell'eccezione, può avere il prototipo seguente:

```
void interrupt_handler (uint32_t eax,
                        uint32_t ecx,
                        uint32_t edx,
                        uint32_t ebx,
                        uint32_t ebp,
                        uint32_t esi,
                        uint32_t edi,
                        uint32_t ds,
                        uint32_t es,
                        uint32_t fs,
                        uint32_t gs,
                        uint32_t isr,
                        uint32_t error,
                        uint32_t eip,
                        uint32_t cs,
                        uint32_t eflags, ...);
```

I puntini di sospensione riguardano la possibilità, eventuale, di accedere anche al valori di ***ESP*** e ***SS***, quando il contesto prevede il loro accumulo.

Una volta definita in qualche modo la funzione esterna che tratta le interruzioni, le procedure ISR del file che le raccoglie (quello mostrato in linguaggio assemblatore) servono ad aggiornare la tabella IDT, la quale inizialmente è stata azzerata in modo da annullare l'effetto dei suoi descrittori. Nel listato seguente, ***idt*** è l'array di descrittori che forma la tabella IDT:

```
idt_descriptor_set (idt, 0, (uint32_t) isr_0, 0x08, 0xE, 0x8);
idt_descriptor_set (idt, 1, (uint32_t) isr_1, 0x08, 0xE, 0x8);
idt_descriptor_set (idt, 2, (uint32_t) isr_2, 0x08, 0xE, 0x8);
…
…
```

Le procedure ISR inserite nella tabella IDT devono essere solo quelle che sono operative effettivamente; per le altre è meglio lasciare i valori a zero.

### 83.5.6 Una funzione banale per il controllo delle interruzioni

Viene mostrato un esempio banale per la realizzazione della funzione ***interrupt_handler()***, a cui si fa riferimento nella sezione precedente. Si parte dal presupposto di poter utilizzare la funzione ***printf()***.

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
```

```
void
interrupt_handler (uint32_t eax, uint32_t ecx, uint32_t edx,
                   uint32_t ebx, uint32_t ebp, uint32_t esi,
                   uint32_t edi, uint32_t ds,  uint32_t es,
                   uint32_t fs,  uint32_t gs,  uint32_t isr,
                   uint32_t error, uint32_t eip,
                   uint32_t cs, uint32_t eflags, ...)
{
    printf ("ISR %3" PRIi32 ", error %08" PRIX32 "\n", isr,
            error);
}
```

### 83.5.7 Privilegi e protezioni

Negli esempi mostrati, ogni riferimento a privilegi di esecuzione e di accesso si riferisce sempre all'anello zero, pertanto non si possono creare problemi. Ma la realtà si può presentare in modo più complesso e va osservato che il livello corrente dei privilegi (CPL), nel momento in cui si verifica un'interruzione, non è prevedibile.

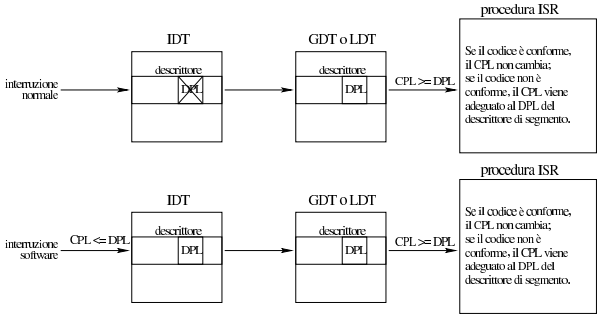
La prima cosa da considerare è il livello di privilegio del descrittore (DPL) del segmento codice in cui si trova la procedura ISR, il quale deve essere numericamente inferiore o uguale al livello corrente (CPL) precedente all'interruzione. Di conseguenza, è normale attendersi che le interruzioni comuni siano gestite da procedure ISR collocate in codice con un livello di privilegio del descrittore di segmento pari a zero.

Nel selettore del descrittore di interruzione non viene considerato il valore RPL, anche se è bene che questo sia azzerato.

Il livello di privilegio del descrittore (DPL) di interruzione viene **considerato solo in presenza di un'interruzione prodotta da software**, ovvero per un'interruzione prodotta volontariamente con le istruzioni apposite. In tal caso, il livello di privilegio corrente (CPL) del processo che la genera deve essere numericamente inferiore o uguale a quello del descrittore di interruzione. Pertanto, mettendo un valore DPL per il descrittore di interruzione pari a zero, si impedisce ai processi non privilegiati di far scattare le interruzioni in modo volontario.

Se il segmento codice dove si trova la procedura ISR è di tipo «non conforme», se il livello di privilegio corrente precedente è diverso (in questo contesto può essere solo numericamente maggiore), allora viene modificato e adeguato a quello del segmento codice raggiunto, con l'aggiunta dello scambio della pila di dati. Se invece il segmento codice dove si trova la procedura ISR è di tipo «conforme», non può avvenire alcun miglioramento di privilegi. Tra le altre cose, questa scelta ha anche delle ripercussioni per ciò che riguarda l'accesso ai dati: **il gestore di interruzione che abbia la necessità di accedere a dati che siano al di fuori della pila, deve trovarsi a funzionare all'interno di un segmento codice «non conforme», con privilegi DPL pari a zero**; diversamente (se si accontenta della pila, ovvero di variabili automatiche proprie), può funzionare semplicemente in un segmento codice conforme.

Figura 83.55. Verifica dei privilegi per l'esecuzione di una procedura ISR, a partire da un'interruzione.






## 83.6 Gestione delle interruzioni

Le interruzioni possono essere fondamentalmente di tre tipi: eccezioni prodotte dal microprocessore, interruzioni hardware (IRQ) e interruzioni prodotte attraverso istruzioni (ovvero interruzioni software). Le interruzioni vanno associate ai descrittori della tabella IDT (*interrupt descriptor table* in modo appropriato.

### 83.6.1 Eccezioni

Le ***eccezioni*** sono eventi che si manifestano in presenza di errori, di cui è competente direttamente il microprocessore. Le eccezioni sono numerate e sono già associate alla tabella IDT con gli stessi numeri: l'eccezione ***n*** è abbinata al descrittore ***n*** della tabella. Sono previste 32 eccezioni, numerate da 0 a 31, pertanto i descrittori da 0 a 31 della tabella IDT sono già impegnati per questa gestione e vanno utilizzati coerentemente in tale direzione.

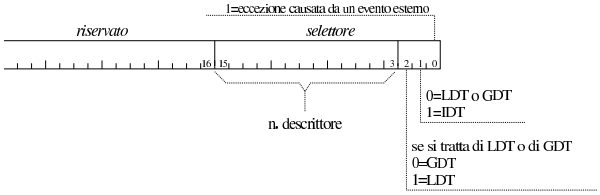
Va ricordato che in presenza di alcuni tipi di eccezione, il microprocessore accumula nella pila un codice di errore, pertanto, per uniformare le procedure ISR (*interrupt service routine*), occorre tenere conto dei casi in cui tale informazione è già inserita nella pila, rispetto a quelli dove questa non c'è ed è bene aggiungere un valore fittizio per coerenza.

Tabella 83.56. Elenco delle eccezioni.

| Eccezione | Codice di errore aggiunto sulla pila? | Definizione dell'eccezione |
|---|---|---|
| 0 | no | *division by zero* |
| 1 | no | *debug* |
| 2 | no | *non maskable interrupt* |
| 3 | no | *breakpoint* |
| 4 | no | *into detected overflow* |
| 5 | no | *out of bounds* |
| 6 | no | *invalid opcode* |
| 7 | no | *no coprocessor* |
| 8 | SÌ | *double fault* |
| 9 | no | *coprocessor segment overrun* |
| 10 | SÌ | *bad TSS* |
| 11 | SÌ | *segment not present* |
| 12 | SÌ | *stack fault* |
| 13 | SÌ | *general protection fault* |
| 14 | SÌ | *page fault* |
| 15 | no | *unknown interrupt* |
| 16 | no | *coprocessor fault* |
| 17 | no | *alignment check exception* |
| 18 | no | *machine check exception* |
| da 19 a 31 | no | eccezioni riservate per il futuro |

Il codice di errore che inserisce il microprocessore sulla pila, quando si verificano le eccezioni che lo prevedono, ha una struttura variabile, in base al tipo di eccezione. Lo schema della figura successiva è abbastanza comune e riguarda un errore per il quale viene fatto riferimento a un selettore (per la tabella GDT, LDT o IDT, in base al contesto).

Figura 83.57. Codice di errore prodotto da alcune eccezioni.




Come si può intendere dal disegno, a seconda dei valori dei bit 1 e 2, il selettore va inteso riguardare una voce della tabella GDT, oppure di una tabella LDT o della tabella IDT stessa.

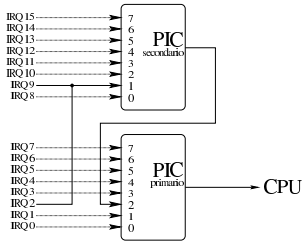
Quando il codice di errore è completamente a zero, almeno nei primi 16 bit meno significativi, vuol dire che non riguarda un problema

collegabile a una voce di una delle tabelle IDT, LDT o GDT.

### 83.6.2 PIC e rimappatura delle interruzioni

Per affrontare al gestione delle interruzioni hardware, occorre prima premettere una breve introduzione, a causa del fatto che non si tratta di una funzione gestita autonomamente dal microprocessore.

Secondo la tradizione dell'architettura IBM PC/AT, per raccogliere le interruzioni hardware dell'elaboratore sono utilizzati due integrati, chiamati generalmente PIC, ovvero *programmable interrupt controller*, collegati assieme in modo da poter recepire complessivamente quindici interruzioni hardware differenti. Per la precisione, il PIC secondario, se riceve un'interruzione, va a provocare un IRQ 2 nel PIC primario; pertanto, se si ricevono interruzioni tra IRQ 8 e IRQ 15, si ottiene anche un'interruzione su IRQ 2. Dal momento che IRQ 2 è impegnato, quello che sarebbe il segnale di IRQ 2 viene ridiretto a IRQ 9. Il disegno seguente serve solo a chiarire il concetto, dal momento che i collegamenti effettivi sono più complessi:




Le interruzioni hardware, o «IRQ», vanno abbinate a interruzioni della tabella IDT, per poterle gestire in qualche modo. Purtroppo, originariamente esiste già un abbinamento, ma incompatibile con quello delle eccezioni del microprocessore; pertanto, va rifatta la mappa di trasformazione.

Per comunicare con i due PIC e per riprogrammarli, esistono delle porte di comunicazione: $20_{16}$ e $21_{16}$ per il PIC principale; $A0_{16}$ e $A1_{16}$ per il PIC secondario. La procedura per rimappare i PIC richiede la scrittura di diversi valori che, a seconda dei casi, prendono il nome di «ICW» (*initialization command word*) e «OCW» (*operation command word*). La funzione seguente, scritta in linguaggio C, permette la rimappatura dei due PIC e abilita automaticamente tutte le interruzioni hardware (che altrimenti potrebbero anche essere mascherate).

```
#include <stdio.h>
void
irq_remap (unsigned int offset_1, unsigned int offset_2)
{
  //
  // PIC_P è il PIC primario o «master»;
  // PIC_S è il PIC secondario o «slave».
  //
  // Quando si manifesta un IRQ che riguarda il PIC
  // secondario, il PIC primario riceve IRQ 2.
  //
  // ICW = initialization command word.
  // OCW = operation command word.
  //
  printf ("kernel: PIC "
          "(programmable interrupt controller) remap: ");

  outb (0x20, 0x10 + 0x01); // Inizializzazione: 0x10
  outb (0xA0, 0x10 + 0x01); // significa che si tratta di
  printf ("ICW1");          // ICW1; 0x01 significa che si
                            // deve arrivare fino a ICW4.

  outb (0x21, offset_1);    // ICW2: PIC_P a partire da
  outb (0xA1, offset_2);    // «offset_1»; PIC_S a partire
  printf (", ICW2");        // da «offset_2».

  outb (0x21, 0x04);  // ICW3 PIC_P: IRQ2 pilotato da PIC_S.
  outb (0xA1, 0x02);  // ICW3 PIC_S: pilota IRQ2 di PIC_P.
  printf (", ICW3");
  outb (0x21, 0x01);  // ICW4: si precisa solo la modalità
  outb (0xA1, 0x01);  // del microprocessore; 0x01 = 8086.
  printf (", ICW4");
  outb (0x21, 0x00);    // OCW1: azzera la maschera in modo
  outb (0xA1, 0x00);    // da abilitare tutti i numeri IRQ.
  printf (", OCW1.\n");
}
```



Nel corso del procedimento di rimappatura delle interruzioni, è necessario fare delle brevissime pause, per dare il tempo ai PIC di recepire le informazioni; a tale proposito sono state aggiunge delle istruzioni che visualizzano il progresso nelle varie fasi di rimappatura. Le sigle che appaiono nei commenti del listato, richiamano i termini usati per identificare i valori che sono attribuiti alle porte, in modo da poter ritrovare nella documentazione dei PIC il significato che hanno.

La funzione proposta nell'esempio riceve due argomenti, corrispondenti allo spostamento delle interruzioni del primo e del secondo PIC. Per esempio, ammesso di voler spostare le interruzioni del primo PIC a partire da $32_{10}$ e quelle del secondo PIC a partire da $40_{10}$, in modo da utilizzare esattamente le voci della tabella IDT successive a quelle delle eccezioni, basta usare la funzione nel modo seguente:

```
…
    irq_remap (32, 40);
…
```

### 83.6.3 Procedura generalizzata per la gestione delle interruzioni

Nella sezione 83.5.5 appare il codice iniziale, in linguaggio assemblatore, per la gestione delle interruzioni. A partire da lì viene richiamata la funzione ***interrupt_handler()***, dalla quale è possibile risalire al numero di procedura ISR da attivare. Per rendere intercambiabili le funzioni che gestiscono specificatamente ogni singola interruzione, potrebbe essere conveniente predisporre un array di puntatori a funzione, ma per comodità viene dichiarato semplicemente come array di puntatori generici, inizialmente azzerati:

```
…
void *isr_func[256] = {0};
…
```

Le funzioni che si associano agli elementi dell'array devono essere tali da poter gestire l'interruzione di propria competenza. Per esempio, ***isr_func[0]*** deve essere il puntatore di una funzione in grado di gestire l'interruzione derivante dall'eccezione *divide error*.

Ammesso di avere popolato correttamente l'array ***isr_func[]***, la funzione ***interrupt_handler()*** potrebbe essere fatta così:

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>
void
interrupt_handler (uint32_t eax, uint32_t ecx, uint32_t edx,
                   uint32_t ebx, uint32_t ebp, uint32_t esi,
                   uint32_t edi, uint32_t ds,  uint32_t es,
                   uint32_t fs,  uint32_t gs,  uint32_t isr,
                   uint32_t error, uint32_t eip,
                   uint32_t cs, uint32_t eflags, ...)
{
  if (isr > 255)
    {
      printf ("kernel: %s: error: cannot handle "
              "ISR %" PRIi32 "!\n",
              __func__, isr);
      return;
    }

  //
  // La variabile handler è un puntatore a funzione che ha
  // due parametri di tipo «unsigned int» a 32 bit e
  // restituisce «void».
  //
  void (*handler) (uint32_t isr, uint32_t error);
  //
  // Carica la funzione associata al numero ISR.
  //
  handler = isr_func[isr];
```

```
  //
  // Se il puntatore a funzione è diverso da NULL, allora
  // procede.
  //
  if (handler)
    {
      handler (isr, error);
    }
  //
  // Se si tratta di un'interruzione hardware, occorre
  // informare i PIC coinvolti che l'elaborazione è
  // terminata, attraverso un messaggio «EOI».
  //
  if (isr >= 40 && isr <= 47)
    {
      // PIC secondario.
      outb (0xA0, 0x20);
    }
  //
  if (isr >= 32 && isr <= 47)
    {
      // Il PIC primario è coinvolto sempre.
      outb (0x20, 0x20);
    }
}
```

Come si vede, per semplificare il tutto, le funzioni che devono elaborare le interruzioni devono avere un prototipo di questo tipo:

```
#include <stdint.h>
void nome_funzione (uint32_t isr, uint32_t error);
```

Una funzione generica, anche se poco graziosa, per il trattamento delle eccezioni potrebbe essere fatta così:

```
#include <inttypes.h>
#include <stdio.h>
void
exception_handler (uint32_t isr, uint32_t error)
{
  printf ("kernel: exception %" PRIi32 ", "
          "error %04" PRIX32 "!\n",
          isr, error);
  //
  // Blocca tutto.
  //
  for (;;);
}
```

Per associare la funzione alle prime 32 voci dell'array ***isr_func()***, si potrebbe procedere così:

```
…
int i;
…
for (i = 0; i < 256; i++)
  {
    isr_func[i] = exception_handler;
  }
…
```

Per quanto riguarda le funzioni che devono gestire le interruzioni di origine hardware, bisogna ricordare che il valore del parametro ***isr*** non dà il numero IRQ, ma se fosse necessario calcolarlo basterebbe sottrarre il numero 32 da quello del numero della voce ISR originale.

### 83.6.4 Attivazione

In precedenza è stato mostrato come si attiva la tabella IDT, attraverso l'istruzione '**LIDT**', ma è evidente che questo va fatto solo dopo che la tabella IDT è stata predisposta e che sono state preparate le funzioni per la gestione delle interruzioni (quelle che si vogliono gestire). Ciò che rimane, ammesso di essere pronti a gestire le interruzioni hardware, è l'attivazione di queste interruzioni, con l'istruzione '**STI**' del linguaggio assemblatore.



## 83.7 Gestione del temporizzatore: PIT, ovvero «programmable interval timer»

Negli elaboratori con architettura IBM PC/AT, è previsto un temporizzatore costituito originariamente da un integrato programmabile, contenente tre contatori: uno associato a IRQ 0, uno associato a qualche funzione particolare, dipendente dall'organizzazione dell'hardware, un altro associato all'altoparlante interno. Questo integrato è noto con la sigla PIT, ovvero *programmable interval timer*.

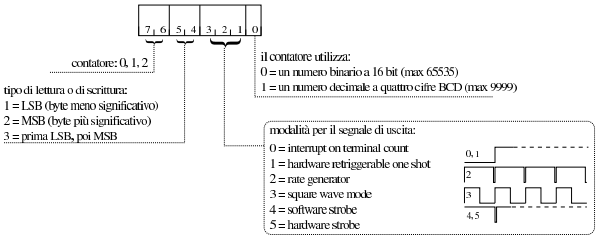
Questo integrato, o comunque ciò che ne fa la funzione, conta degli impulsi provenienti a una frequenza stabilita e, a seconda di come viene programmato, produce un risultato differente nelle sue tre uscite. Per esempio può generare un'onda quadra a una frazione della frequenza ricevuta in ingresso, oppure può emettere altri tipi di segnali, sempre tenendo in considerazione il risultato del conteggio degli impulsi in ingresso.

Per quanto riguarda la gestione del temporizzatore, ovvero della frequenza con cui si vuole ottenere un'interruzione IRQ 0, generalmente si programma il PIT per produrre un'onda quadra.

Secondo lo standard dell'architettura IBM PC/AT, la frequenza che produce gli impulsi in ingresso del PIT è a 1,19 MHz circa. Più precisamente si tratta di 3579545/3 Hz.

La programmazione del PIT avviene inviando un comando (*command word*, o CW), costituito da un byte, alla porta $43_{16}$, con il quale, in particolare, si specifica il contatore a cui ci si vuole riferire. Successivamente, a seconda del comando inviato, possono essere trasmessi altri valori alla porta riservata specificatamente per il contatore a cui si è interessati. Il contatore zero che serve a produrre le interruzioni IRQ 0, riceve questi valori dalla porta $40_{16}$, mentre la porta $42_{16}$ è quella del contatore tre, associato all'altoparlante interno (il contatore uno sarebbe associato alla porta $41_{16}$, ma in pratica non può essere utilizzato).

Figura 83.65. Comando da inviare alla porta $43_{16}$.




La figura appena apparsa schematizza in che modo va composto o interpretato il comando da inviare al PIT. Per quanto riguarda la modalità di funzionamento, quella che serve per generare le interruzioni è la numero 3 (onda quadra); per conoscere il significato delle altre modalità si possono consultare i documenti citati alla fine del capitolo. Il resto delle componenti di un comando dovrebbe essere abbastanza comprensibile, ma vale la pena di riassumere brevemente. I primi due bit più significativi indicano il contatore a cui si vuole fare riferimento. Altri due bit indicano cosa deve essere trasmesso, successivamente al comando, attraverso la porta dei dati: un solo byte, a scelta tra il meno significativo o il più significativo, oppure entrambi i byte, a cominciare da quello meno significativo. Altri tre bit definiscono la modalità. Per quanto riguarda il senso del bit meno significativo, occorre considerare che il contatore degli impulsi ricevuti in ingresso può utilizzare un valore a 16 bit (cosa che si fa normalmente), oppure un numero a sole quattro cifre in base dieci (i 16 bit del contatore verrebbero divisi in quattro gruppi da quattro bit, ognuno dei quali viene usato esclusivamente per rappresentare valori da zero a nove).

Per programmare il contatore zero, in modo che generi una certa frequenza (purché inferiore a 1,19 MHz), si usa normalmente il comando $36_{16}$, il quale: seleziona il contatore zero; stabilisce che il valore da comunicare successivamente viene trasmesso usando due

byte (prima quello meno significativo, poi quello più significativo); richiede una modalità di funzionamento a onda quadra; richiede di utilizzare il contatore in modo binario, a 16 bit. Successivamente al comando si usa il valore che rappresenta il divisore della frequenza di 1,19 MHz. Per esempio, volendo generare una frequenza vicina a 100 Hz, dopo aver inviato il comando $36_{16}$ alla porta $43_{16}$, occorre inviare il valore $11931_{10}$, separandolo in due byte, alla porta $40_{16}$.

Va osservato che il valore del divisore può utilizzare al massimo 16 bit complessivamente, partendo da uno (lo zero non è ammissibile per ovvi motivi). Pertanto, si può dividere la frequenza di ingresso al massimo di 65535 volte.

Segue l'esempio di una funzione con la quale si programma la frequenza delle interruzioni IRQ 0, ma senza verificare che il valore richiesto sia valido:

```
void
timer_freq (int freq)
{
    int input_freq = 1193181;
    int divisor = input_freq / freq;
    outb (0x43, 0x36);             // CW: «command word».
    outb (0x40, divisor & 0x0F);   // LSB: byte inferiore del
                                   // divisore.
    outb (0x40, divisor / 0x10);   // MSB: byte superiore del
                                   // divisore.
}
```

Se il PIT non viene riprogrammato, inizialmente lo si trova configurato in modo da generare una frequenza (a onda quadra) di 18,222 Hz che è quella più bassa possibile.

## 83.8 Tastiera PS/2

La tastiera PS/2 di un elaboratore IBM PC/AT produce un'interruzione ogni volta che si preme o si rilascia un tasto, quindi si può leggere tale codice dalla porta $60_{16}$. Per la precisione, dalla porta $60_{16}$ si può leggere un solo byte alla volta, mentre ci sono situazioni in cui i codici generati dalla pressione o dal rilascio dei tasti sono formati da una sequenza di più byte; pertanto, la tastiera possiede una propria memoria tampone, dalla quale si può leggere sequenzialmente.

Il funzionamento della tastiera può essere configurato, inviando, a porte differenti, dei comandi che qui non vengono trattati; tuttavia la documentazione annotata nella bibliografia riporta tali informazioni.

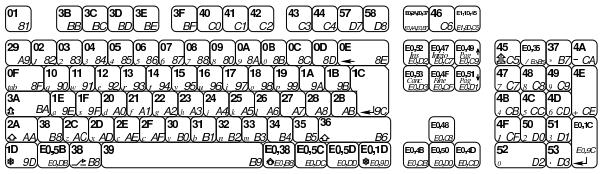
Il codice che si può leggere attraverso la porta $60_{16}$ è definito *scancode*, ma ne esistono normalmente tre versioni, di cui quella standard (predefinita) è la seconda. Per conoscere i codici generati dalla tastiera si può utilizzare il programma '**showkey**', con l'opzione '**-s**', da un sistema GNU/Linux. Con l'aiuto di questo programma si può anche comprendere bene come vengano generati i codici e l'effetto della ripetizione automatica.

Come regola generale va osservato che i tasti premuti producono un codice inferiore o uguale a $127_{10}$ (ovvero $7F_{16}$, oppure $1111111_2$), mentre i tasti rilasciati producono il valore corrispondente alla somma del codice di pressione più $128_{10}$ (ovvero $80_{16}$, oppure $1000000_2$). In pratica, si riconosce il rilascio di un tasto per il fatto che il bit più significativo è impostato a uno.

Le sequenze multiple di alcuni tasti servono normalmente a distinguerli rispetto ad altri equivalenti, inserendo normalmente il codice $E0_{16}$. Per esempio, il tasto [*Ctrl*] sinistro produce il codice $1D_{16}$ alla pressione e $9D_{16}$ al rilascio, mentre il tasto [*Ctrl*] destro produce $E0_{16}$ $1D_{16}$ alla pressione e $E0_{16}$ $9D_{16}$ al rilascio. Pertanto se si vuole semplificare l'interpretazione dei tasti premuti dalla tastiera, si potrebbero ignorare i codici speciali che servono per le sequenze multiple.



Figura 83.67. Mappa dei codici della tastiera. Sulla parte superiore sinistra appare la sequenza generata dalla pressione del tasto, mentre sulla parte inferiore destra appare quella associata al rilascio del tasto. Le sequenze sono espresse in esadecimale.



## 83.9 Gestione di dischi PATA

Qui si descrive come accedere a unità a disco ATA (*AT attachment*) con cavo parallelo (PATA), da un elaboratore con architettura conforme al IBM PC, secondo la modalità PIO (*Programmed input-output*), con la quale si impegna direttamente la CPU per il trasferimento dei dati. La modalità di trasferimento PIO è logicamente quella più costosa per il sistema, ma è anche la più semplice da ottenere in termini di programmazione. Non si considerano unità ATAPI (*AT attachment packet interface*) e comunque ci si sofferma prevalentemente sulla modalità di accesso LBA28. (si veda eventualmente anche la sezione 9.3 sulle unità PATA.)

Le unità a disco PATA, si connettono attraverso un cavo piatto (piattina) a un bus; su tale cavo si possono collegare due dischi: uno dei due viene chiamato *master* e l'altro *slave*. In pratica, la distinzione delle unità ATA attraverso queste denominazioni non è appropriata, perché non esistono ruoli sostanzialmente differenti; piuttosto si tratta solo di distinguere le due unità. È il caso di ricordare che la selezione tra prima e seconda unità può avvenire attraverso una configurazione precisa, di solito con l'ausilio di ponticelli, oppure automatica (*cable select*), in cui la posizione del connettore nel cavo decide il numero dell'unità.

Di norma, le schede madri degli elaboratori conformi all'architettura IBM PC dispongono di due bus ATA, con cui si possono connettere complessivamente un massimo di quattro unità.

I comandi che si danno alle unità ATA comportano la scrittura e la lettura di «registri» interni alla gestione dei bus. Per accedere a tali registri, nell'architettura conforme al IBM PC, si usano delle porte di comunicazione: leggendo o scrivendo una certa porta, si ottiene la lettura o la scrittura di un certo registro del sistema ATA.

I registri sono sempre associati a un certo bus, sul quale però possono essere connessi due dispositivi. A seconda del contesto, i comandi che si impartiscono possono riguardare il bus in generale, o un dispositivo preciso, individuato dai parametri del comando.

Dal momento che le unità di memorizzazione di massa non possono essere sufficientemente veloci nelle loro reazioni, rispetto alle possibilità della CPU, ci sono alcune operazioni che richiedono un tempo di attesa prima di poter leggere un esito corretto o prima di poter proseguire con altri comandi. Nel documento *ATA PIO mode*, *http://wiki.osdev.org/ATA_PIO_Mode* , citato anche in fondo al capitolo, si menziona in certi casi un ritardo di sicurezza di 400 ns, necessario soprattutto per le unità più vecchie. In quel documento si fa riferimento alla possibilità di eseguire per cinque volte lo stesso comando, prima di poter ottenere un esito valido, ma questa procedura riguarda la programmazione in linguaggio assemblatore, perché se si utilizza il C, o altro linguaggio più evoluto, può darsi che non ci sia la stessa efficienza e bastino meno tentativi.

Quando si eseguono operazioni di scrittura, occorre chiedere espressamente lo scarico della memoria trattenuta (*cache*), per ottenere la memorizzazione effettiva nel disco. Se non si ha l'accortezza di procedere in tal modo, si rischia di fare fallire l'operazione di scrittura successiva.

Anche le unità ATA, come tutti i sistemi di memorizzazione di massa a disco, possono trovarsi ad avere dei settori danneggiati e inuti-

lizzabili. Nel caso delle unità ATA di distingue però tra settori che non possono essere letti o scritti in modo permanente e settori che invece non possono essere letti, ma solo temporaneamente. Questa impossibilità temporanea di lettura può derivare da una fase di scrittura incompleta: tali settori tornano a essere leggibili correttamente quando si esegue su di loro una nuova operazione di scrittura che giunge a termine correttamente.

### 83.9.1 Modalità di accesso

L'accesso ai settori delle unità ATA può avvenire secondo tre modalità: CHS, LBA28 e LBA48. La modalità CHS rappresenta il metodo più vecchio per individuare un settore in un disco, in quanto occorre specificare le coordinate composte da cilindro, testina e settore di questa combinazione. L'accesso in modalità CHS (*Cylinder Head Sector*) riguarda concretamente solo le unità a disco degli anni 1980, perché successivamente le unità ATA hanno introdotto la possibilità di raggiungere i settori attraverso un numero sequenziale, senza dover conoscere la geometria effettiva del disco.

Per problemi di compatibilità, è rimasta la facoltà di individuare i settori attraverso coordinate CHS, le quali di norma si riferiscono a una geometria astratta e non reale. Infatti, in condizioni normali, ci possono essere unità a disco composte da una quantità limitatissima di testine (due, quattro, sei), mentre programmi come '`fdisk`' riportano spesso una quantità fantastica di 255 testine. Tutto ciò deriva dai limiti del BIOS (*firmware*) degli elaboratori conformi all'architettura IBM PC.

Ammesso di avere determinato o definito una certa geometria, si convertono le coordinate CHS in numero assoluto del settore con delle formule. Si considerano le variabili seguenti:

| Variabile | Descrizione |
|---|---|
| ***C*** | Quantità totale dei cilindri. |
| ***H*** | Quantità totale delle testine. |
| ***S*** | Quantità totale di settori per traccia (ogni cilindro ha ***H*** tracce). |
| ***c*** | Cilindro di una coordinata, dove il primo è zero e l'ultimo è ***C***−1. |
| ***h*** | Testina di una coordinata, dove la prima è zero e l'ultima è ***H***−1. |
| ***s*** | Settore di una traccia, dove il primo è 1 e l'ultimo è ***S***. Che i settori di una traccia inizino da uno è un fatto importante, a cui è necessario prestare attenzione. |
| ***z*** | Numero assoluto del settore, dove il primo è pari a zero. |

Il numero assoluto di un settore, conoscendo la sua coordinata CHS, si ottiene come:

$$z = c \cdot H \cdot S + h \cdot S + s - 1$$

Partendo invece dal numero assoluto del settore, per determinare le sue coordinate virtuali, valgono le formule seguenti. Si osservi che dalle divisioni si prendono solo i risultati interi.

$$c = \frac{z}{H \cdot S}$$

$$h = \frac{z + 1 - c \cdot H \cdot S}{S}$$

$$s = z + 1 - c \cdot H \cdot S - h \cdot S$$

Quando si accede alle unità ATA specificando il numero assoluto del settore, si può usare la modalità LBA28, che permette di raggiungere al massimo il settore $FFFFFFF_{16}$, oppure, ammesso che il dispositivo lo consenta, la modalità LBA48, che permette di raggiungere al massimo il settore $FFFFFFFFFFF_{16}$. In pratica, con la modalità LBA28, sapendo che i settori sono da 512 byte, si possono gestire dispositivi con una capacità massima di 128 Gibyte, mentre con la modalità LBA48 si può arrivare fino a 128 Pibyte ($128 \cdot 2^{50}$).



La modalità di accesso CHS è superata da molto tempo. Tuttavia, se la si deve usare, va ricordato che, in tal caso, non può esistere il settore zero.

### 83.9.2 Registri per la gestione delle unità ATA

Nella tabella successiva, si riepilogano i registri utili per la gestione delle unità ATA, secondo la modalità PIO. Nella tabella si omette ***data port***, in quanto si riferisce soltanto alla modalità DMA per il trasferimento dei dati. Il registro che nella tabella è chiamato ***data*** è diverso dal ***data port*** e riguarda il trasferimento in modalità PIO. I registri della tabella sono generalmente da un solo byte (8 bit), a eccezione di ***data*** il quale normalmente va letto e scritto a coppie di byte (16 bit).

I registri possono consentire la lettura (r), la scrittura (w) o entrambe le cose. Quando un registro è a senso unico (sola lettura o sola scrittura), vuol dire che l'accesso in senso opposto è relativo a informazioni differenti, a cui si associa un altro nome. Per esempio, quello che viene chiamato ***regular status*** è un registro in sola lettura, ma se vi si accede ugualmente in scrittura, si interviene in pratica nel registro ***device control***, il quale è invece in sola scrittura (naturalmente lo stesso vale in senso inverso). Evidentemente l'attribuzione di nomi differenti ai registri, a seconda della direzione di accesso, consente di evitare facili confusioni.

Lo stato di funzionamento del dispositivo corrente di un certo bus è riportato in modo uguale da due registri: ***regular status*** e ***alternate status***. La distinzione tra questi sta nel fatto che la lettura del primo comporta la «acquisizione» dell'informazione, mentre la lettura del secondo non altera in alcun modo la situazione del dispositivo. Lo stato del dispositivo è costituito da indicatori, ovvero bit che se sono a uno rappresentano l'avverarsi di una certa condizione. A questi indicatori si fa riferimento con un nome. I più importanti sono BSY (*busy*), DF (*drive fault*), DRQ (*data request*) e ERR (*error*).

Tabella 83.73. Registri ATA utili per la modalità PIO.

| Definizione | Dir (r/w) | $Bus_0$ | $Bus_1$ | $Bus_2$ | $Bus_3$ | Descrizione |
|---|---|---|---|---|---|---|
| *data* | r/w | $1F0_{16}$ | $170_{16}$ | $1E8_{16}$ | $168_{16}$ | Consente di leggere o scrivere il dispositivo selezionato precedentemente con il registro ***device***, a coppie di byte, ma prima di poterlo fare è necessario che l'indicatore DRQ sia attivo. La lettura o la scrittura è progressiva, ma riguarda sempre uno o più settori interi, pertanto va ripetuta a multipli di 256 volte. Non è possibile mescolare letture e scritture. |
| *error* | **r**/- | $1F1_{16}$ | $171_{16}$ | $1E9_{16}$ | $169_{16}$ | Questo registro, in sola lettura, descrive il tipo di errore manifestato dall'indicatore ERR. In pratica, il contenuto di questo registro è privo di significato se l'indicatore ERR non fosse attivo. Inoltre, per poter prendere in considerazione l'indicatore ERR, è opportuno che l'indicatore BSY sia a zero. L'interpretazione del registro cambia a seconda del tipo di comando che lo ha causato. L'accesso in scrittura a questo registro porta a intervenire invece nel registro ***feature***. |
| *feature* | -/**w** | $1F1_{16}$ | $171_{16}$ | $1E9_{16}$ | $169_{16}$ | Registro in sola scrittura, per l'indicazione di un parametro, il cui significato cambia a seconda del comando che si impartisce successivamente. La scrittura in questo registro deve essere subordinata al fatto che gli indicatori BSY e DRQ siano a zero. L'accesso in lettura a questo registro porta a intervenire invece nel registro ***error***. |

| Definizione | Dir (r/w) | $Bus_0$ | $Bus_1$ | $Bus_2$ | $Bus_3$ | Descrizione |
|---|---|---|---|---|---|---|
| *count* (*sector count*) | r/w | $1F2_{16}$ | $172_{16}$ | $1EA_{16}$ | $16A_{16}$ | Si usa come parametro di un comando successivo, per indicare una quantità di settori da prendere in considerazione, tenendo conto che il valore zero ha un significato particolare. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero. |
| *low* (*lba low*) | r/w | $1F3_{16}$ | $173_{16}$ | $1EB_{16}$ | $16B_{16}$ | Si usa come parametro di un comando successivo, per indicare l'intervallo $bit_0$..$bit_7$ dell'indirizzo del settore da raggiungere. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero. |
| *mid* (*lba mid*) | r/w | $1F4_{16}$ | $174_{16}$ | $1EC_{16}$ | $16C_{16}$ | Si usa come parametro di un comando successivo, per indicare l'intervallo $bit_8$..$bit_{15}$ dell'indirizzo del settore da raggiungere. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero. |
| *high* (*lba high*) | r/w | $1F5_{16}$ | $175_{16}$ | $1ED_{16}$ | $16D_{16}$ | Si usa come parametro di un comando successivo, per indicare l'intervallo $bit_{16}$..$bit_{23}$ dell'indirizzo del settore da raggiungere. Si può modificare il registro solo se gli indicatori BSY e DRQ sono a zero; allo stesso modo la lettura del registro è valida solo se questi indicatori sono a zero. |
| *device* | r/w | $1F6_{16}$ | $176_{16}$ | $1EE_{16}$ | $16E_{16}$ | È un registro al cui interno si inseriscono diverse informazioni, il cui significato può cambiare a seconda del comando che si impartisce successivamente. Tuttavia, il $bit_4$ (il quinto), rappresenta il dispositivo nel bus a cui si fa riferimento (zero è il primo, uno è il secondo). Non va confuso questo registro con il ***device control***. |
| *regular status* | **r**/- | $1F7_{16}$ | $177_{16}$ | $1EF_{16}$ | $16F_{16}$ | Dà lo stato del bus, con una lettura che comporta l'acquisizione di tale informazione. Quando è attivo l'indicatore BSY, tutti gli altri indicatori sono privi di significato. L'interpretazione del dato è equivalente a quella che si deve fare per il registro ***alternate status***. Questo registro è in sola lettura, in quanto, se vi si accede in scrittura, si interviene invece nel registro ***command***. |
| *command* | -/**w** | $1F7_{16}$ | $177_{16}$ | $1EF_{16}$ | $16F_{16}$ | Consente di impartire un comando, sulla base di parametri già forniti attraverso altri registri. A eccezione del comando DEVICE RESET, in tutti gli altri casi si può scrivere in questo registro soltanto se gli indicatori BSY e DRQ sono entrambi a zero. Questo registro è in sola scrittura, in quanto, se vi si accede in lettura, si ottiene il contenuto del registro ***regular status***. |
| *alternate status* | **r**/- | $3F6_{16}$ | $376_{16}$ | $3E6_{16}$ | $366_{16}$ | Dà lo stato del bus, attraverso una lettura «neutra», ovvero inerte. Quando è attivo l'indicatore BSY, tutti gli altri indicatori sono privi di significato. L'interpretazione del dato è equivalente a quella che si deve fare per il registro ***regular status***. Questo registro è in sola lettura, in quanto, se vi si accede in scrittura, si interviene invece nel registro ***device control***. |
| *control* (*device control*) | -/**w** | $3F6_{16}$ | $376_{16}$ | $3E6_{16}$ | $366_{16}$ | Consente di impostare alcuni indicatori che controllano la gestione dei dispositivi. Questo registro è in sola scrittura, in quanto, se vi si accede in lettura, si ottiene il contenuto del registro ***alternate status***. |

Nella tabella appena apparsa, sono indicati gli indirizzi di I/O per accedere a quattro diversi bus ATA, negli elaboratori che si rifanno all'architettura IBM PC. Va osservato che di norma sono disponibili solo due di tali bus (per un massimo di quattro dispositivi connessi complessivamente), pertanto, in tal caso vanno considerati solo i primi due di questi indirizzi.



Tabella 83.74. Indicatori dei registri di stato: ***regular status*** e ***alternate status***.

| $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|
| BSY *busy* | DRDY *device ready* | DF *device fault* | -- | DRQ *data request* | -- | -- | ERR *error* |

Tabella 83.75. Descrizione degli indicatori dei registri di stato: ***regular status*** e ***alternate status***.

| Indicatore | Descrizione |
|---|---|
| BSY *busy* | Indica che il dispositivo corrente del bus selezionato è impegnato e non si possono impartire dei comandi. Quando questo indicatore è attivo, gli altri, a parte ERR, sono privi di significato. |
| DRDY *device ready* | Questo indicatore **non** è l'opposto di BSY. Semplificando le cose, è a zero se il dispositivo è in pausa (motore fermo), mentre dovrebbe essere a uno in quasi tutte le altre situazioni. |
| DF *device fault* | Indica il verificarsi di un errore del dispositivo, diverso da quelli considerati dall'indicatore ERR. |
| DRQ *data request* | Indica che il dispositivo corrente è pronto per un trasferimento di dati. Al termine di un trasferimento, l'indicatore si azzera. |
| ERR *error* | Indica un errore, interpretabile leggendo il registro ***error***. |

Tabella 83.76. Bit del registro ***device control***.

| $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|
| HOB *high order byte* | -- | -- | -- | -- | SRST *software reset* | nIEN *not interrupt enabled* | 0 |

Tabella 83.77. Descrizione dei bit del registro ***device control***.

| Bit | Descrizione |
|---|---|
| HOB *high order byte* | Riguarda una situazione specifica dell'accesso LBA 48; in tutti gli altri casi va lasciato a zero. |
| SRST *software reset* | Richiede l'inizializzazione software dei dispositivi connessi al bus. |
| nIEN *not interrupt enabled* | Richiede la **disabilitazione** dell'emissione di segnali di interruzioni da parte del dispositivo corrente. |
| $bit_0$ | Il bit meno significativo deve sempre essere lasciato a zero. |

### 83.9.3 Alcuni comandi

L'invio di un comando al dispositivo corrente comporta l'indicazione di alcuni parametri utilizzando i registri, tra cui, soprattutto, il codice del comando stesso. Il comando viene eseguito nel momento in cui si scrive nel registro ***command*** il codice che lo identifica, pertanto questa scrittura va fatta per ultima.

Se prima di dare un comando si intende agire anche sul registro ***device control***, per esempio per inibire l'emissione di interruzioni per il dispositivo coinvolto, la scrittura in tale registro deve avvenire prima della scrittura nel registro ***command*** (per ovvi motivi), ma successivamente alla selezione del dispositivo con la scrittura nel registro ***device***.

#### 83.9.3.1 Comando READ SECTORS

Il comando READ SECTORS, corrispondente al codice $20_{16}$, consente di leggere uno o più settori, in modalità PIO, dal dispositivo specificato nel registro ***device***, con indirizzamento LBA28.

Tabella 83.78. Parametri del comando «READ SECTORS».

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | LBA | -- | DEV | Bit $bit_{24}$..$bit_{28}$ dell'indirizzo del primo settore da leggere. | | | |
| *feature* | -- | | | | | | | |
| *sector count* | Quantità di settori da leggere. | | | | | | | |
| *LBA low* | Bit $bit_0$..$bit_7$ dell'indirizzo del primo settore da leggere. | | | | | | | |
| *LBA mid* | Bit $bit_8$..$bit_{15}$ dell'indirizzo del primo settore da leggere. | | | | | | | |
| *LBA high* | Bit $bit_{16}$..$bit_{23}$ dell'indirizzo del primo settore da leggere. | | | | | | | |
| *command* | $20_{16}$ | | | | | | | |

Tabella 83.79. Descrizione dei registri coinvolti.

| Registro | Utilizzo |
|---|---|
| ***feature*** | Non viene considerato dal comando. |
| ***device*** | Il $bit_5$ e il $bit_7$ possono essere impostati a uno per mantenere la compatibilità con vecchi dispositivi, ma in generale sono semplicemente ignorati. Il $bit_6$, se attivo, indica che il primo settore da leggere viene individuato come numero assoluto (LBA), mentre se viene lasciato a zero, significa che tale settore viene raggiunto con coordinate CHS (cilindro, testina, settore). Il $bit_4$ individua il dispositivo, distinguendo tra primo (0) e secondo (1) del bus a cui si fa riferimento. L'intervallo di $bit_0$..$bit_3$ si utilizza per annotare la parte più significativa di un indirizzo LBA28, per raggiungere il primo settore da leggere. |
| ***lba low*** ***lba mid*** ***lba high*** | Questi tre registri rappresentano complessivamente i primi 24 bit dell'indirizzo del primo settore da raggiungere. |

Dopo l'invio del comando si deve attendere che l'indicatore BSY torni a zero e che l'indicatore DRQ si attivi, per poi procedere alla lettura di ***data***, generalmente a coppie di byte, fino al completamento della dimensione dei settori richiesti, quando l'indicatore DRQ torna a zero.

Tabella 83.80. Esito normale, atteso dopo l'esecuzione completa del comando, nel registro di stato.

| $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|
| BSY | DRDY | DF | -- | DRQ | -- | -- | ERR |
| 0 | 1 | 0 | -- | 0 | -- | -- | 0 |

### 83.9.3.2 Comando WRITE SECTORS

Il comando WRITE SECTORS, corrispondente al codice $30_{16}$, consente di scrivere uno o più settori, in modalità PIO, nel dispositivo specificato nel registro ***device***, con un indirizzamento LBA28.

Tabella 83.81. Parametri del comando WRITE SECTORS.

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | LBA | -- | DEV | Bit $bit_{24}$..$bit_{28}$ dell'indirizzo del primo settore da scrivere. | | | |
| *feature* | -- | | | | | | | |
| *sector count* | Quantità di settori da scrivere. | | | | | | | |
| *LBA low* | Bit $bit_0$..$bit_7$ dell'indirizzo del primo settore da scrivere. | | | | | | | |
| *LBA mid* | Bit $bit_8$..$bit_{15}$ dell'indirizzo del primo settore da scrivere. | | | | | | | |
| *LBA high* | Bit $bit_{16}$..$bit_{23}$ dell'indirizzo del primo settore da scrivere. | | | | | | | |
| *command* | $30_{16}$ | | | | | | | |

Tabella 83.82. Descrizione dei registri coinvolti.

| Registro | Utilizzo |
|---|---|
| ***feature*** | Non viene considerato dal comando. |



| Registro | Utilizzo |
|---|---|
| ***device*** | Il $bit_5$ e il $bit_7$ possono essere impostati a uno per mantenere la compatibilità con vecchi dispositivi, ma in generale sono semplicemente ignorati. Il $bit_6$, se attivo, indica che il primo settore da leggere viene individuato come numero assoluto (LBA), mentre se viene lasciato a zero, significa che tale settore viene raggiunto con coordinate CHS (cilindro, testina, settore). Il $bit_4$ individua il dispositivo, distinguendo tra primo (0) e secondo (1) del bus a cui si fa riferimento. L'intervallo di $bit_0$..$bit_3$ si utilizza per annotare la parte più significativa di un indirizzo LBA, per raggiungere il primo settore da scrivere. |
| ***lba low*** ***lba mid*** ***lba high*** | Questi tre registri rappresentano complessivamente i primi 24 bit dell'indirizzo del primo settore da raggiungere. |

Dopo l'invio del comando si deve attendere che l'indicatore BSY torni a zero e che l'indicatore DRQ si attivi, per poi procedere alla scrittura di ***data***, generalmente a coppie di byte, fino al completamento della dimensione dei settori richiesti, quando l'indicatore DRQ torna a zero.

Tabella 83.83. Esito normale, atteso dopo l'esecuzione completa del comando, nel registro di stato.

| $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|
| BSY | DRDY | DF | -- | DRQ | -- | -- | ERR |
| 0 | 1 | 0 | -- | 0 | -- | -- | 0 |

### 83.9.3.3 Comando FLUSH CACHE

Il comando CACHE FLUSH, corrispondente al codice $E7_{16}$, assicura la memorizzazione dei settori modificati ed è necessario inviarlo prima di procedere con ulteriori comandi di scrittura. L'operazione riguarda il dispositivo specificato nel registro ***device***.

Tabella 83.84. Parametri del comando FLUSH CACHE.

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | -- | -- | DEV | -- | | | |
| *feature* | – | | | | | | | |
| *sector count* | – | | | | | | | |
| *LBA low* | – | | | | | | | |
| *LBA mid* | – | | | | | | | |
| *LBA high* | – | | | | | | | |
| *command* | $E7_{16}$ | | | | | | | |

Tabella 83.85. Esito normale, atteso dopo l'esecuzione completa del comando, nel registro di stato.

| $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|
| BSY | DRDY | DF | -- | DRQ | -- | -- | ERR |
| 0 | 1 | 0 | -- | 0 | -- | -- | 0 |

### 83.9.3.4 Comando IDENTIFY DEVICE

Il comando IDENTIFY DEVICE, corrispondente al codice $EC_{16}$, consente di interrogare le caratteristiche di un dispositivo ATA, esclusi i dispositivi ATAPI.

Tabella 83.86. Parametri del comando IDENTIFY DEVICE.

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | -- | -- | DEV | -- | | | |
| *feature* | – | | | | | | | |
| *sector count* | – | | | | | | | |
| *LBA low* | – | | | | | | | |
| *LBA mid* | – | | | | | | | |
| *LBA high* | – | | | | | | | |
| *command* | $EC_{16}$ | | | | | | | |

Dopo l'invio del comando, si deve verificare il contenuto di uno dei due registri di stato: se questo fosse a zero, significa che il dispositivo richiesto non esiste. Se invece il registro contiene qualcosa, si deve attendere che l'indicatore BSY torni a zero e che l'indicatore DRQ si attivi, per poi procedere alla lettura di ***data***, a blocchi da 16 bit, per 256 volte (in totale si hanno 512 byte, come un settore comune), quando l'indicatore DRQ torna a zero. All'interno di questi blocchi da 16 bit ci sono informazioni che consentono di conoscere nel dettaglio le caratteristiche del dispositivo.

Se invece di un indicatore DRQ attivo si ottiene un errore, rappresentato quindi dall'indicatore ERR attivo, vanno letti i registri ***lba mid*** e ***lba high***:

Tabella 83.87. Contenuto dei registri ***lba mid*** e ***lba high*** in caso di errore ottenuto a seguito del comando IDENTIFY DEVICE.

| Tipo di errore | Registro ***lba mid*** | Registro ***lba high*** |
|---|---|---|
| Si tratta di unità ATA, ma si è verificato ugualmente un errore. | $00_{16}$ | $00_{16}$ |
| Si tratta di un'unità ATAPI parallela. | $14_{16}$ | $EB_{16}$ |
| Si tratta di un'unità SATA. | $3C_{16}$ | $C3_{16}$ |
| Si tratta di un'unità ATAPI seriale. | $69_{16}$ | $96_{16}$ |

Tabella 83.88. Esito normale, atteso dopo l'esecuzione completa del comando, inclusa la lettura del registro ***data***, nel registro di stato.

| $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|
| BSY | DRDY | DF | -- | DRQ | -- | -- | ERR |
| 0 | 1 | 0 | -- | 0 | -- | -- | 0 |

Tabella 83.89. Alcuni dati significativi ottenuti dalla lettura del registro ***data***, dopo il comando IDENTIFY DEVICE. I blocchi da 16 bit letti sono numerati da 0 (il primo), fino a 255 (l'ultimo).

| Blocco | Contenuto |
|---|---|
| 60, 61 | Complessivamente formano un numero a 32 bit che rappresenta la quantità totale di settori che si possono indirizzare in modalità LBA28. Se questo valore fosse a zero, indicherebbe che il dispositivo non è in grado di utilizzare un indirizzamento LBA28. |
| 100, 101, 102, 103 | Complessivamente formano un numero a 64 bit che rappresenta la quantità totale di settori che si possono indirizzare in modalità LBA48. Se questo valore fosse a zero, indicherebbe che il dispositivo non è in grado di utilizzare un indirizzamento LBA48, ma non è detto il contrario. |

### 83.9.4 Individuazione dei bus utilizzati

Il tentativo di comunicare con un bus privo di unità collegate, comporta delle risposte errate, pertanto è necessario, prima di ogni altra cosa, scandire i bus presenti per verificare quali di questi sono effettivamente utilizzabili. Si tratta di leggere il contenuto del registro di stato (il ***regular status*** per la precisione): se questo ha tutti i bit a uno, si tratta di un bus a cui non è collegato alcunché.

Il pezzetto di codice seguente, attraverso la funzione ***inb()***, che si intuisce serva a leggere un byte da una porta di I/O, si ottiene lo stato del primo bus, corrispondente all'indirizzo $1F7_{16}$.

```
…
status = inb (0x1F7);
if (status == 0xFF)
  {
    // Non ci sono dispositivi nel primo bus.
    …
  }
else
  {
    // Ci potrebbe essere almeno un dispositivo nel primo
    // bus.
    …
  }
…
```



### 83.9.5 Azzeramento dello stato dei dispositivi

Quando si verifica un errore, le unità ATA normali (non ATAPI) richiedono un azzeramento software, provocato attraverso il bit SRST attivo nel registro ***device control***. L'azzeramento riguarda però tutti i dispositivi connessi al bus, e, d'altro canto, non essendo coinvolto in questo un comando, non ci sarebbe il modo di precisare un dispositivo particolare. Va osservato che una volta scritto nel registro ***device control*** il valore corrispondente all'attivazione del bit SRST, occorre riscrivere un valore pari a zero per questo bit, altrimenti il bus rimarrebbe in uno stato di inizializzazione.

```
…
outb (0x3F6, 0x02);     // Azzera.
outb (0x3F6, 0x00);     // Ripristina la condizione normale.
…
```

### 83.9.6 Controllo delle interruzioni

In varie occasioni, i dispositivi possono mettersi in uno «stato di interruzione», a cui corrisponde effettivamente un interruzione hardware nell'architettura IBM PC. Dal momento che le situazioni in cui tali interruzioni si verificano sono varie e complesse, la loro gestione potrebbe essere troppo impegnativa. D'altro canto è possibile gestire i dispositivi ATA anche senza considerare le interruzioni.

A questo proposito è possibile scrivere nel registro ***device control*** il valore $01_{16}$, corrispondente al bit nIEN attivo, ogni volta che è appena stato selezionato un dispositivo nel registro ***device***, per evitare che il comando che si va a impartire produca poi un'interruzione.

### 83.9.7 Verifica dell'esito di un comando

Ogni volta che si dà un comando a un dispositivo ATA, se non si vogliono considerare le interruzioni, occorre controllare ripetutamente il registro di stato, precisamente il ***regular status***, per sapere quando è possibile procedere ulteriormente.

Si deve attendere che l'indicatore BSY si azzeri, quindi si deve verificare che gli indicatori ERR e DF siano a zero: se uno dei due ha un valore diverso, significa che si è verificato un errore. Se gli indicatori di errore sono a zero, se dopo il comando ci si attende di leggere o scrivere dati attraverso il registro ***data***, prima di poterlo fare, è necessario che l'indicatore DRQ sia attivo. Nell'esempio successivo si interroga l'esito di un comando appena impartito a un dispositivo del primo bus:

```
…
    // Legge 8 bit dal registro «regular status».
    status = inb (0x1F7);
    while (status & 0x80)
      {
        // BSY: continua a interrogare fino a che
        //      l'indicatore si azzera.
        status = inb (0x1F7);
      }
    if (status & 0x21)
      {
        // DF o ERR.
        return …;
      }
    if (status & 0x08)
      {
        // DRQ
        for (i = 0; i < 256; i++)
          {
            // Trasferisce dati attraverso il registro
            // «data».
            …
          }
      }
…
```

### 83.9.8 Identificazione delle unità

Una volta chiarito quali sono i bus che potrebbero contenere almeno un dispositivo, per sapere quali dispositivi sono presenti effettivamente e per conoscere le caratteristiche delle unità ATA presenti, si deve utilizzare il comando IDENTIFY DEVICE. A titolo di esempio si propone una funzione semplificata che riceve l'indicazione del numero del bus e del dispositivo di cui si vuole conoscere la dimensione massima in settori (da 512 byte), per un accesso in modalità LBA28: se la funzione restituisce zero, significa che il dispositivo non è disponibile o non può operare in modalità LBA28 oppure si è verificato un errore che ne impedisce l'identificazione. Nell'esempio, la funzione ***outb()*** serve a scrivere un byte in una certa porta di I/O, mentre la funzione ***inw()*** serve a leggere un intero a 16 bit da una certa porta.

```
unsigned int
identify_device (int bus, int drive)
{
    int           reg_device;
    int           reg_command;
    int           reg_status;
    int           reg_data;
    int           reg_control;
    unsigned char device;
    unsigned char status;
    int           i;
    uint16_t      id[256];
    unsigned int  size;
    //
    switch (bus)
      {
        case 0:
            reg_device  = 0x1F6;
            reg_command = 0x1F7;
            reg_status  = 0x1F7;
            reg_data    = 0x1F0;
            reg_control = 0x3F6;
            break;
        case 1:
            …
            break;
        …
      }
    device = 0x00;
    if (drive)
      {
        device = 0x10;
      }
    // Scrive 8 bit nel registro «device».
    outb (reg_device, device);
    // Scrive 8 bit nel registro «control».
    outb (reg_control, 0x01);
    // Scrive 8 bit nel registro «command».
    outb (reg_command, 0xEC);
    // Legge 8 bit dal registro «regular status».
    status = inb (reg_status);
    if (status == 0)
      {
        // L'unità non c'è.
        return 0;
      }
    while (status & 0x80)
      {
        // BSY
        status = inb (reg_status);
      }
    if (status & 0x21)
      {
        // DF o ERR: potrebbe trattarsi di unità ATAPI o
        // SATA.
        return 0;
      }
    //
    if (status & 0x08)
      {
        // DRQ
        for (i = 0; i < 256; i++)
          {
            // Legge 16 bit dal registro «data».
            id[i] = inw (reg_data);
          }
      }
    else
      {
        // Per un motivo inspiegabile, non si ottiene
        // il permesso di leggere i dati.
        return 0;
      }
    // Si estrapola la quantità di settori disponibili in
    // modalità LBA28: si prende il contenuto della memoria,
    // partendo da 'id[60]', per un'estensione di 32 bit
    // (4 byte), che così include anche 'id[61]',
    // trasformando la cosa attraverso dei puntatori.
    // Il meccanismo funziona perché i valori si
    // intendono rappresentati in modalità «little endian»,
    // per cui i byte meno significativi del valore appaiono
    // per primi.
    size = *((uint32_t *) &(id[60]));
    //
    return size;
}
```



### 83.9.9 Scomposizione dell'indirizzo

Quando si utilizzano comandi di lettura e scrittura di uno o più settori, si deve specificare l'indirizzo di questo, suddiviso in qualche modo nei registri che rappresentano i parametri del comando. Si distinguono tre casi, in base alle tre modalità di accesso: CHS, LBA28 e LBA48.

Tabella 83.94. Collocazione delle coordinate CHS.

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | 0 | -- | DEV | h (testina) | | | |
| *sector count* | quantità di settori da trattare | | | | | | | |
| *LBA low* | s (numero del settore della traccia, a partire da uno) | | | | | | | |
| *LBA mid* | c (cilindro) $bit_0$..$bit_7$ | | | | | | | |
| *LBA high* | c (cilindro) $bit_8$..$bit_{15}$ | | | | | | | |

Tabella 83.95. Collocazione dell'indirizzo LBA28.

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | 1 | -- | DEV | Bit $bit_{24}$..$bit_{28}$ | | | |
| *sector count* | quantità di settori da trattare | | | | | | | |
| *LBA low* | Bit $bit_0$..$bit_7$ | | | | | | | |
| *LBA mid* | Bit $bit_8$..$bit_{15}$ | | | | | | | |
| *LBA high* | Bit $bit_{16}$..$bit_{23}$ | | | | | | | |

Le due tabelle già apparse mostrano come articolare l'informazione CHS o l'indirizzo LBA28, assieme alla quantità di settori da prendere in considerazione. Nel caso in cui fosse specificata una quantità di settori pari a zero, si intenderebbero invece 256. Per la modalità LBA48, si procede in modo simile alla LBA28, con la differenza che i registri vanno scritti in due tornate e che il registro ***device*** non contiene alcuna porzione di questo.

Tabella 83.96. Utilizzo dei registri per collocare un indirizzo LBA48 in due fasi.

| Registro | $bit_7$ | $bit_6$ | $bit_5$ | $bit_4$ | $bit_3$ | $bit_2$ | $bit_1$ | $bit_0$ |
|---|---|---|---|---|---|---|---|---|
| *device* | -- | 1 | -- | DEV | – | | | |
| *sector count* | quantità di settori $bit_8$..$bit_{15}$ | | | | | | | |
| *LBA low* | Bit $bit_{24}$..$bit_{31}$ | | | | | | | |
| *LBA mid* | Bit $bit_{32}$..$bit_{39}$ | | | | | | | |
| *LBA high* | Bit $bit_{40}$..$bit_{48}$ | | | | | | | |
| *sector count* | quantità di settori $bit_0$..$bit_7$ | | | | | | | |
| *LBA low* | Bit $bit_0$..$bit_7$ | | | | | | | |
| *LBA mid* | Bit $bit_8$..$bit_{15}$ | | | | | | | |
| *LBA high* | Bit $bit_{16}$..$bit_{23}$ | | | | | | | |

In modalità LBA48, se si indica una quantità di settori pari a zero, si intendono invece 65536 settori.

### 83.9.10 Lettura LBA28 PIO

Viene proposto un esempio di funzione per la lettura di un settore, fornendo il numero del bus, del dispositivo all'interno del bus, il numero del settore (partendo da zero) e il puntatore all'inizio della memoria tampone che deve ricevere il settore. La funzione richiede ancora la verifica dei dati in ingresso e manca la possibilità di far scadere il ciclo di lettura del registro di stato nel caso in cui passasse troppo tempo.

La scrittura del registro ***device*** avviene per prima, per individuare subito il dispositivo e per consentire la scrittura successiva del registro ***control***, allo scopo di inibire una risposta tramite segnale di interruzione. Per il resto tutto procede come richiesto per il comando READ SECTORS.

```
int
read_sector (int bus, int drive, unsigned int sector,
             void *buffer)
{
    int           reg_device;
    int           reg_control;
    int           reg_feature;
    int           reg_count;
    int           reg_lba_low;
    int           reg_lba_mid;
    int           reg_lba_high;
    int           reg_command;
    int           reg_status;
    int           reg_data;
    unsigned char device;
    unsigned char lba_low;
    unsigned char lba_mid;
    unsigned char lba_high;
    unsigned char status;
    int           i;
    uint16_t     *destination = (uint16_t *) buffer;
    //
    switch (bus)
      {
        case 0:
            reg_device   = 0x1F6;
            reg_control  = 0x3F6;
            reg_feature  = 0x1F1;
            reg_count    = 0x1F2
            reg_lba_low  = 0x1F3;
            reg_lba_mid  = 0x1F4;
            reg_lba_high = 0x1F5;
            reg_command  = 0x1F7;
            reg_status   = 0x1F7;
            reg_data     = 0x1F0;
            break;
        case 1:
            …
            break;
        …
      }
    …
```



```
    // Preparazione e scrittura del registro «device». Nel
    // registro  va specificato il fatto che si utilizza un
    // accesso LBA, il dispositivo e la parte più
    // significativa dell'indirizzo LBA28.
    device = 0x00;
    device |= 0x40;                 // LBA.
    if (drive)
      {
        device |= 0x10;             // Secondo dispositivo;
      }
    device |= ((sector & 0x0F000000) >> 24);
    outb (reg_device, device);
    // Registro «control», per disabilitare il segnale di
    // interruzione.
    outb (reg_control, 0x02);
    // Registro «feature».
    outb (reg_feature, 0);
    // Registro «sector count»: si vuole leggere un solo
    // settore.
    outb (reg_count, 1);
    // LBA low, mid e high.
    lba_low  = (sector & 0x000000FF);
    lba_mid  = ((sector & 0x0000FF00) >> 8);
    lba_high = ((sector & 0x00FF0000) >> 16);
    outb (reg_lba_low,  lba_low);
    outb (reg_lba_mid,  lba_mid);
    outb (reg_lba_high, lba_high);
    // Registro «command».
    outb (reg_command, 0x20);
    // Si attende che lo stato del dispositivo corrente
    // torni a essere pronto.
    status = inb (reg_status);
    while (status & 0x80)
      {
        // BSY
        status = inb (reg_status);
      }
    if (status & 0x21)
      {
        // DF o ERR.
        return -1;
      }
    //
    if (status & 0x08)
      {
        // DRQ: si procede alla lettura del settore.
        for (i = 0; i < 256; i++)
          {
            // Legge 16 bit dal registro «data».
            destination[i] = inw (reg_data);
          }
      }
    else
      {
        // Errore sconosciuto.
        return -1;
      }
    // Attesa ulteriore che l'unità sia di nuovo pronta.
    status = inb (reg_status);
    while (status & 0x80)
      {
        // BSY
        status = inb (reg_status);
      }
    if (status & 0x21)
      {
        // DF o ERR.
        return -1;
      }
    // Fine normale.
    return 0;
}
```

### 83.9.11 Scrittura LBA28 PIO

Viene proposto un esempio di funzione per la scrittura di un settore, fornendo il numero del bus, del dispositivo all'interno del bus, il numero del settore (partendo da zero) e il puntatore all'inizio della memoria tampone che contiene il settore da scrivere. La funzione è

molto simile a quella proposta per la lettura e vengono omesse le porzioni uguali.

La differenza fondamentale sta nel fatto che, dopo la scrittura del settore (con l'aiuto della funzione ***outw()***), dopo aver verificato che il registro di stato segnala la conclusione corretta dell'operazione, va dato un comando di scarico della memoria trattenuta (*cache*).

```
int
write_sector (int bus, int drive, unsigned int sector,
              void *buffer)
{
    …
    uint16_t     *source = (uint16_t *) buffer;
    …
    …
    // Registro «command».
    outb (reg_command, 0x30);
    …
    if (status & 0x08)
      {
        // DRQ: si procede alla scrittura del settore.
        for (i = 0; i < 256; i++)
          {
            // Scrive 16 bit nel registro «data».
            outw (reg_data, source[i]);
          }
      }
    …
    // Quando il dispositivo torna a essere pronto, si dà
    // un comando di scarico della memoria trattenuta:
    // CACHE FLUSH. Il registro «device» contiene già
    // tutti i dati che servono e non c'è bisogno di
    // aggiornarlo.
    outb (reg_command, 0xE7);
    // Si attende ancora che il dispositivo torni a essere
    // pronto, prima di concludere.
    status = inb (reg_status);
    while (status & 0x80)
      {
        // BSY
        status = inb (reg_status);
      }
    if (status & 0x21)
      {
        // DF o ERR.
        return -1;
      }
    // Fine normale.
    return 0;
}
```

## 83.10 PCI

Il bus PCI (*Peripheral component interconnect*) si occupa di organizzare le risorse assegnate ai componenti a lui connessi, offrendo al programmatore le informazioni necessarie per accedervi. Tale organizzazione automatica avviene attraverso una procedura software, la quale però è già inclusa nel firmware degli elaboratori IBM AT.

Su un bus PCI possono essere connessi altri bus, sia per estenderlo, sia per permettere il collegamento con altri tipi di bus. Quando si deve accedere a un componente che si trova in un bus secondario, si creano delle complicazioni che qui non vengono considerate, limitando l'attenzione al caso in cui si gestiscano esclusivamente componenti PCI connessi direttamente al primo bus. Pertanto, con questa semplificazione, è sufficiente interrogare i registri del bus PCI per sapere quali dispositivi sono connessi e quali riferimenti servono per comunicare con loro.

Va tenuto in considerazione che tutta la gestione del bus PCI prevede che i dati contenuti in scalari interi siano organizzati in modalità *little endian*; pertanto, in un'architettura x86 non si creano problemi di conversione, ammesso che si rispetti la struttura prevista per questi valori.



### 83.10.1 Registri e porte

Per comunicare con il bus PCI, allo scopo di ottenere le informazioni che servono sul bus e sui componenti inseriti, si usano due porte di I/O. La prima, corrispondente all'indirizzo $0CF8_{16}$, serve per fornire delle coordinate, con cui individuare un registro accessibile attraverso la seconda, corrispondente all'indirizzo $0CFC_{16}$.

| Indirizzo I/O | Accesso | Nome mnemonico | Descrizione |
|---|---|---|---|
| $0CF8_{16}$ | 32 bit | *address* | Scrivendo un valore a 32 bit, strutturato opportunamente, si individua un registro accessibile poi all'indirizzo *data*. |
| $0CFC_{16}$ | 32 bit | *data* | Si legge o si scrive un valore a 32 bit che costituisce l'informazione a cui puntano le coordinate fornite in precedenza attraverso l'indirizzo *address*. |

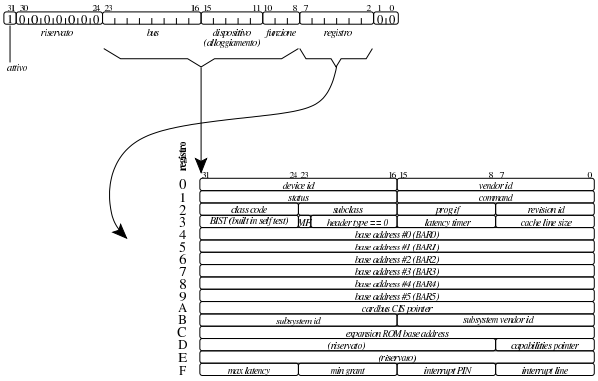
Le informazioni che il bus PCI ha da offrire sono strutturate in «tabelle» (ma nella documentazione standard si parla di «intestazioni»), suddivise in registri da 32 bit, suddivisi a loro volta in campi non omogenei tra loro. L'accesso a queste tabelle avviene registro per registro, attraverso la porta $0CFC_{16}$ (*data*), ma per individuare il registro a cui si è interessati, prima di ciò occorre scrivere un valore appropriato nella porta $0CF8_{16}$ (*address*).

Per raggiungere un dato occorre conoscere il numero del bus, dell'alloggiamento e del registro.

### 83.10.2 Strutture dei dati

La figura successiva mostra le strutture in cui si articolano il selettore per individuare un registro e la tabella dei registri corrispondente (viene mostrata solo la tabella di tipo $00_{16}$).

Figura 83.100. Selettore e tabella di tipo $00_{16}$.




Vengono proposte anche due strutture, in linguaggio C, per descrivere il selettore e la tabella di tipo $00_{16}$.

Listato 83.101. Struttura del selettore. Il membro ***selector*** consente di azzerare velocemente tutto il selettore, senza dover intervenire sui singoli campi.

```
typedef union {
  uint32_t    selector;
  struct {
    uint32_t  zero      : 2,
              reg       : 6,
              function  : 3,
              slot      : 5,
              bus       : 8,
              reserved  : 7,
              enable    : 1;
  };
} pci_address_t;
```

Listato 83.102. Struttura della tabella di tipo $00_{16}$. L'array ***r[16]*** consente di individuare facilmente un registro nel suo complesso.

```
typedef union {
  uint32_t r[16];
  struct {
    struct {
        uint32_t  vendor_id                 : 16,
                  device_id                 : 16;
        //
        uint32_t  command                   : 16,
                  status                    : 16;
        //
        uint32_t  revision_id               : 8,
                  prog_if                   : 8,
                  subclass                  : 8,
                  class_code                : 8;
        //
        uint32_t  cache_line_size           : 8,
                  latency_timer             : 8,
                  header_type               : 7,
                  multi_function            : 1,
                  bist                      : 8;
        //
        uint32_t  bar0;
        uint32_t  bar1;
        uint32_t  bar2;
        uint32_t  bar3;
        uint32_t  bar4;
        uint32_t  bar5;
        uint32_t  cardbus_cis_pointer;
        uint32_t  expansion_rom_base_address;
        //
        uint32_t  subsystem_vendor_id       : 16,
                  subsystem_id              : 16;
        //
        uint32_t  capabilities_pointer      : 8,
                  reserved_1                : 24;
        //
        uint32_t  reserved_2;
        //
        uint32_t  interrupt_line            : 8,
                  interrupt_pin             : 8,
                  min_grant                 : 8,
                  max_latency               : 8;
    };
  };
} pci_header_type_00_t;
```

Tabella 83.103. Campi in cui si articola il selettore.

| Bit | Denominazione | Descrizione |
|---|---|---|
| 31 | *enable* | Questo bit deve essere posto a uno. |
| 30..24 | *reserved* | Campo non utilizzato, da lasciare azzerato. |
| 23..16 | *bus* | Il numero del bus a cui si intende fare riferimento. Il primo bus è quello corrispondente al numero zero. |
| 15..11 | *device* | Il numero dell'alloggiamento all'interno del bus, a partire da zero. |
| 10..8 | *funzione* | Nel caso di un dispositivo che incorpora più funzioni (per esempio più interfacce di rete assieme), questo è il numero che consente di selezionare il componente singolo, ovvero la funzione singola, tra tutte quelle incorporate. |
| 7..2<br>1..0 | *register* | I bit 7..2 individuano il numero del registro relativo alla tabella (*header*) che raccoglie la configurazione del dispositivo selezionato. I primi due bit rimangono generalmente a zero, perché l'accesso alla tabella è a blocchi da 32 bit, ovvero 4 byte, pertanto non c'è motivo di raggiungere posizioni intermedie. |



La tabella di tipo $00_{16}$, a cui si fa riferimento qui, è quella che riguarda i dispositivi comuni. Hanno invece tabelle differenti i dispositivi che connettono assieme più bus, dello stesso tipo o di tipo differente. A ogni modo, per facilitare un po' le cose, i primi quattro registri di queste tabelle sono uguali in tutte le tipologie.

Tabella 83.104. Campi principali della tabella di tipo $00_{16}$.

| Registro | Bit | Denominazione | Descrizione |
|---|---|---|---|
| $0_{16}$ | 31..16 | *device id* | Codice identificativo del tipo di dispositivo, stabilito dal costruttore. |
| $0_{16}$ | 15..0 | *vendor id* | Codice identificativo del costruttore: se in questo campo si ottiene il valore $FFFF_{16}$, significa che il dispositivo non è presente nel bus e nell'alloggiamento cercato. |
| $2_{16}$ | 31..24<br>23..16<br>15..8 | *class code*<br>*subclass*<br>*prog if* | Codice che descrive la classe, la sottoclasse e la funzione del dispositivo (tabella 83.107). |
| $3_{16}$ | 23 | *multi function* | Se questo bit è a uno, significa che il dispositivo è multifunzione. |
| $3_{16}$ | 22..16 | *header type* | Codice che descrive la struttura della tabella: $00_{16}$ indica quella di un dispositivo normale, $01_{16}$ riguarda un ponte tra bus PCI e $02_{16}$ riguarda un ponte verso unità cardbus. |
| $4_{16}..9_{16}$ | 31..0 | *BAR0*<br>*BAR1*<br>*BAR2*<br>*BAR3*<br>*BAR4*<br>*BAR5* | Si tratta di indirizzi che rappresentano un riferimento nella memoria o negli indirizzi di I/O. Con i dispositivi a cui si accede attraverso un indirizzo IRQ e un solo indirizzo I/O, BAR0 corrisponde all'indirizzo di I/O. Tuttavia, il valore contenuto in questi registri va interpretato e non si può usare tale e quale, come appare (figura 83.105). |
| $F_{16}$ | 7..0 | *Interrupt line* | Si tratta del numero di IRQ usato dal dispositivo, ma se si ottiene il valore $FF_{16}$, vuol dire che il dispositivo non utilizza un IRQ. |

Figura 83.105. Interpretazione di un indirizzo contenuto in un campo BAR***n***. Si osserva che il bit meno significativo consente di capire se si tratta di un indirizzo in memoria o di una porta di I/O.

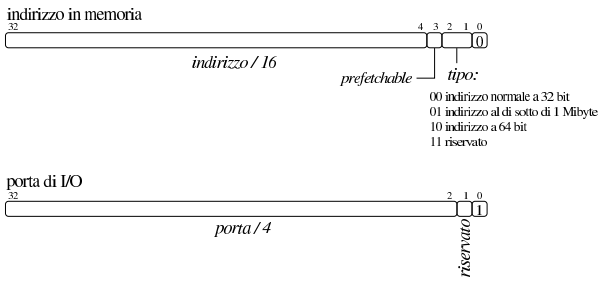


Tabella 83.106. Interpretazione della classe del dispositivo (*class code*).

| *class code* | Definizione | *class code* | Definizione |
|---|---|---|---|
| $00_{16}$ | dispositivo realizzato prima della definizione delle classi | $01_{16}$ | memoria di massa |
| $02_{16}$ | dispositivo di rete | $03_{16}$ | dispositivo di visualizzazione |
| $04_{16}$ | dispositivo multimediale | $05_{16}$ | memoria |
| $06_{16}$ | ponte (connessione con altri bus) | $07_{16}$ | porta seriale |
| $08_{16}$ | dispositivo di sistema | $09_{16}$ | dispositivo di ingresso |
| $0A_{16}$ | *docking stations* | $0B_{16}$ | microprocessore |
| $0C_{16}$ | bus seriale | $0D_{16}$ | dispositivo senza fili |
| $0E_{16}$ | dispositivo di I/O intelligente | $0F_{16}$ | comunicazione satellitare |
| $10_{16}$ | dispositivo crittografico | $11_{16}$ | acquisizione dati ed elaborazione dei segnali |
| $12_{16}..FE_{16}$ | riservato | $FF_{16}$ | dispositivo diverso dalle classi esistenti |

Tabella 83.107. Interpretazione completa della classe, sottoclasse e interfaccia di programmazione.

| *class code* | *subclass* | *prog if* | Descrizione |
|---|---|---|---|
| $00_{16}$ | dispositivo realizzato prima della definizione delle classi | $01_{16}$ | memoria di massa |
| $00_{16}$ | $00_{16}$ | $00_{16}$ | dispositivo non meglio precisato, escludendo però il caso di un dispositivo VGA |
| $00_{16}$ | $01_{16}$ | $00_{16}$ | VGA e compatibili |
| $01_{16}$ | $00_{16}$ | $00_{16}$ | SCSI |
| $01_{16}$ | $01_{16}$ | $--_{16}$ | IDE |
| $01_{16}$ | $02_{16}$ | $00_{16}$ | unità a dischetti |
| $01_{16}$ | $03_{16}$ | $00_{16}$ | IPI |
| $01_{16}$ | $04_{16}$ | $00_{16}$ | RAID |
| $01_{16}$ | $05_{16}$ | $20_{16}$ | PATA, DMA singolo |
| $01_{16}$ | $05_{16}$ | $30_{16}$ | PATA, DMA concatenato |
| $01_{16}$ | $06_{16}$ | $00_{16}$ | SATA |
| $01_{16}$ | $80_{16}$ | $00_{16}$ | memoria di massa diversa |
| $02_{16}$ | $00_{16}$ | $00_{16}$ | Ethernet |
| $02_{16}$ | $01_{16}$ | $00_{16}$ | Token Ring |
| $02_{16}$ | $02_{16}$ | $00_{16}$ | FDDI |
| $02_{16}$ | $03_{16}$ | $00_{16}$ | ATM |
| $02_{16}$ | $04_{16}$ | $00_{16}$ | ISDN |
| $02_{16}$ | $05_{16}$ | $00_{16}$ | WorldFip |
| $02_{16}$ | $06_{16}$ | -- | PICMG 2.14 Multi Computing |
| $02_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di rete diversa |
| $03_{16}$ | $00_{16}$ | $00_{16}$ | VGA |
| $03_{16}$ | $00_{16}$ | $01_{16}$ | 8512 |
| $03_{16}$ | $01_{16}$ | $00_{16}$ | XGA |
| $03_{16}$ | $02_{16}$ | $00_{16}$ | 3D (non VGA) |
| $03_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia video-grafica diversa |
| $04_{16}$ | $00_{16}$ | $00_{16}$ | Video multimediale |
| $04_{16}$ | $01_{16}$ | $00_{16}$ | Audio multimediale |
| $04_{16}$ | $02_{16}$ | $00_{16}$ | Computer Telephony |
| $04_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia multimediale diversa |
| $05_{16}$ | $00_{16}$ | $00_{16}$ | RAM |
| $05_{16}$ | $01_{16}$ | $00_{16}$ | Flash |
| $05_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di memoria diversa |
| $06_{16}$ | $00_{16}$ | $00_{16}$ | Host Bridge |
| $06_{16}$ | $01_{16}$ | $00_{16}$ | ISA Bridge |
| $06_{16}$ | $02_{16}$ | $00_{16}$ | EISA Bridge |
| $06_{16}$ | $03_{16}$ | $00_{16}$ | MCA Bridge |



| *class code* | *subclass* | *prog if* | Descrizione |
|---|---|---|---|
| $06_{16}$ | $00_{16}$ | $00_{16}$ | PCI-to-PCI Bridge |
| $06_{16}$ | $00_{16}$ | $01_{16}$ | PCI-to-PCI Bridge (Subtractive Decode) |
| $06_{16}$ | $05_{16}$ | $00_{16}$ | PCMCIA Bridge |
| $06_{16}$ | $06_{16}$ | $00_{16}$ | NuBus Bridge |
| $06_{16}$ | $07_{16}$ | $00_{16}$ | CardBus Bridge |
| $06_{16}$ | $08_{16}$ | $--_{16}$ | RACEway Bridge |
| $06_{16}$ | $09_{16}$ | $40_{16}$ | PCI-to-PCI Bridge (Semi-Transparent, Primary) |
| $06_{16}$ | $09_{16}$ | $80_{16}$ | PCI-to-PCI Bridge (Semi-Transparent, Secondary) |
| $06_{16}$ | $0A_{16}$ | $00_{16}$ | InfiniBrand-to-PCI Host Bridge |
| $06_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di connessione ad altri bus, diversa dai tipi previsti |
| $07_{16}$ | $00_{16}$ | $00_{16}$ | porta seriale XT |
| $07_{16}$ | $00_{16}$ | $01_{16}$ | porta seriale 16450 |
| $07_{16}$ | $00_{16}$ | $02_{16}$ | porta seriale 16550 |
| $07_{16}$ | $00_{16}$ | $03_{16}$ | porta seriale 16650 |
| $07_{16}$ | $00_{16}$ | $04_{16}$ | porta seriale 16750 |
| $07_{16}$ | $00_{16}$ | $05_{16}$ | porta seriale 16850 |
| $07_{16}$ | $00_{16}$ | $06_{16}$ | porta seriale 16950 |
| $07_{16}$ | $01_{16}$ | $00_{16}$ | porta parallela |
| $07_{16}$ | $01_{16}$ | $01_{16}$ | porta parallela bidirezionale |
| $07_{16}$ | $01_{16}$ | $02_{16}$ | ECP 1.X |
| $07_{16}$ | $01_{16}$ | $03_{16}$ | IEEE 1284 |
| $07_{16}$ | $01_{16}$ | $FE_{16}$ | IEEE 1284 |
| $07_{16}$ | $02_{16}$ | $00_{16}$ | unità seriale multipla |
| $07_{16}$ | $03_{16}$ | $00_{16}$ | modem generico |
| $07_{16}$ | $03_{16}$ | $01_{16}$ | modem Hayes 16450 |
| $07_{16}$ | $03_{16}$ | $02_{16}$ | modem Hayes 16550 |
| $07_{16}$ | $03_{16}$ | $03_{16}$ | modem Hayes 16650 |
| $07_{16}$ | $03_{16}$ | $04_{16}$ | modem Hayes 16750 |
| $07_{16}$ | $04_{16}$ | $00_{16}$ | IEEE 488.1/2 (GPIB) |
| $07_{16}$ | $05_{16}$ | $00_{16}$ | Smart Card |
| $07_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di comunicazione diversa |
| $08_{16}$ | $00_{16}$ | $00_{16}$ | 8259 PIC |
| $08_{16}$ | $00_{16}$ | $01_{16}$ | ISA PIC |
| $08_{16}$ | $00_{16}$ | $02_{16}$ | EISA PIC |
| $08_{16}$ | $00_{16}$ | $10_{16}$ | I/O APIC |
| $08_{16}$ | $00_{16}$ | $20_{16}$ | I/O(x) APIC |
| $08_{16}$ | $01_{16}$ | $00_{16}$ | 8237 DMA |
| $08_{16}$ | $01_{16}$ | $01_{16}$ | ISA DMA |
| $08_{16}$ | $01_{16}$ | $02_{16}$ | EISA DMA |
| $08_{16}$ | $02_{16}$ | $00_{16}$ | 8254 System Timer |
| $08_{16}$ | $02_{16}$ | $01_{16}$ | ISA System Timer |
| $08_{16}$ | $02_{16}$ | $02_{16}$ | EISA System Timer |
| $08_{16}$ | $03_{16}$ | $00_{16}$ | RTC generico |
| $08_{16}$ | $03_{16}$ | $01_{16}$ | ISA RTC |
| $08_{16}$ | $04_{16}$ | $00_{16}$ | unità PCI Hot-Plug generica |
| $08_{16}$ | $80_{16}$ | $00_{16}$ | dispositivo di sistema diverso |
| $09_{16}$ | $00_{16}$ | $00_{16}$ | tastiera |
| $09_{16}$ | $01_{16}$ | $00_{16}$ | *digitizer* |
| $09_{16}$ | $02_{16}$ | $00_{16}$ | mouse |
| $09_{16}$ | $03_{16}$ | $00_{16}$ | scanner |
| $09_{16}$ | $04_{16}$ | $00_{16}$ | gameport |
| $09_{16}$ | $04_{16}$ | $10_{16}$ | gameport |
| $09_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di input diversa |
| $0A_{16}$ | $00_{16}$ | $00_{16}$ | Docking Station |
| $0A_{16}$ | $80_{16}$ | $00_{16}$ | Docking Station diversa |
| $0B_{16}$ | $00_{16}$ | $00_{16}$ | CPU 386 |
| $0B_{16}$ | $01_{16}$ | $00_{16}$ | CPU 486 |
| $0B_{16}$ | $02_{16}$ | $00_{16}$ | CPU Pentium |
| $0B_{16}$ | $10_{16}$ | $00_{16}$ | CPU Alpha |
| $0B_{16}$ | $20_{16}$ | $00_{16}$ | CPU PowerPC |

| *class code* | *subclass* | *prog if* | Descrizione |
|---|---|---|---|
| $0B_{16}$ | $30_{16}$ | $00_{16}$ | CPU MIPS |
| $0B_{16}$ | $40_{16}$ | $00_{16}$ | coprocessore |
| $0C_{16}$ | $00_{16}$ | $00_{16}$ | IEEE 1394 (FireWire) |
| $0C_{16}$ | $00_{16}$ | $10_{16}$ | IEEE 1394 (1394 Open-HCI Spec) |
| $0C_{16}$ | $01_{16}$ | $00_{16}$ | ACCESS.bus |
| $0C_{16}$ | $02_{16}$ | $00_{16}$ | SSA |
| $0C_{16}$ | $03_{16}$ | $00_{16}$ | USB (Universal Host) |
| $0C_{16}$ | $03_{16}$ | $10_{16}$ | USB (Open Host) |
| $0C_{16}$ | $03_{16}$ | $20_{16}$ | USB2 Host (Intel Enhanced Host Controller Interface) |
| $0C_{16}$ | $03_{16}$ | $80_{16}$ | USB |
| $0C_{16}$ | $03_{16}$ | $FE_{16}$ | USB (Not Host Controller) |
| $0C_{16}$ | $04_{16}$ | $00_{16}$ | Fibre Channel |
| $0C_{16}$ | $05_{16}$ | $00_{16}$ | SMBus |
| $0C_{16}$ | $06_{16}$ | $00_{16}$ | InfiniBand |
| $0C_{16}$ | $07_{16}$ | $00_{16}$ | IPMI SMIC |
| $0C_{16}$ | $07_{16}$ | $01_{16}$ | IPMI Kybd |
| $0C_{16}$ | $07_{16}$ | $02_{16}$ | IPMI Block Transfer |
| $0C_{16}$ | $08_{16}$ | $00_{16}$ | SERCOS (IEC 61491) |
| $0C_{16}$ | $09_{16}$ | $00_{16}$ | CANbus |
| $0D_{16}$ | $00_{16}$ | $00_{16}$ | iRDA |
| $0D_{16}$ | $01_{16}$ | $00_{16}$ | Consumer IR |
| $0D_{16}$ | $10_{16}$ | $00_{16}$ | RF Controller |
| $0D_{16}$ | $11_{16}$ | $00_{16}$ | Bluetooth |
| $0D_{16}$ | $12_{16}$ | $00_{16}$ | Broadband |
| $0D_{16}$ | $20_{16}$ | $00_{16}$ | Ethernet WiFi (802.11a) |
| $0D_{16}$ | $21_{16}$ | $00_{16}$ | Ethernet WiFi (802.11b) |
| $0D_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di rete senza fili, diversa da quelle previste |
| $0E_{16}$ | $00_{16}$ | -- | I20 Architecture |
| $0E_{16}$ | $00_{16}$ | $00_{16}$ | Message FIFO |
| $0F_{16}$ | $01_{16}$ | $00_{16}$ | TV |
| $0F_{16}$ | $02_{16}$ | $00_{16}$ | Audio |
| $0F_{16}$ | $03_{16}$ | $00_{16}$ | Voice |
| $0F_{16}$ | $04_{16}$ | $00_{16}$ | Data |
| $10_{16}$ | $00_{16}$ | $00_{16}$ | Network and Computing Encryption/Decryption |
| $10_{16}$ | $10_{16}$ | $00_{16}$ | Entertainment Encryption/Decryption |
| $10_{16}$ | $80_{16}$ | $00_{16}$ | Other Encryption/Decryption |
| $11_{16}$ | $00_{16}$ | $00_{16}$ | DPIO Modules |
| $11_{16}$ | $01_{16}$ | $00_{16}$ | Performance Counters |
| $11_{16}$ | $10_{16}$ | $00_{16}$ | Communications Synchronization Plus Time and Frequency Test/Measurement |
| $11_{16}$ | $20_{16}$ | $00_{16}$ | Management Card |
| $11_{16}$ | $80_{16}$ | $00_{16}$ | interfaccia di acquisizione dati o di elaborazione dei segnali, diversa da quelle previste |

### 83.10.3 Raccolta delle informazioni

Per raccogliere le informazioni sui dispositivi connessi a un bus PCI, occorre predisporre un selettore. Per esempio, utilizzando una variabile strutturata di tipo '**`pci_address_t`**' si potrebbe richiedere di accedere al bus ***b***, al dispositivo ***s*** (*slot*), alla funzione ***f*** e al registro ***r***:



```
int           b;
int           s;
int           f;
int           r;
pci_address_t pci_addr;
uint32_t      reg;
…
pci_addr.selector   = 0;
pci_addr.enable     = 1;
pci_addr.bus        = b;
pci_addr.slot       = s;
pci_addr.function   = f;
pci_addr.reg        = r;
out_32 (0x0CF8, pci_addr.selector);
reg = in_32 (0x0CFC);
```

Nell'esempio, le funzioni ***out_32()*** e ***in_32()*** utilizzano in pratica le istruzioni '**`OUTL`**' e '**`INL`**' del linguaggio assemblatore (si vedano eventualmente i listati 94.6, 94.6.5 e 94.6.2). Alla fine, la variabile ***reg*** raccoglie il contenuto del registro selezionato.

Per scandire un bus PCI è possibile procedere provando tutte le combinazioni di bus, alloggiamento e funzione, verificando che il primo registro sia diverso da $0xFFFFFFFF_{16}$. Se è così si può raccogliere il contenuto della tabella corrispondente. Nell'esempio seguente si scandiscono tutti i bus PCI, ignorando i dispositivi di classe $06_{16}$, raccogliendo alcuni dati dei dispositivi validi in un array con elementi di tipo '**`struct pci`**'. Si esclude che si possano incontrare dispositivi con più funzioni; inoltre si ritiene di incontrare soltanto dispositivi che contengono nel campo ***BAR0*** una porta di I/O.

```
struct {
      unsigned char      bus;
      unsigned char      slot;
      unsigned short int vendor_id;
      unsigned short int device_id;
      unsigned char      class_code;
      unsigned char      subclass;
      unsigned char      prog_if;
      uintptr_t          base_io;
      unsigned char      irq;
  } pci[8];
pci_header_type_00_t pci;
pci_address_t        pci_addr;
//
int t;        // PCI table index.
int b;        // PCI bus index.
int s;        // PCI slot index.
int r;        // PCI header register index.
//
// Reset the PCI table.
//
for (t = 0; t < 8; t++)
  {
    pci_table[t].bus        = 0;
    pci_table[t].slot       = 0;
    pci_table[t].vendor_id  = 0;
    pci_table[t].device_id  = 0;
    pci_table[t].class_code = 0;
    pci_table[t].subclass   = 0;
    pci_table[t].prog_if    = 0;
    pci_table[t].base_io    = 0;
    pci_table[t].irq        = 0;
  }
//
// Scan PCI buses and slots.
//
t = 0;
//
for (b = 0; b < 256 && t < 8; b++)
  {
    for (s = 0; s < 32 && t < 8; s++)
      {
        pci_addr.selector   = 0;
        pci_addr.enable     = 1;
        pci_addr.bus        = b;
        pci_addr.slot       = s;
        //
        pci_addr.reg        = 0;
        out_32 (0x0CF8, pci_addr.selector);
        pci.r[0] = in_32 (0x0CFC);
```

```
          //
          if (pci.r[0] == 0xFFFFFFFF)
            {
              //
              // There is no such bus:slot combination!
              //
              continue;
            }
          else
            {
              for (r = 1; r < 16; r++)
                {
                  pci_addr.reg        = r;
                  out_32 (0x0CF8, pci_addr.selector);
                  pci.r[r] = in_32 (0x0CFC);
                }
            }
          //
          // We consider only PCI header type 0x00!
          //
          if (pci.header_type != 0)
            {
              continue;
            }
          //
          // We do not consider PCI bridge devices!
          //
          if (pci.class_code == 0x06)
            {
              continue;
            }
          //
          // Save the device inside the PCI table.
          //
          pci_table[t].bus        = b;
          pci_table[t].slot       = s;
          pci_table[t].vendor_id  = pci.vendor_id;
          pci_table[t].device_id  = pci.device_id;
          pci_table[t].class_code = pci.class_code;
          pci_table[t].subclass   = pci.subclass;
          pci_table[t].prog_if    = pci.prog_if;
          pci_table[t].base_io    = pci.bar0 & 0xFFFFFFFC;
          pci_table[t].irq        = pci.interrupt_line;
          //
          // Next PCI table row.
          //
          t++;
        }
    }
```

Come si può osservare, la presunta porta di I/O viene filtrata con una maschera, in modo da azzerare i due bit meno significativi:

```
pci_table[t].base_io = pci.bar0 & 0xFFFFFFFC;
```

Al termine della scansione, la combinazione dei codici identificativi del produttore e del dispositivo, permettono di sapere di che cosa si tratta, disponendo naturalmente di un elenco appropriato. Per esempio, il produttore $10EC_{16}$ corrisponde a Realtek Semiconductor, mentre il dispositivo $8029_{16}$ (relativo a tale produttore) corrisponde a un'interfaccia di rete RT8029, compatibile con la vecchia NE2000.

## 83.11 NE2000

Le interfacce di rete NE2000 hanno delle limitazioni significative e sono complesse da programmare. Tuttavia, questo tipo di dispositivo è quello più facilmente disponibile negli emulatori, per esempio è presente sia in Bochs, sia in Qemu, pertanto diventa una scelta obbligata, almeno inizialmente.

Le annotazioni fatte qui, a proposito delle interfacce di rete NE2000, sono insufficienti per una gestione completa di tali unità. Eventualmente si possono consultare due schede tecniche, citate anche alla fine del capitolo: *DP8390D/NS32490D NIC network interface controller* e *Writing drivers for DP8390 NIC family of ethernet controllers*.



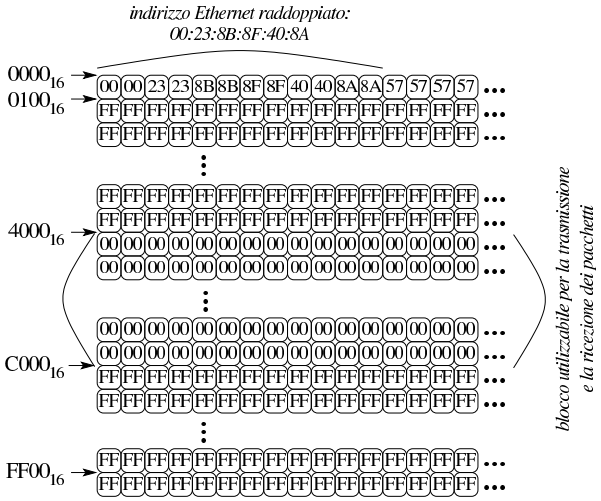
### 83.11.1 Memoria interna

Un aspetto importante della programmazione dell'interfaccia di rete NE2000 riguarda la memoria interna, complessivamente di 64 Kibyte, entro la quale va individuata una porzione per i pacchetti da trasmettere (in questo contesto sono più precisamente trame) e un'altra per la coda di ricezione. Per accedere a questa memoria, sia in scrittura, sia in lettura, occorrono dei comandi opportuni, per poi eseguire l'operazione attraverso una porta di I/O.

La memoria interna è organizzata a blocchi da 256 byte ($100_{16}$), perché alcuni registri usati come puntatori possono farvi riferimento, disponendo solo di 8 bit.

La porzione iniziale di questa memoria contiene delle informazioni importanti sull'interfaccia; in particolare è annotato lì l'indirizzo Ethernet attribuito dal costruttore. Va osservato che la porzione utile per la collocazione dei pacchetti da trasmettere o da ricevere va dall'indirizzo $4000_{16}$ a $BFFF_{16}$ (estremi inclusi); il resto deve essere lasciato alla gestione interna dell'interfaccia.

Figura 83.111. Organizzazione della memoria tampone interna di un'interfaccia di rete NE2000.

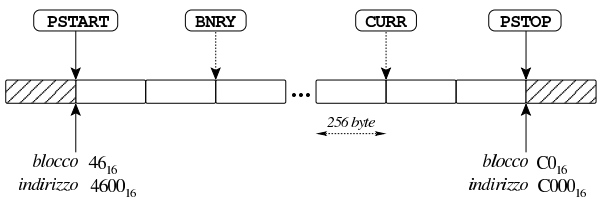



Per un'interfaccia di rete NE2000, il contenuto di un pacchetto, esclusa l'intestazione Ethernet, può andare da un minimo di 46 byte a un massimo di 1500 byte. Data l'organizzazione della memoria interna dell'interfaccia, un pacchetto utilizza da uno a sei blocchi da 256 byte (se un pacchetto è molto breve, utilizza ugualmente un blocco intero di memoria).

### 83.11.2 Coda di ricezione

Quando l'interfaccia di rete riceve un pacchetto, lo colloca in una porzione della propria memoria tampone, organizzata in forma di coda, a blocchi di 256 byte. Questa porzione di memoria viene chiamato «anello», perché una volta raggiunto l'ultimo blocco, si riprende dal primo.

Figura 83.112. Esempio di coda per la ricezione dei pacchetti, dal byte $4600_{16}$ al $BFFF_{16}$, ovvero dal blocco $46_{16}$ al $BF_{16}$.




La zona di memoria da usare per la coda è delimitata dal valore di due registri, ***PSTART*** (*page start*) e ***PSTOP*** (*page stop*). Seguendo l'esempio che si vede nella figura, ***PSTART*** contiene il valore $46_{16}$, mentre $PSTOP_{16}$ ha il valore $C0_{16}$.

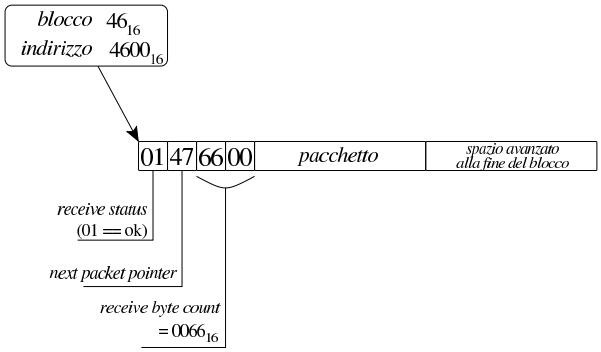
Mano a mano che la scrittura nella coda procede, viene incrementato un indice rappresentato dal registro ***CURR*** (*current page*), il quale rappresenta il prossimo blocco di memoria da utilizzare per la scrittura. Nell'esempio, il registro ***CURR*** contiene il valore $BE_{16}$, corrispondente al penultimo blocco.

Per la lettura dei blocchi si usa l'indice rappresentato dal registro ***BNRY*** (*boundary pointer*), il quale si riferisce al prossimo blocco ancora da leggere. La lettura dei blocchi si deve arrestare quando l'indice ***BNRY*** raggiunge l'indice ***CURR***; per converso, la ricezione dei pacchetti si deve arrestare quando l'indice ***CURR*** raggiunge l'indice ***BNRY***, anche se ciò comporta la perdita di pacchetti.

Quando la ricezione di un pacchetto fa sì che l'indice ***CURR*** raggiunga l'indice ***BNRY***, senza avere completato la ricezione, si ottiene uno straripamento, ma la porzione di pacchetto depositata nella coda non viene rimossa.

All'inizio di ogni pacchetto ricevuto appaiono 4 byte di intestazione, con le informazioni che si possono vedere nella figura successiva. A seconda di come avviene la lettura, l'ordine dei dati può variare: nella figura si ipotizza un accesso a byte singoli.

Figura 83.113. Esempio di blocco collocato all'indirizzo $4600_{16}$, contenente un'intestazione e un pacchetto abbastanza corto da non superare il blocco stesso. Si osservi che la dimensione riportata nel terzo e quarto byte dell'intestazione, include i quattro byte dell'intestazione stessa.




La figura ipotizza che nel blocco di memoria $46_{16}$, pari all'indirizzo $4600_{16}$, sia stato annotato un pacchetto ricevuto, i cui primi quattro byte indicano una ricezione corretta e una dimensione di $0066_{16}$ byte. Se successivamente a questo pacchetto ne è stato ricevuto un altro, questo lo si trova a partire dal blocco $47_{16}$, come annotato nel secondo byte dell'intestazione del primo.

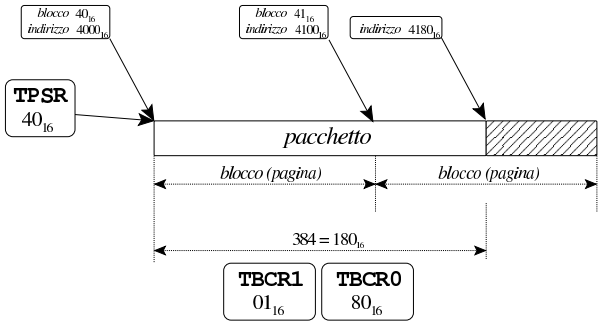
### 83.11.3 Tampone di trasmissione

Sapendo che un pacchetto può occupare al massimo sei blocchi di memoria, per la trasmissione è sufficiente riservare un'area di sei blocchi, per esempio da $4000_{16}$ a $45FF_{16}$ (estremi inclusi).

Una volta collocato il pacchetto da trasmettere nella memoria interna dell'interfaccia, occorre indicare il blocco iniziale in cui si trova il pacchetto e la sua dimensione in byte, attraverso i registri ***TPSR*** (*transmit page start*), ***TBCR0*** e ***TBCR1*** (*transit byte count*), come si vede nella figura successiva.



Figura 83.114. Esempio di blocco da trasmettere, collocato all'indirizzo $4000_{16}$, lungo 384 byte (un blocco e mezzo), con i valori da assegnare ai registri responsabili per l'invio.

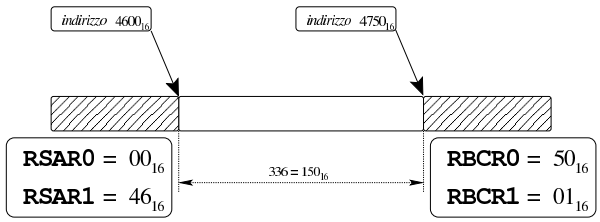



Si osservi che il pacchetto da trasmettere contiene solo ciò che serve per una trama Ethernet; pertanto, quei quattro byte di intestazione che si trovano in ricezione, non ci sono affatto in trasmissione.

### 83.11.4 Trasferimento dati tra la memoria interna e quella dell'elaboratore

I pacchetti ricevuti e quelli da trasmettere, si trovano necessariamente nella memoria interna dell'interfaccia. Per trasferire i dati tra la memoria interna a quella dell'elaboratore, occorre comunicare attraverso la porta di comunicazione $10_{16}$, più l'indirizzo relativo dell'interfaccia, ma prima va definita la posizione iniziale nella memoria interna, attraverso i registri ***RSAR0*** e ***RSAR1*** (*remote start address*), inoltre va specificata la quantità di byte da trasferire, con i registri ***RBCR0*** e ***RBCR1*** (*remote byte count*).

Figura 83.115. Lettura o scrittura di un'area di memoria interna. La variabile ***io*** dell'esempio rappresenta l'indirizzo di I/O dell'interfaccia, a cui poi si somma l'indirizzo relativo della porta di comunicazione ($10_{16}$).




*lettura*

```
for (i = 0; i < 336; i++)
  {
    buffer[i] = io_8 (0x10+io);
  }
```

*scrittura*

```
for (i = 0; i < 336; i++)
  {
    out_8 (0x10+io, buffer[i]);
  }
```

### 83.11.5 Registri e porte

Per la gestione dell'interfaccia di rete NE2000 è necessario accedere a dei registri, leggendo o scrivendo dei dati. Questi registri si raggiungono attraverso delle porte di I/O, di cui si conosce lo scostamento rispetto a un indirizzo iniziale. Per esempio, se l'interfaccia è configurata complessivamente per operare a partire dalla porta $0300_{16}$, dovendo intervenire con la porta «dati», già vista in precedenza, che si trova nell'indirizzo relativo $10_{16}$, in pratica occorre comunicare con la porta $0310_{16}$. Quando si utilizza un'interfaccia connessa a un bus PCI, si ottiene l'indirizzo della porta iniziale dal campo ***BAR0***, azzerando i due bit meno significativi (sezione 83.10).

I registri sono raggruppati in tre pagine, numerate da zero a due, ma in ogni pagina si distingue tra registri in lettura o in scrittura. Nei casi più semplici, lo stesso registro è accessibile in lettura e scrittura nella stessa pagina, come nel caso del registro ***CURR***, nella pagina uno; in altre situazioni le cose si complicano, come nel caso dei registri

***PSTART*** e ***PSTOP*** a cui si accede in scrittura nella pagina zero, oppure in lettura nella pagina due.

Dal momento che con una stessa porta di comunicazione si possono individuare registri differenti, prima occorre selezionare una pagina, attraverso un comando che si impartisce con il registro ***CR*** (*command register*), il quale ha la particolarità di essere accessibile in tutte le pagine.

Tabella 83.116. Registri NE2000.

| Scostamento | pagina 0 lettura | pagina 0 scrittura | pagina 1 lettura | pagina 1 scrittura | pagina 2 lettura | pagina 2 scrittura |
|---|---|---|---|---|---|---|
| $00_{16}$ | ***CR*** | ***CR*** | ***CR*** | ***CR*** | ***CR*** | ***CR*** |
| $01_{16}$ | ***CLDA0*** | ***PSTART*** | ***PAR0*** | ***PAR0*** | ***PSTART*** | ***CLDA0*** |
| $02_{16}$ | ***CLDA1*** | ***PSTOP*** | ***PAR1*** | ***PAR1*** | ***PSTOP*** | ***CLDA1*** |
| $03_{16}$ | ***BNRY*** | ***BNRY*** | ***PAR2*** | ***PAR2*** | -- | -- |
| $04_{16}$ | ***TSR*** | ***TPSR*** | ***PAR3*** | ***PAR3*** | ***TPSR*** | -- |
| $05_{16}$ | ***NCR*** | ***TBCR0*** | ***PAR4*** | ***PAR4*** | -- | -- |
| $06_{16}$ | ***FIFO*** | ***TBCR1*** | ***PAR5*** | ***PAR5*** | -- | -- |
| $07_{16}$ | ***ISR*** | ***ISR*** | ***CURR*** | ***CURR*** | -- | -- |
| $08_{16}$ | ***CRDA0*** | ***RSAR0*** | ***MAR0*** | ***MAR0*** | -- | -- |
| $09_{16}$ | ***CRDA1*** | ***RSAR1*** | ***MAR1*** | ***MAR1*** | -- | -- |
| $0A_{16}$ | -- | ***RBCR0*** | ***MAR2*** | ***MAR2*** | -- | -- |
| $0B_{16}$ | -- | ***RBCR1*** | ***MAR3*** | ***MAR3*** | -- | -- |
| $0C_{16}$ | ***RSR*** | ***RCR*** | ***MAR4*** | ***MAR4*** | ***RCR*** | -- |
| $0D_{16}$ | ***CNTR0*** | ***TCR*** | ***MAR5*** | ***MAR5*** | ***TCR*** | -- |
| $0E_{16}$ | ***CNTR1*** | ***DCR*** | ***MAR6*** | ***MAR6*** | ***DCR*** | -- |
| $0F_{16}$ | ***CNTR2*** | ***IMR*** | ***MAR7*** | ***MAR7*** | ***IMR*** | -- |

Tabella 83.117. Altre porte di comunicazione nelle interfacce di rete NE2000.

| Scostamento | Descrizione |
|---|---|
| $10_{16}$-$17_{16}$ | Accesso alla memoria interna (*remote DMA*). |
| $18_{16}$-$1F_{16}$ | Azzeramento. |

Figura 83.118. Sintesi dell'utilizzo del registro ***CR***, *command register*.

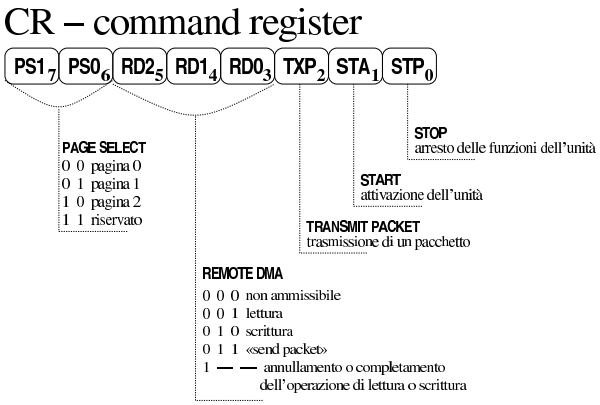





Figura 83.119. Sintesi dell'utilizzo del registro ***ISR***, *interrupt status register*.

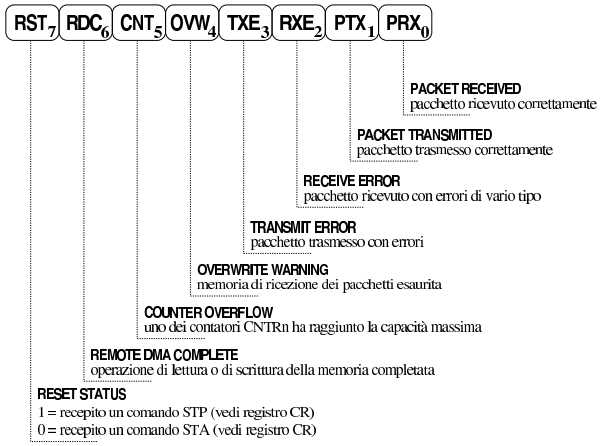



Figura 83.120. Sintesi dell'utilizzo del registro ***DCR***, *data configuration register*.

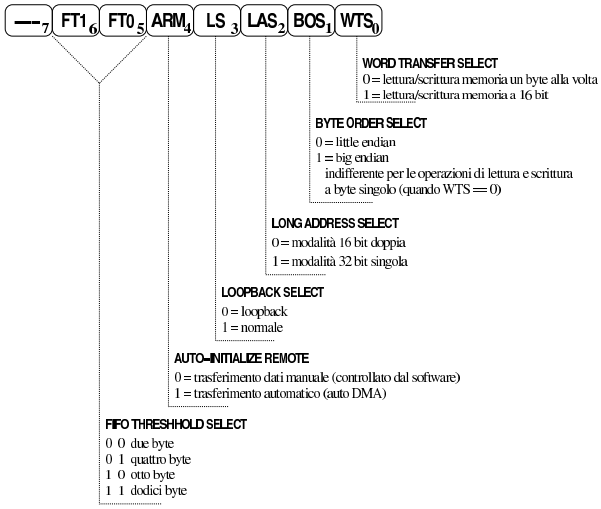



Figura 83.121. Sintesi dell'utilizzo del registro ***TCR***, *transmit configuration register*.

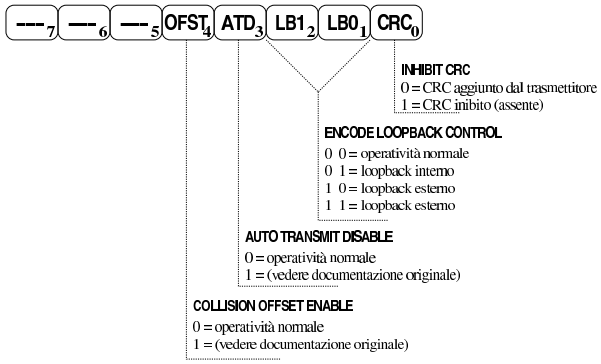


Figura 83.122. Sintesi dell'utilizzo del registro ***TSR***, *transmit status register*.

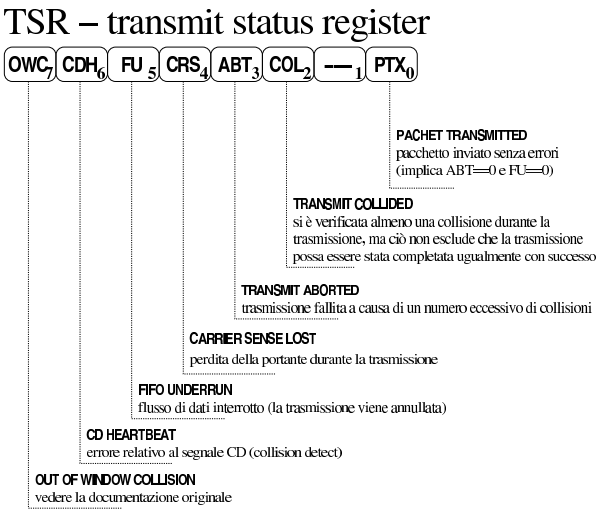



Figura 83.123. Sintesi dell'utilizzo del registro ***RCR***, *receive configuration register*.

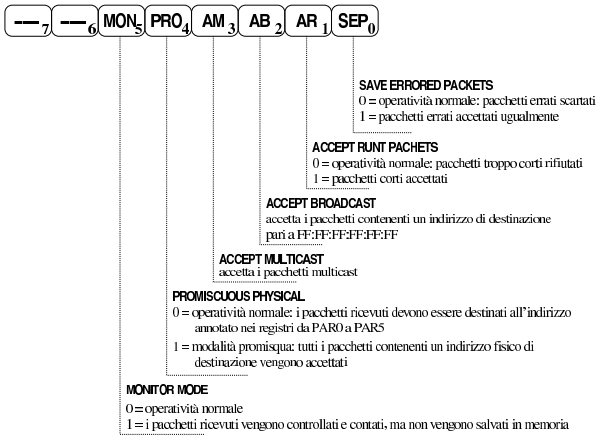
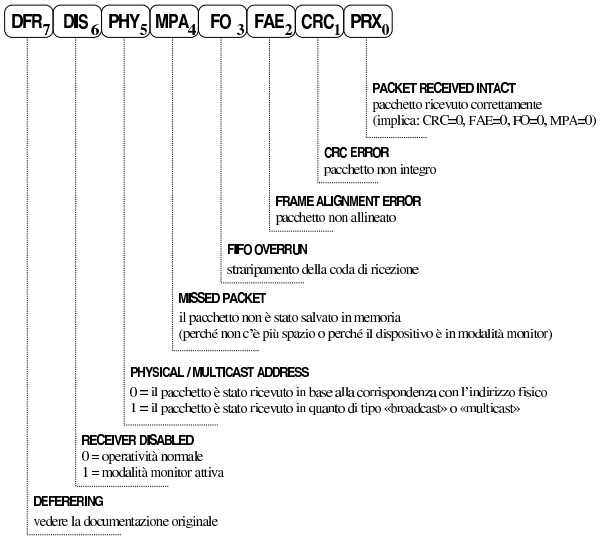



Figura 83.124. Sintesi dell'utilizzo del registro ***RSR***, *receive status register*.






### 83.11.6 Procedura di riconoscimento

Nel listato successivo si annota una procedura utile per riconoscere un'interfaccia NE2000, partendo dalla porta di I/O di partenza. Il procedimento è stato determinato leggendo il codice del kernel Linux, precisamente nel file '***linux***`/drivers/net/ne2k-pci.c`'. La variabile ***io*** rappresenta la porta di I/O iniziale, per accedere all'interfaccia; le funzioni ***in_8()*** e ***out_8()*** servono rispettivamente per leggere un byte da una porta o per scrivercelo.

```
int status;
int reg_00;
int reg_0d;
//
// Legge il registro CR (0x00), il quale non può restituire
// un valore pari a 0xFF.
//
reg_00 = in_8 (io + 0x00);
if (reg_00 == 0xFF)
  {
    return (-1);
  }
//
// Tenta di selezionare la seconda pagina di registri,
// attraverso la modifica del registro CR (0x00).
//
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
// '-------------------------------'
//   \___/   \_______/           |
//     |         |               STOP
//     |         Abort/complete
//     |         remote DMA
//     Register
//     page 1
//
out_8 ((io + 0x00), 0x61);
//
// Legge e poi modifica il contenuto del registro MAR4 (0x0D
// nella seconda pagina).
//
reg_0d = in_8 (io + 0x0D);
out_8 (io + 0x0D, 0xFF);
//
// Tenta nuovamente di selezionare la prima pagina di
// registri.
//
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
// '-------------------------------'
//   \___/   \_______/           |
//     |         |               STOP
//     |         Abort/complete
//     |         remote DMA
//     Register
//     page 0
//
out_8 ((io + 0x00), 0x21);
//
// Legge il contatore CNTR0 (0x0D).
//
in_8 (io + 0x0D);
//
// La lettura del contatore dovrebbe averlo azzerato: lo si
// rilegge per verificare questo fatto.
//
status = in_8 (io + 0x0D);
if (status)
  {
    //
    // Dal momento che il contatore non si è azzerato, non
    // può trattarsi di un'interfaccia NE2000, e molto
    // probabilmente lo scambio da una pagina all'altra di
    // registri non ha avuto effetto; per cui, viene
    // ripristinato ciò che era contenuto
    // nella posizione 0x00 e 0x0D.
    //
```

```
    out_8 (io, reg_00);
    out_8 ((io + 0x0D), reg_0d);
    return (-1);
  }
//
// Se non si sono verificati casi di errore, è plausibile
// che si tratti di un'interfaccia NE2000.
//
return (0);
```

### 83.11.7 Procedura di inizializzazione

Una volta chiarito che alla porta ***io*** risponde un'interfaccia NE2000, si può procedere con la sua inizializzazione. Durante questa fase è importante determinare l'indirizzo fisico dell'interfaccia, il quale va poi trascritto nei registri da ***PAR0*** a ***PAR5***; inoltre si stabilisce la zona della memoria interna utilizzata per i pacchetti da trasmettere e per la coda di ricezione: viene usato lo stesso schema già apparso nella figura 83.111.

```
int     status;
int     i;
uint8_t sa_prom[12];
uint8_t par[6];
//
// Viene azzerata l'interfaccia, scrivendo nella porta 0x1F.
//
status = in_8 (io + 0x1F);
out_8 ((io + 0x1F), 0xFF);
out_8 ((io + 0x1F), status);
//
// Dopo l'azzeramento occorre attendere che il registro
// ISR (0x07) segnali il recepimento di questo stato.
//
// Interrupt status register (ISR)
// .-------------------------------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-------------------------------|
// | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
// '-------------------------------'
//   |
//   Reset status
//
while (1)
  {
    if (in_8 (io + 0x07) & 0x80)                    // ISR
      {
        //
        // Il registro ISR segnala l'azzeramento avvenuto e
        // si esce dal ciclo di attesa.
        //
        break;
      }
  }
//
// Azzera tutti gli indicatori del registro ISR (0x07).
//
out_8 ((io + 0x07), 0xFF);                          // ISR
//
// Si procede con la lettura dell'indirizzo fisico,
// contenuto nella «SA-PROM» (station address PROM).
// Per prima cosa si imposta il registro CR (0x00),
// richiedendo la conclusione di ogni attività di
// trasferimento dati (in questo caso
// sarebbe superfluo, ma non può far male).
//
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
// '-------------------------------'
//  \___/ \_____/         |
//    |       |           STOP
//    |       |
//    |      Abort/complete
//    |      remote DMA
//    |
//    Register
//    page 0
//
out_8 ((io + 0x00), 0x21);                          // CR
//
// Come già avvenuto in precedenza, si attende che il
// registro ISR (0x07) confermi l'avvenuto azzeramento.
//
// Interrupt status register (ISR)
// .-------------------------------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-------------------------------|
// | 1 | ? | ? | ? | ? | ? | ? | ? | 0x80
// '-------------------------------'
//   |
//   Reset status
//
while (1)
  {
    if (in_8 (io + 0x07) & 0x80)                    // ISR
      {
        //
        // Il registro ISR segnala l'azzeramento avvenuto e
        // si azzera l'indicatore relativo, senza
        // interferire con gli altri.
        //
        out_8 ((io + 0x07), 0x80);                  // ISR
        break;
      }
  }
//
// Si procede con la configurazione necessaria per poter
// leggere dei dati dalla memoria interna, impostando il
// registro DCR (0x0E).
//
// Data configuration register (DCR)
// .-------------------------------.
// | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
// |-------------------------------|
// | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
// '-------------------------------'
//     \___/  :   |   :   :   :
//       |    :   |   :   :   Byte DMA transfer
//       |    :   |   :   :
//       |    :   |   :   Little endian byte order
//       |    :   |   :
//       |    :   |   Dual 16 bit DMA mode
//       |    :   |
//       |    :   Loopback OFF (normal operation)
//       |    :
//       |    All packets removed from
//       |    Buffer Ring under program control
//       |
//       FIFO threshold 8 bytes
//
out_8 ((io + 0x0E), 0x48);
//
// Azzera i registri RBCRn (0x0A e 0x0B) che rappresentano
// la quantità di byte da trasferire.
//
out_8 ((io + 0x0A), 0x00);                          // RBCR0
out_8 ((io + 0x0B), 0x00);                          // RBCR1
//
// Disabilita tutti gli impulsi di interruzione, azzerando
// la maschera nel registro IMR (0x0F).
//
out_8 ((io + 0x0F), 0x00);                          // IMR
//
// Azzera nuovamente tutti gli indicatori del registro
// ISR (0x07) assegnando tutti i bit a uno.
//
out_8 ((io + 0x07), 0xFF);                          // ISR
//
// Configura il registro RCR (0x0C) in modo da rimanere in
// modalità monitor (per non accumulare i pacchetti
// eventualmente ricevuti).
//
// Receive configuration register (RCR)
// .-------------------------------.
// | - | - |MON|PRO| AM| AB| AR|SEP|
// |-------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0x20
// '-------------------------------'
```

```
//          |   :   :   :   :   :
//          |   :   :   :   :   Packets with receive errors
//          |   :   :   :   :   are rejected
//          |   :   :   :   :
//          |   :   :   :   Packets with fewer than 64
//          |   :   :   :   bytes rejected
//          |   :   :   :
//          |   :   :   Packets with broadcast destination
//          |   :   :   rejected accepted
//          |   :   :
//          |   :   Packets with multicast destination
//          |   :   address not checked
//          |   :
//          |   Physical address of node must match the
//          |   station address
//          |
//          Monitor mode: packets checked but not buffered
//          to memory
//
out_8 ((io + 0x0C), 0x20);                              // RCR
//
// Configura il registro TCR (0x0D) in modo da rimanere in
// modalità «loopback» (per non trasmettere pacchetti).
//
// Transmit configuration register (TCR)
// .-------------------------------.
// | - | - | - |OFST|ATD|LB1|LB0|CRC|
// |-------------------------------|
// | 0 | 0 | 0 | 0  | 0 | 0 | 1 | 0 | 0x02
// '-------------------------------'
//                       \_____/  :
//                          |     CRC appended by the
//                          |     transmitter
//                          |
//                          Internal loopback (mode 1)
//
out_8 ((io + 0x0D), 0x02);                              // TCR
//
// Si procede finalmente indicando la quantità di byte da
// leggere dalla memoria. Dato che l'indirizzo Ethernet
// appare doppio, vanno letti 12 byte. Si devono impostare i
// registri RBCRn (0x0A e 0x0B), suddividendo la quantità
// nei due ottetti.
//
out_8 ((io + 0x0A), 12);                                // RBCR0
out_8 ((io + 0x0B), 12 >> 8));                          // RBCR1
//
// Si imposta la posizione iniziale di lettura a zero, da
// dove inizia l'informazione da leggere. Si impostano i
// registri RSARn (0x08 e 0x09).
//
out_8 ((io + 0x08), 0);                                 // RSAR0
out_8 ((io + 0x09), 0);                                 // RSAR1
//
// Si imposta il registro CR (0x00) per iniziare la lettura.
//
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// '-------------------------------'
//   \___/   \_______/       |
//     |         |           START
//     |         Read
//     |
//     Register
//     page 0
//
out_8 ((io + 0x00), 0x0A);                              // CR
//
// Attraverso la porta di comunicazione 0x10 si leggono i
// dati richiesti, salvandoli opportunamente.
//
for (i = 0; i < 12; i++)
  {
    sa_prom[i] = in_8 (io + 0x10);                      // DATA
  }
//
// Si trasferiscono i dati utili nell'array che contiene
// l'indirizzo fisico dell'interfaccia.
//
par[0] = sa_prom[0];
par[1] = sa_prom[2];
par[2] = sa_prom[4];
par[3] = sa_prom[6];
par[4] = sa_prom[8];
par[5] = sa_prom[10];
//
// Dopo l'azzeramento iniziale e l'acquisizione
// dell'indirizzo fisico, si procede finalmente con la
// sequenza di inizializzazione finale. Di nuovo si ferma
// tutto con il registro CR (0x00).
//
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0x21
// '-------------------------------'
//   \_____/ \_________/         |
//      |         |              STOP
//      |         |
//      |         Abort/complete
//      |         remote DMA
//      |
//      Register
//      page 0
//
out_8 ((io + 0x00), 0x21);                              // CR
//
// A questo punto, il registro ISR (0x07) potrebbe riportare
// lo stato di azzeramento, oppure quello di completamento
// del trasferimento dati tra la memoria interna e quella
// esterna. Tuttavia, la sua interrogazione non dovrebbe
// essere necessaria.
//
// Si imposta il registro DCR (0x0E) per una modalità di
// funzionamento normale.
//
// Data configuration register (DCR)
// .-------------------------------.
// | - |FT1|FT0|ARM| LS|LAS|BOS|WTS|
// |-------------------------------|
// | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0x48
// '-------------------------------'
//       \_____/  :   |   :   :   :
//          |     :   |   :   :   Byte DMA transfer
//          |     :   |   :   :
//          |     :   |   :   Little endian byte order
//          |     :   |   :
//          |     :   |   Dual 16 bit DMA mode
//          |     :   |
//          |     :   Loopback OFF (normal operation)
//          |     :
//          |     Send Command non executed: all packets
//          |     removed from Buffer Ring under program
//          |     control
//          |
//          FIFO threshold 8 bytes
//
out_8 ((io + 0x0E), 0x48);                              // DCR
//
// Azzera i registri RBCRn (0x0A e 0x0B).
//
out_8 ((io + 0x0A), 0);                                 // RBCR0
out_8 ((io + 0x0B), 0);                                 // RBCR1
//
// Imposta il registro RCR (0x0C) per un funzionamento
// normale.
//
// Receive configuration register (RCR)
// .-------------------------------.
// | - | - |MON|PRO| AM| AB| AR|SEP|
// |-------------------------------|
// | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0x04
// '-------------------------------'
//           :   :   :   |   :   :
//           :   :   :   |   :   Packets with receive errors
//           :   :   :   |   :   are rejected
//           :   :   :   |   :
//           :   :   :   |   Packets with less than 64 bytes
```

```
//          :   :   :   |   rejected
//          :   :   :   |
//          :   :   :   Packets with broadcast destination
//          :   :   :   address accepted
//          :   :   :
//          :   :   Packets with multicast destination
//          :   :   address not checked
//          :   :
//          :   Physical address of node must match the
//          :   station address
//          :
//          Packets buffered to memory
//
out_8 ((io + 0x0C), 0x04);                          // RCR
//
// Con il registro TPSR (0x04) configura la «pagina»
// iniziale dell'area di memoria usata per collocare i
// pacchetti da trasmettere: 0x40 (pari all'indirizzo
// 0x4000).
//
out_8 ((io + 0x04), 0x40);                          // TPSR
//
// Per il momento si lascia il trasmettitore in modalità
// «loopback».
//
// Transmit configuration register (TCR)
// .--------------------------------.
// | - | - | - |OFST|ATD|LB1|LB0|CRC|
// |--------------------------------|
// | 0 | 0 | 0 | 0  | 0 | 0 | 1 | 0 | 0x02
// '--------------------------------'
//                       \_____/  :
//                          |    CRC appended by the
//                          |    transmitter
//                          |
//                          Internal loopback (mode 1)
//
out_8 ((io + 0x0C), 0x02);                          // TCR
//
// Imposta la pagina iniziale per la coda di ricezione dei
// pacchetti, con il registro PSTART (0x01). Si indica 0x46,
// pari all'indirizzo 0x4600.
//
out_8 ((io + 0x01), 0x46);                         // PSTART
//
// Imposta l'indice di lettura, attraverso il registro BNRY
// (0x03). All'inizio questo indice coincide con la pagina
// iniziale della coda di ricezione.
//
out_8 ((io + 0x03), 0x46);                          // BNRY
//
// Imposta la pagina di memoria finale della coda di
// ricezione (precisamente si tratta della pagina successiva
// alla fine della coda), attraverso il registro PSTOP
// (0x02). Viene indicata la pagina 0xC0, pari all'indirizzo
// 0xC000.
//
out_8 ((io + 0x02), 0xC0);                          // PSTOP
//
// Utilizza il registro CR (0x00) per passare alla seconda
// pagina di registri.
//
// Command register (CR)
// .--------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |--------------------------------|
// | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0x61
// '--------------------------------'
//  \_____/ \_________/          |
//     |         |               STOP
//     |         |
//     |         Abort/complete
//     |         remote DMA
//     |
//     Register
//     page 1
//
out_8 ((io + 0x00), 0x61);                          // CR
//
// Imposta i registri PARn e MARn con l'indirizzo fisico e
// l'indirizzo multicast. I registri PARn vanno da 0x01 a
```



```
// 0x06; i registri MARn vanno da 0x08 a 0x0F.
//
out_8 ((io + 0x01), par[0]);                        // PAR0
out_8 ((io + 0x02), par[1]);                        // PAR1
out_8 ((io + 0x03), par[2]);                        // PAR2
out_8 ((io + 0x04), par[3]);                        // PAR3
out_8 ((io + 0x05), par[4]);                        // PAR4
out_8 ((io + 0x06), par[5]);                        // PAR5
//
out_8 ((io + 0x08), 0);                             // MAR0
out_8 ((io + 0x09), 0);                             // MAR1
out_8 ((io + 0x0A), 0);                             // MAR2
out_8 ((io + 0x0B), 0);                             // MAR3
out_8 ((io + 0x0C), 0);                             // MAR4
out_8 ((io + 0x0D), 0);                             // MAR5
out_8 ((io + 0x0E), 0);                             // MAR6
out_8 ((io + 0x0F), 0);                             // MAR7
//
// Imposta l'indice di scrittura nella coda di ricezione,
// per i i pacchetti ricevuti. Per questo si usa il registro
// CURR (0x07). Viene indicata la pagina di memoria iniziale
// della coda di ricezione.
//
out_8 ((io + 0x07), 0x46);                          // CURR
//
// Attraverso il registro CR (0x00) viene ripristinata la
// prima pagina di registri e viene attivata l'interfaccia.
//
// Command register (CR)
// .--------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |--------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
// '--------------------------------'
//  \_____/ \_________/      |
//     |         |           START
//     |         |
//     |         Abort/complete
//     |         remote DMA
//     |
//     Register
//     page 0
//
out_8 ((io + 0x00), 0x22);                           // CR
//
// Si azzerano tutti gli indicatori del registro ISR (0x07).
//
out_8 ((io + 0x07), 0xFF);                          // ISR
//
// A questo punto bisogna decidere se si vogliono ottenere
// dei segnali di interruzione, o meno. Nel caso si vogliano
// gestire le interruzioni, potrebbe essere conveniente
// abilitare quelle generate da un errore di trasmissione
// (0x08), dalla conclusione corretta di una trasmissione
// (0x02) e dalla ricezione corretta di un pacchetto (0x01).
// In tal caso, la maschera da adottare dovrebbe essere
// 0x0B.
//
// Tuttavia, volendo interrogare l'interfaccia in modo
// regolare, senza utilizzare le interruzioni, si può
// azzerare la maschera del registro IMR (0x0F), come in
// questo caso.
//
out_8 ((io + 0x0F), 0x00);                          // IMR
//
// Azzera il registro TCR (0x0D) per ottenere una modalità
// normale del funzionamento del trasmettitore (non più in
// «loopback»).
//
// Transmit configuration register (TCR)
// .--------------------------------.
// | - | - | - |OFST|ATD|LB1|LB0|CRC|
// |--------------------------------|
// | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0x00
// '--------------------------------'
//
out_8 ((io + 0x0D), 0x00);                          // TCR
//
// Fine del procedimento di inizializzazione.
//
return (0);
```

### 83.11.8 Trasmissione di un pacchetto

Per la trasmissione è sufficiente riservare un'area di memoria interna pari alla dimensione massima che un pacchetto singolo può occupare. In pratica si tratta di 1536 byte, pari a sei blocchi (pagine) da 256 byte. Negli esempi apparsi in precedenza, lo spazio destinato alla trasmissione è stato collocato tra $4000_{16}$ e $45FF_{16}$, estremi inclusi.

Avendo già impostato l'interfaccia come descritto nella sezione precedente, per poter trasmettere un pacchetto occorre scriverlo nell'area di memoria interna prevista e poi richiederne la trasmissione. Durante questa fase può succedere di scoprire che il trasmettitore sia già impegnato, per cui conviene rinunciare e riprovare in un momento successivo. D'altra parte, in un momento successivo alla trasmissione occorre verificare che non si sia presentato un errore nella trasmissione stessa (eventualmente a seguito della ricezione di un'interruzione, se abilitata).

Nel listato successivo, ***buffer*** è un puntatore a un'area di memoria dell'elaboratore, contenente il pacchetto da trasmettere, mentre ***size*** contiene la dimensione complessiva in byte del pacchetto stesso. La variabile ***io*** rappresenta sempre la porta di I/O iniziale, per accedere all'interfaccia.

```
int      i;
int      status;
uint8_t *b = buffer;
//
// Si legge il registro CR (0x00) per determinare se
// l'interfaccia è già in fase di trasmissione. Nel caso,
// occorre rinunciare per riprovare in un momento
// successivo.
//
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
// '-------------------------------'
//  \_____/ \_________/  |   |
//     |         |       |  Start
//     |         |       Transmit packet
//     |        Abort/complete
//     |        remote DMA
//     Register
//     page 0
//
status = in_8 (io + 0x00);                              // CR
if (status == 0x26)
  {
    errset (EBUSY);
    return (-1);
  }
//
// Si immette la dimensione da trasmettere nei registri
// RBCRn (0x0A e 0x0B).
//
out_8 ((io + 0x0A), (size & 0xFF));                     // RBCR0
out_8 ((io + 0x0B), (size >> 8));                       // RBCR1
//
// Si indica la posizione iniziale della memoria interna in
// cui depositare la copia del pacchetto da trasmettere
// (0x4000). Si utilizzano per questo i registri RSARn
// (0x08 e 0x09).
//
out_8 ((io + 0x08), 0x00);                              // RSAR0
out_8 ((io + 0x09), 0x40);                              // RSAR1
//
// Si dichiara l'intenzione di procedere con una scrittura
// nella memoria interna.
//
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0x12
// '-------------------------------'
//  \_____/ \_________/     |
//     |         |         Start
//     |         |        Write
//     |        Register
//     |        page 0
//
out_8 ((io + 0x00), 0x12);                              // CR
//
// Si procede con il trasferimento della copia del
// pacchetto, attraverso la scrittura della porta 0x10.
//
for (i = 0; i < size; i++)
  {
    out_8 ((io + 0x10), b[i]);                          // DATA
  }
//
// Si attende che il registro ISR (0x07) confermi il
// completamento dell'operazione.
//
// Interrupt status register (ISR)
// .-------------------------------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-------------------------------|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// '-------------------------------'
//       |
//       Remote DMA complete
//
while (1)
  {
    if (in_8 (io + 0x07) & 0x40)                        // ISR
      {
        //
        // Il registro ISR dà la conferma e se ne azzera
        // l'indicatore relativo, senza interferire con gli
        // altri.
        //
        out_8 ((io + 0x07), 0x40);                      // ISR
        break;
      }
  }
//
// Si dichiara il blocco di memoria interna (pagina) in cui
// è contenuto il pacchetto da inviare. Si imposta il
// registro TPSR (0x04) con il valore 0x40, corrispondente
// al blocco che inizia all'indirizzo 0x4000.
//
out_8 (io + 0x04, 0x40);                                // TPSR
//
// Si dichiara la quantità di byte da trasmettere,
// utilizzando i registri TBCRn (0x05 e 0x06).
//
out_8 ((io + 0x05), (size & 0xFF));                     // TBCR0
out_8 ((io + 0x06), (size >> 8));                       // TBCR1
//
// Finalmente si trasmette il pacchetto.
//
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0x26
// '-------------------------------'
//  \_____/ \_________/  |   |
//     |         |       |  Start
//     |         |       Transmit packet
//     |        Abort/complete remote DMA
//     Register
//     page 0
//
out_8 ((io + 0x00), 0x26);                              // CR
//
// Si attende che il pacchetto sia stato trasmesso o che sia
// riportato un errore.
//
// Interrupt status register (ISR)
// .-------------------------------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-------------------------------|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// '-------------------------------'
//                   |       |
```

```
//                  |       Packet transmitted with no
//                  |       errors
//                  Transmit error
//
while (1)
  {
    if (in_8 (io + 0x07) & 0x0A)                    // ISR
      {
        //
        // Azzeramento degli indicatori previsti.
        //
        out_8 ((io + 0x07), 0x0A);                  // ISR
        break;
      }
  }
//
// Si verifica nel registro TSR (0x04) l'esito della
// trasmissione.
//
// Transmit status (TSR)
// .-----------------------------.
// |OWC|CDH| FU|CRS|ABT|COL| - |PTX|
// |-----------------------------|
// | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0x38
// '-----------------------------'
//           |   |   |
//           |   |   Transmit aborted
//           |   Carrier sense lost
//           FIFO underrun
//
status = in_8 (io + NE2K_TSR);
if (status & 0x38)
  {
    errset (EIO);
    return (-1);
  }
//
// Fine
//
return (0);
```

### 83.11.9 Ricezione

Alla ricezione dei pacchetti (trame) provvede l'interfaccia, ammesso di avere configurato tutto opportunamente, come mostrato in precedenza, sapendo che il registro ***CURR*** indica il blocco (la pagina) della memoria interna in cui va collocato il prossimo pacchetto ricevuto. Per il prelievo di questi dati dalla memoria interna, si utilizza il registro ***BNRY***, il quale rappresenta il blocco di memoria ancora da leggere. Sapendo che inizialmente i registri ***BNRY*** e ***CURR*** puntano entrambi al blocco iniziale di memoria interna destinato ad accogliere i dati ricevuti, il valore contenuto nel registro ***BNRY*** non può superare ***CURR***. Quando però la ricezione procede velocemente, più di quanto si provveda a estrarre i pacchetti, il registro ***CURR*** può raggiungere di nuovo ***BNRY***, ma in tal caso si ottiene uno straripamento che deve essere gestito in qualche modo.

Secondo l'organizzazione prevista in precedenza, la porzione di memoria interna destinata alla ricezione dei pacchetti va da $4600_{16}$ a $BFFF_{16}$, inclusi.

Nell'esempio del listato seguente, come già in quelli precedenti, la variabile ***io*** rappresenta la porta di I/O iniziale, per accedere all'interfaccia; inoltre, ***destination*** è un puntatore all'area di memoria in cui va collocato un pacchetto; tale puntatore si ottiene attraverso una funzione, denominata ***new_frame()***.

```
int   i;
int   bytes;
int   curr;
int   bnry;
int   next;
int   frame_status;
int   frame_size;
int   status;
char *destination;
//
// Si seleziona la seconda pagina di registri, per poter
// accedere poi al registro CURR.
//
// Command register (CR) 0x00
// .-----------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----------------------------|
// | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0x62
// '-----------------------------'
//  \_____/ \_________/     |
//     |         |         START
//     |         |
//     |         Abort/complete remote DMA
//     |
//     Register page 1
//
out_8 ((io + 0x00), 0x62);                          // CR
//
// Legge la posizione corrente dell'indice di scrittura
// CURR (0x07).
//
curr = in_8 (io + 0x07);                            // CURR
//
// Si seleziona nuovamente la prima pagina di registri.
//
// Command register (CR) 0x00
// .-----------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-----------------------------|
// | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0x22
// '-----------------------------'
//  \_____/ \_________/     |
//     |         |         START
//     |         |
//     |         Abort/complete remote DMA
//     |
//     Register page 0
//
out_8 ((io + 0x00), 0x22);                          // CR
//
// Si legge il valore contenuto nel registro BNRY (0x03).
//
bnry = in_8 (io + 0x03);                            // BNRY
//
// Si parte dal presupposto che ci sia almeno un pacchetto
// da leggere; pertanto, se anche i registri CURR e BNRY
// fossero uguali, significherebbe solo che tutta la memoria
// interna destinata alla ricezione, richiede di essere letta.
//
// Si passa al prelievo di tutti i pacchetti pronti per il
// prelievo nell'area di memoria interna.
//
while (1)
  {
    //
    // Trova un posto dove mettere un pacchetto ricevuto.
    //
    destination = new_frame ();
    //
    // Ci si prepara a leggere i primi quattro byte,
    // iniziando dal blocco di memoria a cui si riferisce la
    // variabile 'bnry'.
    //
    out_8 ((io + NE2K_RBCR0), 4);
    out_8 ((io + NE2K_RBCR1), 0);
    //
    out_8 ((io + NE2K_RSAR0), 0); // Deve essere zero!
    out_8 ((io + NE2K_RSAR1), bnry);
    //
    // Si predispone il registro CR (0x00) per la lettura
    // della memoria interna.
    //
    // Command register (CR)
    // .-----------------------------.
    // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
    // |-----------------------------|
    // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
    // '-----------------------------'
    //  \_____/ \_________/     |
    //     |         |         START
    //     |         |
    //     |         Read
    //     |
```

```
//     Register page 0
//
out_8 (io + 0x00, 0x1A);                          // CR
//
// Si leggono e si salvano le informazioni ottenute. La
// lettura si esegue dalla porta di comunicazione
// destinata alla trasferimento di dati con la memoria
// interna.
//
frame_status = in_8 (io + 0x10);         // frame status
next         = in_8 (io + 0x10);         // next frame
frame_size   = in_8 (io + 0x10);         // frame size
                                         // low
frame_size  += (in_8 (io + 0x10) * 256); // frame size
                                         // high
//
// Si attende che il registro ISR (0x07) confermi il
// completamento dell'operazione di lettura.
//
// Interrupt status register (ISR)
// .-------------------------------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-------------------------------|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// '-------------------------------'
//       |
//       Remote DMA complete
//
while (1)
  {
    if (in_8 (io + 0x07) & 0x40)                  // ISR
      {
        //
        // Azzeramento degli indicatori previsti.
        //
        out_8 ((io + 0x07), 0x40);                // ISR
        break;
      }
  }
//
// Adesso si sa quanto è grande il pacchetto da leggere,
// ma si deve ripetere la lettura a cominciare dai primi
// quattro byte di intestazione, anche se questi non
// fanno parte del pacchetto (la trama Ethernet).
//
if (next > bnry)
  {
    bytes = frame_size;
  }
//
if (next < bnry)
  {
    //
    // Se il pacchetto successivo si trova in un blocco
    // antecedente alla posizione iniziale di lettura,
    // significa che quello attuale si estende oltre la
    // fine dell'area di memoria (riprendendo all'inizio
    // della stessa), pertanto in questo caso occorre
    // procedere con due fasi di lettura. Il valore 0xC0
    // rappresenta il blocco di memoria successivo alla
    // conclusione dell'area di memoria destinata alla
    // ricezione. La quantità di blocchi da leggere
    // viene moltiplicata per 256, essendo questa la
    // dimensione di un blocco.
    //
    bytes = ((0xC0 - bnry) * 256);
    bytes = min (bytes, frame_size);
  }
//
// Prima fase di lettura. Si ricorda che la lettura
// riparte dall'inizio, pertanto vanno poi eliminati i
// primi quattro byte di intestazione.
//
out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
out_8 ((io + NE2K_RBCR1), bytes >> 8);
//
out_8 ((io + NE2K_RSAR0), 0);     // MUST be zero. :-(
out_8 ((io + NE2K_RSAR1), bnry);
//
// Si imposta il registro CR (0x00) per la lettura.
//
```



```
// Command register (CR)
// .-------------------------------.
// |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
// |-------------------------------|
// | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
// '-------------------------------'
//  \_____/ \_________/      |
//     |         |          START
//     |         |
//     |         Read
//     |
//     Register page 0
//
out_8 (io + 0x00, 0x0A);                           // CR
//
// Legge e scarta i primi quattro byte di intestazione.
//
in_8 (io + 0x10);                                 // DATA
in_8 (io + 0x10);                                 // DATA
in_8 (io + 0x10);                                 // DATA
in_8 (io + 0x10);                                 // DATA
//
// Riduce la quantità di byte ancora da leggere di
// conseguenza.
//
bytes -= 4;
//
// Passa alla lettura del pacchetto.
//
i = 0;
for (; bytes > 0; i++, bytes--)
  {
    destination[i] = in_8 (io + 0x10);            // DATA
  }
//
// Si attende la conclusione dell'operazione,
// controllando l'esito nel registro ISR (0x07).
//
// Interrupt status register (ISR)
// .-------------------------------.
// |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
// |-------------------------------|
// | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
// '-------------------------------'
//       |
//       Remote DMA complete
//
while (1)
  {
    if (in_8 (io + 0x07) & 0x40)                  // ISR
      {
        //
        // Azzeramento degli indicatori previsti.
        //
        out_8 ((io + 0x07), 0x40);                // ISR
        break;
      }
  }
//
if (next < bnry)
  {
    //
    // Dato che il pacchetto successivo si trova in un
    // blocco di memoria antecedente all'inizio di
    // quello letto, potrebbe esserci una seconda parte
    // da leggere.
    // Se il calcolo successivo da un risultato maggiore
    // di zero, c'è effettivamente ancora da leggere, a
    // partire dal blocco di memoria 0x46.
    //
    bytes = frame_size - ((0xC0 - bnry) * 256);
  }
//
if (bytes > 0)
  {
    //
    out_8 ((io + NE2K_RBCR0), bytes & 0xFF);
    out_8 ((io + NE2K_RBCR1), bytes >> 8);
    //
    out_8 ((io + NE2K_RSAR0), 0);
    out_8 ((io + NE2K_RSAR1), 0x46);
```

```
        //
        // Command register (CR)
        // .-------------------------------.
        // |PS1|PS0|RD2|RD1|RD0|TXP|STA|STP|
        // |-------------------------------|
        // | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0x0A
        // '-------------------------------'
        //  \_____/ \_________/     |
        //     |         |         START
        //     |         |
        //     |        Read
        //     |
        //    Register page 0
        //
        out_8 (io + 0x00, 0x0A);                        // CR
        //
        for (; bytes > 0; i++, bytes--)
          {
            destination[i] = in_8 (io + 0x10);        // DATA
          }
        //
        // Interrupt status register (ISR)
        // .-------------------------------.
        // |RST|RDC|CNT|OVW|TXE|RXE|PTX|PRX|
        // |-------------------------------|
        // | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0x40
        // '-------------------------------'
        //       |
        //      Remote DMA complete
        //
        while (1)
          {
            if (in_8 (io + 0x07) & 0x40)              // ISR
              {
                //
                // Azzeramento degli indicatori previsti.
                //
                out_8 ((io + 0x07), 0x40);            // ISR
                break;
              }
          }
      }
    //
    // Essendosi conclusa la lettura del pacchetto, si aggiorna BNR
    // 0x03.
    //
    bnry = next;
    out_8 (io + 0x03, bnry);                          // BNRY
    //
    // Se il valore attuale di BNRY coincide con quello di
    // CURR, il ciclo termina.
    //
    if (bnry == curr)
      {
        //
        // Fine.
        //
        return;
      }
  }
```

## 83.12 IPv4 in una rete Ethernet

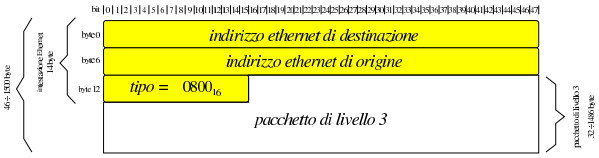
Il protocollo IPv4 è ormai superato, ma rimane ancora in uso e, per la sua semplicità, si presta meglio a un primo approccio alla gestione dei protocolli TCP/IP.

Negli schemi delle figure che appaiono in questa sezione, si intende che i dati siano ordinati secondo il così detto *network byte order*, ossia come sequenza di byte, da sinistra verso destra; per la stessa ragione, i bit sono numerati partendo dal bit più significativo in giù. Per converso, negli esempi di strutture in linguaggio C, usati per rappresentare i dati contenuti nei pacchetti dei protocolli, si intende di operare in un'architettura di tipo *little endian*

I pacchetti del protocollo Ethernet hanno l'intestazione descritta dalla figura successiva. Ciò che segue tale intestazione è poi il pacchetto dal punto di vista del protocollo di rete.



Figura 83.129. PDU di livello 2, nelle reti Ethernet.




### 83.12.1 ARP

Il protocollo ARP permette di risalire all'indirizzo fisico, partendo da quello di rete. In una rete Ethernet, gli indirizzi fisici occupano 48 bit, ovvero 6 byte. I pacchetti del protocollo ARP hanno la struttura evidenziata dalla figura successiva.

Figura 83.130. Struttura di un pacchetto del protocollo ARP, relativo a una rete fisica Ethernet e al protocollo IPv4.

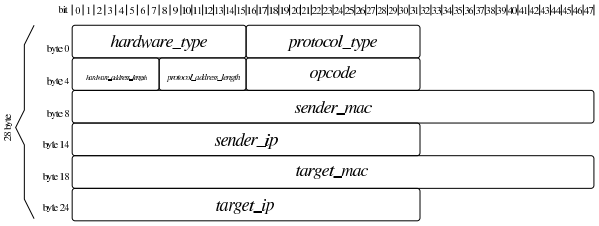



Tabella 83.131. Campi di un pacchetto del protocollo ARP, relativo a una rete fisica Ethernet e al protocollo IPv4.

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***hardware_type*** | 16 bit | Descrive il tipo di rete fisica; per una rete Ethernet si usa il codice $0001_{16}$. |
| ***protocol_type*** | 16 bit | Descrive il tipo di indirizzamento a livello 3 del modello ISO-OSI; per il protocollo IPv4 si usa il codice $0800_{16}$. |
| ***hardware_address_length*** | 8 bit | Dichiara la dimensione dell'indirizzo hardware in ottetti (byte); nel caso del protocollo Ethernet, l'indirizzo occupa 6 ottetti. |
| ***protocol_address_length*** | 8 bit | Dichiara la dimensione dell'indirizzo di rete in ottetti (byte); nel caso del protocollo IPv4, l'indirizzo occupa 4 ottetti. |
| ***opcode*** | 16 bit | Dichiara il tipo di operazione richiesta; il valore $0001_{16}$ indica la richiesta di conoscere l'indirizzo fisico corrispondente a un certo indirizzo di rete; il valore $0002_{16}$ indica la risposta con l'informazione richiesta. |
| ***sender_mac*** | variabile | Contiene l'indirizzo fisico di origine. |
| ***sender_ip*** | variabile | Contiene l'indirizzo di rete di origine. |
| ***target_mac*** | variabile | Contiene l'indirizzo fisico di destinazione. |
| ***target_ip*** | variabile | Contiene l'indirizzo di rete di destinazione. |

Listato 83.132. Esempio di struttura per descrivere un pacchetto del protocollo ARP, destinato a essere usato in una rete Ethernet. L'ultimo campo, denominato qui ***filler***, serve a far sì che il pacchetto raggiunga almeno la dimensione minima richiesta dal protocollo Ethernet.

```
typedef struct {
  uint16_t hardware_type;
  uint16_t protocol_type;
  uint8_t  hardware_address_length;             // 6 byte
  uint8_t  protocol_address_length;             // 4 byte
  uint16_t opcode;
  uint8_t  sender_mac[6];
  uint32_t sender_ip;  // Network byte order.
  uint8_t  target_mac[6];
  uint32_t target_ip;  // Network byte order.
  uint8_t  filler[4];
} __attribute__ ((packed)) arp_packet_t;
```

L'utilizzo più comune del protocollo prevede l'invio di un pacchetto di richiesta, per conoscere l'indirizzo fisico di un certo indirizzo di rete, per il quale ci si aspetta di ottenere un pacchetto di risposta, contenente l'informazione richiesta, dal nodo che utilizza effettivamente quell'indirizzo di rete cercato.

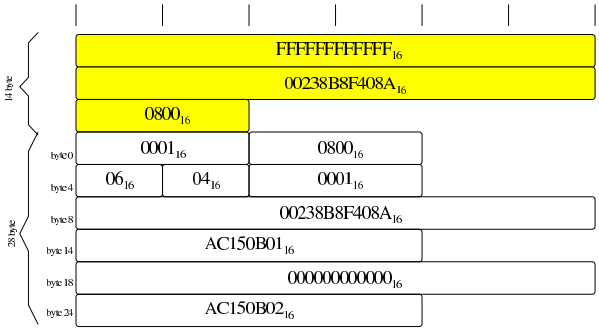
Va però osservato che il protocollo ARP serve a collegare il protocollo fisico con quello di rete; pertanto, per l'analisi dei pacchetti ARP occorre considerare anche il loro involucro a livello fisico.

A titolo di esempio, si parte dalla situazione schematizzata dalla figura successiva, dove il nodo «A» cerca di contattare il nodo «B», ma per farlo deve conoscerne l'indirizzo fisico, attraverso l'ausilio del protocollo ARP.

Figura 83.133. Situazione ipotetica di due nodi che utilizzano il protocollo ARP.


Il nodo «A» invia un pacchetto di richiesta nella rete fisica locale. Questo pacchetto deve essere diretto a tutti i nodi fisici raggiungibili, pertanto è contenuto in un pacchetto Ethernet «circolare», ovvero *broadcast*.

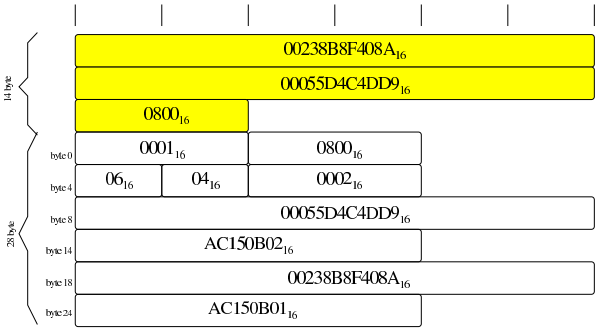
Figura 83.134. Pacchetto di richiesta ARP per conoscere l'indirizzo fisico del nodo «B» (172.21.11.2). I primi 14 byte rappresentano l'involucro Ethernet, nel quale si può osservare che la destinazione è indefinita, utilizzando un indirizzo *broadcast* (ff:ff:ff:ff:ff:ff).




Quando il nodo «B» intercetta il pacchetto di richiesta ARP, nota che l'indirizzo IPv4 contenuto riguarda la sua interfaccia di rete, pertanto risponde con un altro pacchetto ARP, ma in tal caso la destinazione è precisa, perché conosciuta dal pacchetto di richiesta.



Figura 83.135. Pacchetto di risposta ARP per comunicare l'indirizzo fisico del nodo «B» (172.21.11.2) al nodo «A» (172.21.11.1). I primi 14 byte rappresentano l'involucro Ethernet.




## 83.12.2 IPv4

Il protocollo IPv4 si colloca al primo dei livelli interessati dal TCP/IP, mentre nel modello ISO/OSI si tratta del terzo livello, essendo un protocollo di rete. Il protocollo IP utilizza degli indirizzi propri per individuare i vari nodi con cui avviene la comunicazione; tuttavia, questi indirizzi, a livello di rete, vanno tradotti in indirizzi fisici per raggiungere effettivamente la destinazione, cosa che di norma viene gestita con l'ausilio del protocollo ARP, come descritto nella sezione precedente.

L'intestazione di un pacchetto IPv4 ha delle componenti che possono essere piuttosto complesse; in particolare possono essere previste delle opzioni che allungano in modo variabile questa intestazione. Tuttavia, qui si presume di non gestire mai tali opzioni e di ignorarle semplicemente se contenute nei pacchetti che si ricevono.

Figura 83.136. Struttura di un pacchetto del protocollo IPv4.

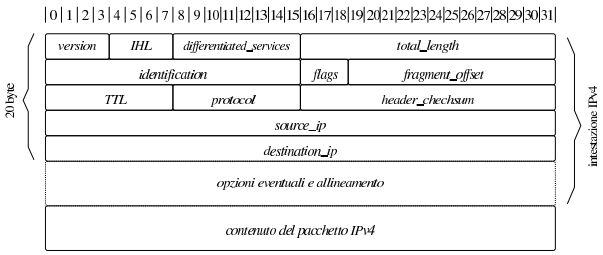



Tabella 83.137. Campi dell'intestazione di un pacchetto del protocollo IPv4. Le opzioni, se presenti, devono occupare uno spazio multiplo di 32 bit, riempiendo eventualmente i bit mancanti con valori a zero.

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***version*** | 4 bit | Descrive la versione del pacchetto a livello del protocollo di rete; per IPv4 si tratta del valore 4. |
| ***IHL*** (*internet header length*) | 4 bit | Dichiara la lunghezza dell'intestazione del pacchetto IPv4, in multipli di 32 bit. Dal momento che l'intestazione ha una parte iniziale fissa di 20 ottetti (byte), il valore minimo che può assumere questo campo è pari a 5 e aumenta in presenza di opzioni. |
| ***differentiated_services*** | 8 bit | Si tratta di quello che originariamente era noto con la sigla «TOS» (*type of service*). In generale, per un pacchetto «normale», questo campo ha il valore zero. |
| ***total_length*** | 16 bit | Dichiara la lunghezza complessiva del pacchetto IPv4, intestazione inclusa. |

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***identification*** | 16 bit | Numero di identificazione del pacchetto IPv4. Nel caso di pacchetti IP frammentati, questo numero deve rimanere lo stesso per tutti i frammenti che compongono lo stesso pacchetto complessivo. |
| ***flags*** | 3 bit | Bit indicatori. Nell'ambito di questo gruppo di tre bit, il $bit_0$ (ovvero quello meno significativo) è noto con la sigla ***MF*** (*more fragments*) e, se attivo, indica che il pacchetto attuale è composto da altri frammenti successivi. Il $bit_1$ è noto con la sigla ***DF*** (*don't fragment*) e, se attivo, indica che il pacchetto non può essere frammentato. Il $bit_2$ è riservato. |
| ***fragment_offset*** | 13 bit | Questo campo viene usato quando si tratta di un frammento di un pacchetto IPv4 più grande, per indicare che l'inizio del contenuto di questo frammento va collocato a partire dallo scostamento indicato. In un pacchetto non frammentato il contenuto di questo campo è pari a zero. |
| ***TTL*** | 8 bit | Il valore di questo campo, noto come *time do live*, viene stabilito nel momento della sua trasmissione e decrementato di una unità all'attraversamento di ogni router; se questo valore raggiunge lo zero, il pacchetto viene scartato. |
| ***protocol*** | 8 bit | Il contenuto di un pacchetto IPv4 riguarda un protocollo di trasporto (livello 4 del modello ISO/OSI); questo campo indica di che protocollo si tratta. In particolare:<br>$01_{16}$ = ICMP;<br>$06_{16}$ = TCP;<br>$11_{16}$ = UDP. |
| ***header_checksum*** | 16 bit | Codice di controllo calcolato sull'intestazione IPv4, opzioni incluse. |
| ***source_ip*** | 32 bit | Indirizzo IPv4 di origine. |
| ***destination_ip*** | 32 bit | Indirizzo IPv4 di destinazione. |

Listato 83.138. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo IPv4. A sinistra nella versione *little endian*; a destra in quella *big endian*. Il campo ***frag_off*** include qui i bit ***DF*** e ***MF***.

*little endian*

```
struct iphdr
{
  uint16_t  ihl     : 4,
            version : 4;
  uint8_t   tos;
  uint16_t  tot_len;
  uint16_t  id;
  uint16_t  frag_off;
  uint8_t   ttl;
  uint8_t   protocol;
  uint16_t  check;
  uint32_t  saddr;
  uint32_t  daddr;
};
```

*big endian*

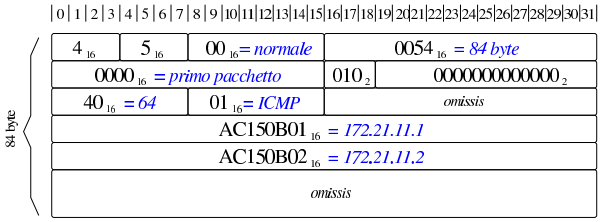
```
struct iphdr
{
  uint16_t  version : 4,
            ihl     : 4;
  uint8_t   tos;
  uint16_t  tot_len;
  uint16_t  id;
  uint16_t  frag_off;
  uint8_t   ttl;
  uint8_t   protocol;
  uint16_t  check;
  uint32_t  saddr;
  uint32_t  daddr;
};
```

A titolo di esempio si analizza l'intestazione di un pacchetto IPv4 relativo all'invio di un «ping», tra il nodo «A» e il nodo «B», della figura successiva.



Figura 83.139. Situazione ipotetica di due nodi che utilizzano il protocollo IPv4.


Figura 83.140. Esempio di pacchetto «ping», inviato dall'indirizzo 172.21.11.1 a 172.21.11.2. Il pacchetto IPv4 non è frammentato (***MF*** == 0) e non è nemmeno frammentabile (***DF*** == 1).




Il codice di controllo che nell'esempio è stato omesso, viene calcolato sul contenuto dell'intestazione, considerando inizialmente che al posto del codice di controllo ci siano solo bit a zero. Il modo in cui questo viene calcolato è descritto nella sezione successiva; va tenuto conto, inoltre, che una volta calcolato questo viene collocato nell'intestazione invertendo i suoi bit (facendone il complemento a uno, ovvero applicando l'operatore binario NOT). Così facendo, per controllare la validità dell'intestazione, è sufficiente ripetere il calcolo del codice di controllo, utilizzando però questa volta anche quanto contenuto nel campo ***header_checksum***, e verificando che il risultato sia pari a $FFFF_{16}$, oppure $0000_{16}$.

È interessante osservare che, ogni volta che il campo ***TTL*** viene modificato da un router, questo deve provvedere ad aggiornare il codice di controllo dell'intestazione.

### 83.12.3 Il codice di controllo del TCP/IP

Il codice di controllo usato nell'intestazione dei pacchetti IPv4 e anche in altre situazioni, è calcolato suddividendo l'informazione di partenza in blocchi da 16 bit e sommando assieme questi blocchi, in modo binario, usando però l'aritmetica del complemento a uno.

Usando il sistema del complemento a uno, i numeri interi positivi si rappresentano in binario come di consueto, purché il bit più significativo sia pari a zero, mentre i numeri negativi sono rappresentati con il loro complemento a uno. Per esempio, disponendo di otto bit, il numero +5 si rappresenta come $00000101_2$, mentre il numero -5 diventa $11111010_2$. Pertanto, si distingue tra uno zero positivo ($00000000_2$) e uno zero negativo ($11111111_2$). Si osservino gli esempi seguenti:

```
+5 +  00000101 +    +5 +  00000101 +     +5 +  00000101 +
+2 =  00000010 =    -5 =  11111010 =     -7 =  11111000 =
----  --------      ----  --------       ----  --------
+7    00000111       0    11111111 (-0)  -2    11111101
```

Va però fatta attenzione ai riporti, perché questi vanno sommati al risultato:

```
+5 +    00000101 +
-3 =    11111100 =
----    --------
+2     100000001   (si ottiene un riporto)

        00000001 +
               1 = (si somma il riporto)
        --------
        00000010   (questo è il risultato corretto)
```

Se si esegue una somma di più valori, i riporti si possono sommare tutti alla fine, senza farlo necessariamente a ogni coppia:

```
+7 +    00000111 +
-2 +    11111101 +
-3 +    11111100 =
----    --------
+2    1000000000   (si ottiene un riporto)

        00000000 +
              10 = (si somma il riporto)
        --------
        00000010   (questo è il risultato corretto)
```

Per fare questo tipo di somma in un'architettura che utilizza l'aritmetica del complemento a due, è sufficiente utilizzare una variabile intera senza segno, di rango maggiore rispetto ai blocchi sommati, quindi si separano i riporti dal risultato per poi sommarli nuovamente a quello. Per esempio, nel caso del codice di controllo necessario ai protocolli TCP/IP, si utilizza una variabile intera, senza segno, a 32 bit. Si somma tutto quello che serve, quindi alla fine si separa il risultato contenuto nei 16 bit meno significativi, per sommargli i riporti contenuti nei 16 bit più significativi.

Nel listato successivo si vede come può essere realizzata una funzione per calcolare un codice di controllo relativo al contenuto di memoria che parte dalla posizione ***data*** e si estende per ***size*** byte. Nel procedimento va osservato il fatto che in memoria i dati si intendono essere ordinati nel modo naturale relativo alle comunicazioni di rete (*network byte order*); pertanto, nel calcolo viene usata la funzione ***ntohs()*** (*network to host short*) per garantire che i blocchi da 16 bit siano interpretati correttamente. Inoltre, dal momento che i dati su cui calcolare il codice di controllo potrebbero essere composti da una quantità dispari di byte, l'ultimo ottetto viene trattato come se rappresentasse gli otto bit più significativi di un blocco di sedici.

Listato 83.144. Esempio di funzione per il calcolo della codice di controllo usato nei protocolli TCP/IP. La funzione ***ntohs()*** serve a garantire che i blocchi da 16 bit vengano interpretati nell'ordine giusto.

```
#include <stdint.h>
#include <arpa/inet.h>
uint16_t
checksum_tcpip (uint16_t *data, size_t size)
{
  int      i;
  uint32_t sum;
  uint16_t carry;
  uint16_t checksum;
  uint16_t last;
  uint8_t *octet;
  //
  // Somma
  //
  sum = 0;
  //
  for (i = 0; i < (size/2); i++)
    {
      sum += ntohs (data[i]);
    }
  //
  if (size % 2)
    {
      //
      // La dimensione è dispari, pertanto va considerato
      // anche l'ultimo ottetto.
      //
      octet = (uint8_t *) data;
      last = octet[size-1];
      last = last << 8;
      sum += last;
    }
  //
  // Riporti
  //
  carry     = sum >> 16;
  checksum  = sum & 0x0000FFFF;
  checksum += carry;
  //
  //
  return (checksum);
}
```



### 83.12.4 ICMP

Il protocollo ICMP si colloca al di sopra di quello IP, per l'invio di messaggi elementari, composti da un numero di messaggio (più precisamente si tratta di tipo e codice) con qualche informazione allegata. Il protocollo ICMP è molto importante per segnalare il fatto che un certo nodo non può essere raggiunto, ma spesso si usa per provare il funzionamento della rete con l'invio di una richiesta di eco, per la quale si attende una risposta equivalente.

Un pacchetto ICMP si inserisce all'interno di un pacchetto IP e si scompone come si vede nella figura successiva.

Figura 83.145. Struttura comune di un pacchetto del protocollo ICMP (all'interno di IPv4).

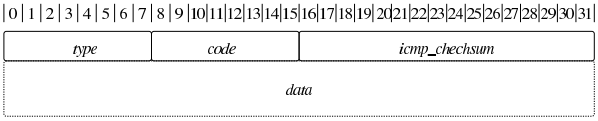



Tabella 83.146. Campi dell'intestazione comune di un pacchetto del protocollo ICMP.

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***type*** | 8 bit | Numero del tipo di messaggio:<br>$01_{16}$ *echo reply*<br>$03_{16}$ *destination unreachable*<br>$08_{16}$ *echo request* |
| ***code*** | 8 bit | Qualificazione ulteriore del tipo di messaggio; di norma è pari a zero. |
| ***icmp_checksum*** | 16 bit | Codice di controllo, calcolato sul pacchetto ICMP, a partire dal campo ***type***, fino alla fine del pacchetto. |
| ***data*** | variabile | Contenuto del pacchetto ICMP che varia a seconda del tipo; in ogni caso, la dimensione massima dipende dalla dimensione massima di un pacchetto IPv4 non frammentato. |

Tuttavia, il contenuto di un pacchetto ICMP può avere un'intestazione ulteriore, a seconda del tipo dichiarato nel campo ***type***.

Figura 83.147. Struttura di un pacchetto del protocollo ICMP, di tipo *echo request* o *echo reply*.

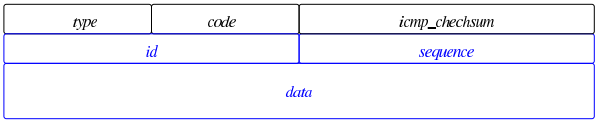



Tabella 83.148. Campi specifici dei pacchetti ICMP di eco.

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***id*** | 16 bit | Numero identificativo di una sequenza; nella richiesta di eco viene attribuito in modo casuale dal programma '**`ping`**' o da ciò che ne svolge la funzione. |
| ***sequence*** | 16 bit | Numero di sequenza del pacchetto: la risposta a una richiesta di eco viene fatta usando lo stesso numero di identificazione e lo stesso numero di sequenza. |

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***data*** | variabile | Lo spazio rimanente nel pacchetto viene attribuito dal programma che richiede l'eco, in base alle esigenze; spesso si usano sequenze casuali di byte, per verificare il funzionamento della rete, dal momento che il pacchetto di risposta all'eco deve poi contenere gli stessi dati, confrontabili nell'origine. |

Listato 83.149. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo ICMP. Questa struttura vale indifferentemente per le architetture *little endian* e *big endian*.

```
struct icmphdr
{
  uint8_t  type;
  uint8_t  code;
  uint16_t checksum;
  union
  {
    struct
    {
      uint16_t id;
      uint16_t sequence;
    } __attribute__ ((packed)) echo;    // echo datagram
    uint32_t gateway;                   // gateway address
    struct
    {
      uint16_t unused;
      uint16_t mtu;
    } __attribute__ ((packed)) frag;    // path mtu discovery
  } un;
} __attribute__ ((packed));
```

A titolo di esempio si considerano due nodi, come nella figura successiva, e si analizza il contenuto di un pacchetto ICMP di richiesta di eco, da «A» a «B», seguito da una risposta conforme, in senso opposto.

Figura 83.150. Il nodo «A» invia una richiesta di eco al nodo «B» e il nodo «B» risponde.


Figura 83.151. Esempio di pacchetto ICMP, di tipo *echo request* (ping), inviato dall'indirizzo 172.21.11.15 a 172.21.254.254, completo dell'intestazione IPv4.

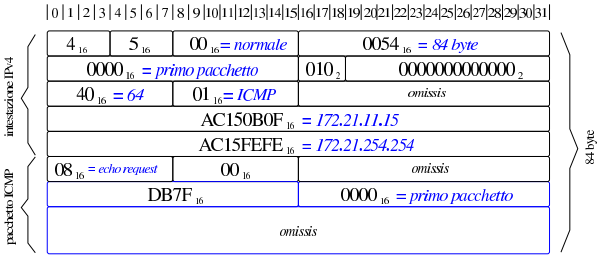
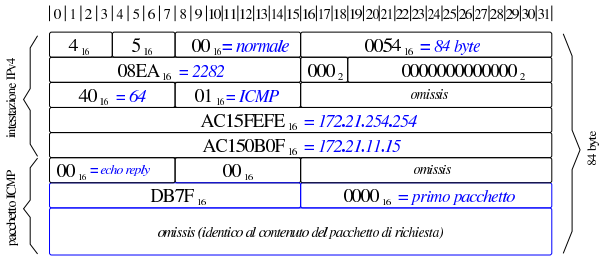





Figura 83.152. Esempio di pacchetto ICMP *echo reply* (pong), restituito da 172.21.254.254, completo di intestazione IPv4.




## 83.12.5 UDP

I pacchetti del protocollo UDP si inseriscono all'interno di pacchetti IP. I pacchetti UDP contengono a loro volta una propria intestazione, nella quale si prevede l'uso di un codice di controllo, relativo a tutto il pacchetto UDP, a cui però si aggiunge una pseudo-intestazione («pseudo», in quanto viene usata solo ai fini del calcolo del codice di controllo e non fa parte effettivamente del pacchetto). Il protocollo UDP inserisce il concetto di «porta» (*port*), distinto tra origine e destinazione.

Figura 83.153. Struttura effettiva di un pacchetto del protocollo UDP.

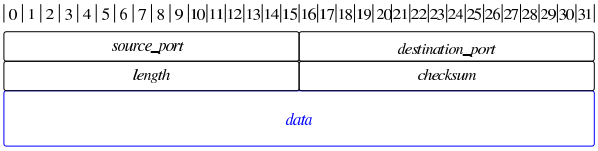



Figura 83.154. Pseudo-intestazione IPv4, utile per il calcolo del codice di controllo UDP, ma non facente parte del pacchetto.

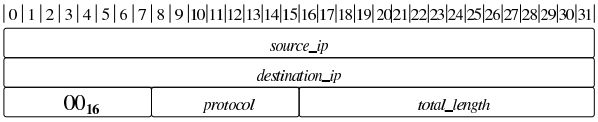



Tabella 83.155. Campi dell'intestazione UDP e della pseudo-intestazione relativa al protocollo IPv4.

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***source_port*** ***destination_port*** | 16 bit | Numero della porta di origine e di quella di destinazione del pacchetto. |
| ***length*** | 16 bit | La lunghezza in ottetti (byte) del pacchetto UDP, composto da intestazione UDP e contenuto associato. Il valore minimo di questo valore è otto, ovvero il contenuto della sola intestazione. |
| ***checksum*** | 16 bit | Codice di controllo, ma facoltativo se usato nel protocollo IPv4. Per non applicare il codice di controllo, occorre mettere qui il valore zero. Se si vuole calcolare, questo deve riguardare tutto il pacchetto UDP e la pseudo-intestazione, con l'accortezza di trasformare un eventuale risultato nullo nel valore $FFFF_{16}$. |
| ***source_ip*** ***destination_ip*** | 32 bit | Come nell'intestazione IPv4. |
| ***protocol*** | 16 bit | Il codice del protocollo, corrispondente in questo caso a $11_{16}$, ovvero UDP. |
| ***total_length*** | 16 bit | Lunghezza complessiva del pacchetto UDP, equivalente al campo ***length*** dell'intestazione UDP reale. |

Listato 83.156. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo UDP. Questa struttura vale indifferentemente per le architetture *little endian* e *big endian*.

```
struct udphdr
{
  uint16_t source;      // source port
  uint16_t dest;        // destination port
  uint16_t len;         // length
  uint16_t check;       // checksum
} __attribute__ ((packed));
```

A titolo di esempio si considerano due nodi, come nella figura successiva, e si analizza il contenuto di un pacchetto UDP, inviato da «A» a «B».

Figura 83.157. Dal nodo «A», porta 48281, parte un pacchetto UDP verso il nodo «B», alla porta 1234.


Figura 83.158. Esempio di pacchetto UDP, inviato dall'indirizzo 172.21.11.15, porta 48281, a 172.21.254.254 porta 1234, contenente la stringa '**ciao\n**'. Il pacchetto è completo dell'intestazione IPv4 e i codici di controllo sono visibili.

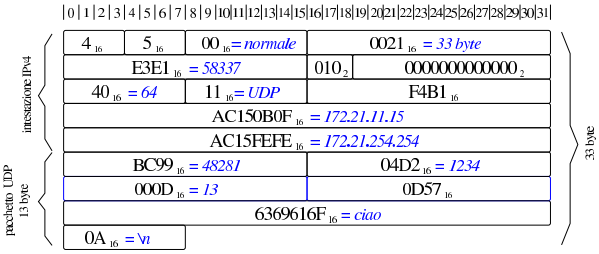
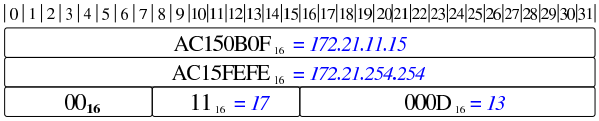



Figura 83.159. Pseudo-intestazione relativa al pacchetto di esempio della figura precedente.




Nel listato successivo si vede un piccolo programma che calcola il codice di controllo del pacchetto IPv4 dell'esempio.

Listato 83.160. Calcolo del codice di controllo nell'intestazione IPv4 dell'esempio. Una volta compilato, il programma visualizza correttamente il valore atteso: $F4B1_{16}$.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
  uint32_t sum = 0;
  uint16_t carry;
  uint16_t checksum;
  //
  sum += 0x4500;
  sum += 0x0021;
  sum += 0xe3e1;
  sum += 0x4000;
  sum += 0x4011;
  sum += 0x0000;
  sum += 0xac15;
  sum += 0x0b0f;
  sum += 0xac15;
  sum += 0xfefe;
  //
  carry     = sum >> 16;
  checksum  = sum & 0x0000FFFF;
  checksum += carry;
  //
  checksum = ~checksum;
  //
  printf ("0x%04x\n", checksum);
  //
  return 0;
}
```

Listato 83.161. Calcolo del codice di controllo nell'intestazione UDP dell'esempio che tiene conto della pseudo-intestazione. Una volta compilato, il programma visualizza correttamente il valore atteso: $0D57_{16}$.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
  uint32_t sum = 0;
  uint16_t carry;
  uint16_t checksum;
  //
  sum += 0xbc99;
  sum += 0x04d2;
  sum += 0x000d;
  sum += 0x0000;
  sum += 0x6369;
  sum += 0x616f;
  sum += 0x0a00;

  sum += 0xac15;
  sum += 0x0b0f;
  sum += 0xac15;
  sum += 0xfefe;
  sum += 0x0011;
  sum += 0x000d;
  //
  carry     = sum >> 16;
  checksum  = sum & 0x0000FFFF;
  checksum += carry;
  //
  checksum = ~checksum;
  //
  if (checksum == 0)
    {
      checksum = 0xffff;
    }
  //
  printf ("0x%04x\n", checksum);
  //
  return 0;
}
```

### 83.12.6 TCP

Il protocollo TCP si distingue da UDP in quanto permette di stabilire un flusso di dati bidirezionale tra due porte di due nodi. Il processo che inizia una connessione TCP, apre una porta presso il nodo locale in cui si trova a funzionare, contattando una porta di un altro nodo, presso la quale si deve trovare un altro processo in attesa. Successivamente i processi coinvolti non si preoccupano di altro, a parte il fatto di trasmettere e ricevere dati attraverso il canale costituito dalla connessione. Infatti, la gestione della connessione TCP avviene per opera del sistema operativo, attraverso l'invio e la ricezione dei pacchetti relativi, con tutti i controlli necessari a garantire la correttezza del flusso di dati.

L'intestazione di un pacchetto del protocollo TCP contiene degli indicatori (*flag*), alcuni dei quali sono essenziali e appaiono descritti nella tabella successiva:

| Indicatore | Denominazione | Descrizione |
|---|---|---|
| ACK | *acknowledge* | Si conferma la ricezione, specificando qual è il prossimo byte che si attende di ricevere dalla controparte. |
| PSH | *push* | Si richiede che i dati trasmessi fino a quel punto siano recapitati senza indugio al processo in ricezione dall'altra parte. |

| Indicatore | Denominazione | Descrizione |
|---|---|---|
| RST | *reset* | Azzeramento della connessione. |
| SYN | *synchronization* | Inizio di una connessione. |
| FIN | *finalization* | Conclusione della trasmissione. |

Il protocollo TCP, gestito dal sistema operativo, richiede che la ricezione dei pacchetti contenenti dati sia confermata dalla controparte. Ma non è strettamente necessario confermare ogni pacchetto ricevuto, in quanto il riferimento è al byte ***n***-esimo, che quindi convalida anche quelli ricevuti in precedenza. In tal modo, una delle due parti può tentare di trasmettere più pacchetti in rapida successione, prima di ricevere una conferma; poi, se la conferma arriva solo parzialmente, può ritrasmettere a partire dalla porzione non ancora confermata.

Una connessione TCP prevede undici stati, descritti dalla tabella successiva:

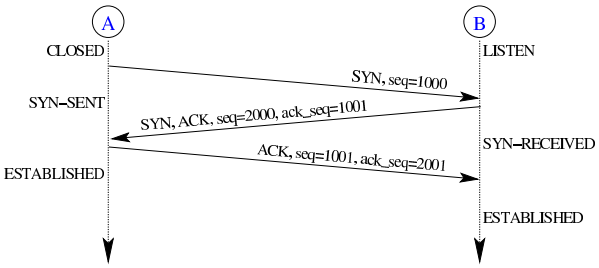
| Stato | Descrizione |
|---|---|
| LISTEN | Si è in attesa di una richiesta di connessione. Questa condizione si verifica quando un processo apre una porta TCP locale e attende di ricevere una richiesta da un altro processo (locale o remoto) per poter instaurare con lui una connessione. |
| SYN-SENT | È già stato inviato un pacchetto SYN e si è in attesa di un pacchetto SYN di risposta. Si verifica questa condizione quando un processo tenta di contattarne un altro, già in ascolto su una certa porta di un certo nodo. |
| SYN-RECEIVED | È stato ricevuto un pacchetto SYN e a questo è stato risposto con una conferma e un altro pacchetto SYN, del quale si attende conferma (ACK). Dopo la conferma attesa, si passa allo stato successivo: ESTABLISHED. |
| ESTABLISHED | La connessione è stata instaurata e il flusso dei dati, nelle due direzioni, può avere luogo. |
| FIN-WAIT-1 | L'applicazione locale (rispetto alla connessione) ha chiuso il proprio flusso di trasmissione ed è stato inviato un pacchetto FIN alla controparte, rimanendo in attesa di una conferma o di un altro pacchetto FIN. |
| CLOSE-WAIT | È stato ricevuto un pacchetto FIN dalla controparte ed è stata trasmessa la conferma relativa, mentre l'applicazione locale non ha ancora chiuso il proprio flusso in uscita (in scrittura). |
| FIN-WAIT-2 | Dopo lo stato FIN-WAIT-1 è stata ricevuta la conferma e si passa quindi all'attesa che la controparte completi l'invio di dati con un pacchetto FIN. |
| CLOSING | Dopo lo stato FIN-WAIT-1, prima di ricevere una conferma dall'altra parte, è stato ricevuto un pacchetto FIN ed è stata inviata conferma di questo: si attende quindi la conferma al proprio pacchetto FIN. |
| LAST-ACK | È stato ricevuto un pacchetto FIN, è stato inviata conferma e successivamente è stato inviato un pacchetto FIN: si rimane quindi in attesa di una conferma dall'altra parte. |
| TIME-WAIT | I due lati della connessione sono stati chiusi con i pacchetti FIN e le conferme rispettive sono state inviate: si attende comunque qualche tempo prima di eliminare la connessione dalla gestione del sistema. Il tempo previsto è di 2 MLS (praticamente 4 minuti). |
| CLOSED | Condizione immaginaria di una connessione ormai chiusa e dimenticata dal sistema di gestione del protocollo TCP. |

Nelle figure successive si esemplifica il procedere degli stati di una connessione, partendo dalla sua creazione, fino alla sua conclusione, ipotizzando un breve scambio di dati. Ognuno dei lati della connessione decide qual è il proprio numero iniziale di sequenza; da quel punto in poi, l'incremento di quel valore serve a consentire la verifica dell'ordine che devono avere i pacchetti. Il valore rappresentato dalla metavariabile ***seq*** è il numero di sequenza iniziale del pacchetto, mentre ***ack_seq*** è il valore di sequenza che ci si attende di ricevere. Per esempio, un valore di ***ack_seq*** pari a 1234 significa che sono stati ricevuti dati fino al byte corrispondente alla sequenza 1233 e, se altri dati devono giungere, il prossimo byte da ricevere deve essere quello con il numero di sequenza 1234. In ogni caso, va osservato



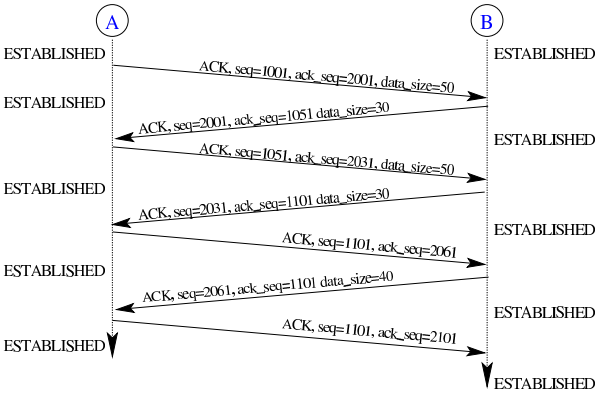
che nella creazione della connessione e nella sua conclusione, c'è un momento in cui il numero di sequenza viene incrementato di una unità, senza che ciò sia dovuto alla trasmissione effettiva di un byte.

Figura 83.164. Negoziazione iniziale: *three way handshake*. Dal lato «A» si inizia il procedimento per instaurare una connessione con il lato «B». La sequenza iniziale dal lato «A» è pari a 1000, mentre quella iniziale dal lato «B» è qui pari a 2000.




Dopo la negoziazione con i valori ipotizzati nell'esempio, il primo byte che «A» può trasmettere a «B» ha il numero di sequenza 1001, mentre nel senso opposto questo numero è 2001. Nella figura successiva, «A» e «B» si inviano dati reciprocamente per 100 byte ciascuno.

Figura 83.165. Quando i due lati della connessioni sono pronti, le due parti possono trasmettersi dei dati. In questo caso si ipotizza che ogni pacchetto sia sempre confermato.




Dopo il breve scambio di dati, il lato «A» decide di chiudere la propria trasmissione (scrittura), informando la controparte, la quale, tuttavia, rimane nella facoltà di continuare a inviare dati.

Figura 83.166. Il lato «A» chiude il suo canale di trasmissione.

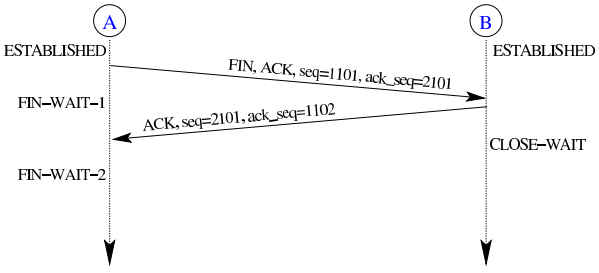


Figura 83.167. Dopo la chiusura dal lato «A», si ipotizza che il lato «B» continui a trasmettere in tutto altri 100 byte di dati.

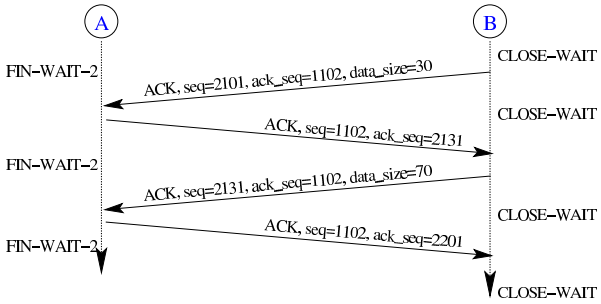
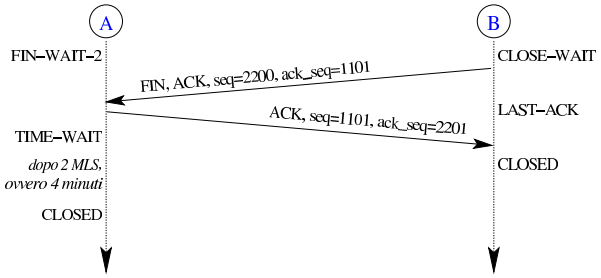
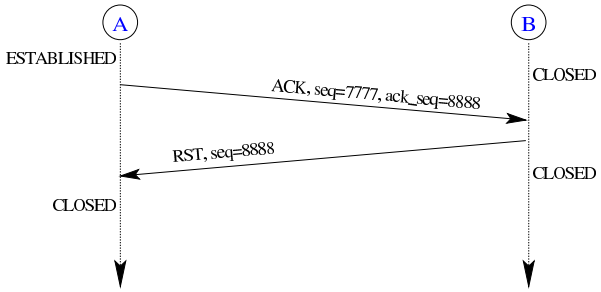



Figura 83.168. Dopo la chiusura dal lato «A», e dopo che il lato «B» ha finito con la propria trasmissione, anche questo decide di chiudere e si arriva alla conclusione della connessione. Tuttavia, il lato «A» che alla fine non può sapere se il lato «B» ha ricevuto effettivamente la conferma, rimane nello stato di TIME-WAIT per un certo ammontare di tempo, prima che per lui la connessione si possa considerare completamente chiusa.




Quando una delle parti viene confusa per qualche motivo, ricevendo un pacchetto che non sa qualificare nella connessione in corso o perché non fa proprio parte di una connessione, la risposta avviene attraverso un pacchetto con l'indicatore RST attivo.

Figura 83.169. Il ricevimento di un pacchetto fuori contesto, provoca una risposta di azzeramento.




Nella figura successiva si vede la struttura effettiva di un pacchetto TCP, da considerare all'interno di un pacchetto IPv4. Le opzioni sono facoltative e non sempre vengono trasportati dei dati.

Figura 83.170. Struttura effettiva di un pacchetto del protocollo TCP.

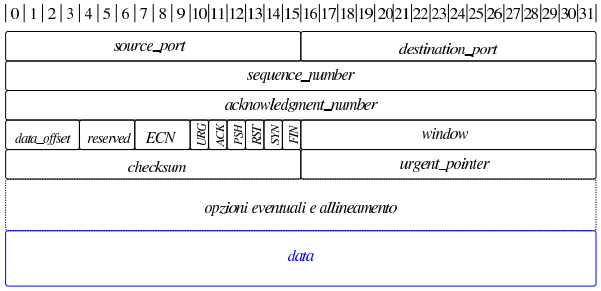





Figura 83.171. Pseudo-intestazione IPv4, utile per il calcolo del codice di controllo TCP, ma non facente parte del pacchetto.

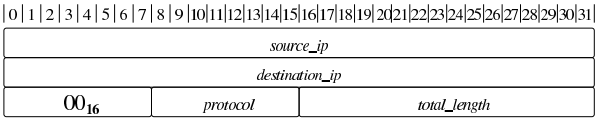



Tabella 83.172. Campi dell'intestazione TCP e della pseudo-intestazione relativa al protocollo IPv4.

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***source_port*** ***destination_port*** | 16 bit | Numero della porta di origine e di quella di destinazione del pacchetto. |
| ***sequence_number*** | 32 bit | Numero di sequenza del primo byte di dati allegati, ammesso che ci siano effettivamente dati contenuti nel pacchetto. Va osservato che quando viene inviato un pacchetto SYN, questo numero rappresenta la sequenza iniziale della connessione, mentre il primo byte di dati che può poi essere trasmesso deve avere un numero di sequenza pari a questo valore più una unità. |
| ***acknowledgment_number*** | 32 bit | Numero di sequenza del primo byte di dati che si attende di ricevere dalla controparte, in quanto le sequenze precedenti sono già state ricevute correttamente. |
| ***data_offset*** | 4 bit | La quantità di blocchi a 32 bit di cui si compone l'intestazione, opzioni incluse; in pratica, moltiplicando questo valore per quattro, si ottiene il puntatore al primo byte di dati contenuto nel pacchetto. |
| ***reserved*** | 3 bit | Un campo da lasciare a zero. |
| ***ECN*** | 3 bit | *Explicit congestion notification*<br>Il contenuto di questo campo è definito dal RFC 3160. In condizioni normali, questo campo viene lasciato a zero. |
| ***URG*** | 1 bit | *urgent*<br>Se l'indicatore è attivo (ha il valore 1), significa che il valore contenuto nel campo ***urgent pointer*** è significativo. In condizioni normali, questo indicatore è a zero e il campo ***urgent_pointer*** è a zero. |
| ***ACK*** | 1 bit | *acknowledge*<br>Se l'indicatore è attivo significa che il contenuto del campo ***acknowledgement_sequence*** indica il numero di sequenza del primo byte di dati attesi dalla controparte; diversamente, significa che quel numero di sequenza non è da considerare. |
| ***PSH*** | 1 bit | *push*<br>Se l'indicatore è attivo significa che si sta chiedendo di recapitare senza indugio i dati trasmessi fino a questo punto, all'applicazione di destinazione. |

| Campo | Lunghezza | Descrizione |
|---|---|---|
| ***RST*** | 1 bit | *reset*<br>Se l'indicatore è attivo significa che è stato ricevuto un pacchetto TCP dalla controparte con un numero ***acknowledgement_sequence*** pari al numero di sequenza del pacchetto attuale, ma ciò risulta al di fuori di una connessione conosciuta e si richiede pertanto la cancellazione di tale comunicazione. |
| ***SYN*** | 1 bit | *synchronization*<br>Se l'indicatore è attivo significa che si sta tentando di instaurare una connessione. |
| ***FIN*** | 1 bit | *finalization*<br>Se l'indicatore è attivo significa che si sta chiudendo il canale di trasmissione verso la controparte. |
| ***Window*** | 16 bit | Si tratta della «finestra di ricezione», pari alla quantità di byte che possono essere ricevuti simultaneamente, in uno o più pacchetti successivi. |
| ***checksum*** | 16 bit | Codice di controllo, relativo a tutto il pacchetto TCP e alla pseudo-intestazione (come per UDP). |
| ***urgent_pointer*** | 16 bit | Puntatore a dati «urgenti», valido solo se l'indicatore ***URG*** risulta attivo. |
| ***source_ip***<br>***destination_ip*** | 32 bit | Come nell'intestazione IPv4. |
| ***protocol*** | 16 bit | Il codice del protocollo, corrispondente in questo caso a $06_{16}$, ovvero TCP. |
| ***total_length*** | 16 bit | Lunghezza complessiva del pacchetto TCP. |

Listato 83.173. Esempio di struttura in linguaggio C, per descrivere l'intestazione di un pacchetto del protocollo TCP. A sinistra nella versione *little endian*; a destra in quella *big endian*. I campi ***res1*** e ***res2*** includono i tre bit riservati successivi a ***data_offset*** e il campo ***ECN***.

*little endian*

```
struct tcphdr
{
  uint16_t source;
  uint16_t dest;
  uint32_t seq;
  uint32_t ack_seq;
  uint16_t res1 : 4,
           doff : 4,
           fin  : 1,
           syn  : 1,
           rst  : 1,
           psh  : 1,
           ack  : 1,
           urg  : 1,
           res2 : 2;
  uint16_t window;
  uint16_t check;
  uint16_t urg_ptr;
};
```

*big endian*

```
struct tcphdr
{
  uint16_t source;
  uint16_t dest;
  uint32_t seq;
  uint32_t ack_seq;
  uint16_t doff : 4,
           res1 : 4,
           res2 : 2,
           urg  : 1,
           ack  : 1,
           psh  : 1,
           rst  : 1,
           syn  : 1,
           fin  : 1;
  uint16_t window;
  uint16_t check;
  uint16_t urg_ptr;
};
```

A titolo di esempio si considerano due nodi, come nella figura successiva, e si analizza il contenuto di un pacchetto TCP, inviato da «A» a «B», nell'ambito di una connessione in corso (entrambi i lati sono nella condizione CONNECTED).



Figura 83.174. Dal nodo «A», porta 11111, parte un pacchetto TCP verso il nodo «B», alla porta 12345.


Figura 83.175. Esempio di pacchetto TCP, inviato dall'indirizzo 172.21.11.15, porta 11111, a 172.21.254.254 porta 12345, contenente la stringa '**`ciao\n`**'. Il pacchetto è completo dell'intestazione IPv4 e i codici di controllo sono visibili.

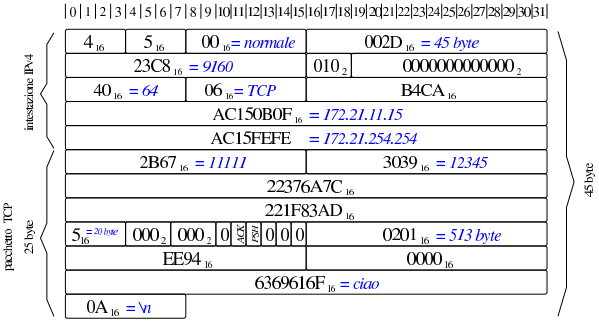
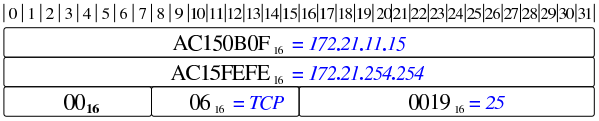



Figura 83.176. Pseudo-intestazione relativa al pacchetto di esempio della figura precedente.




Nel listato successivo si vede un piccolo programma che calcola il codice di controllo del pacchetto IPv4 dell'esempio.

Listato 83.177. Calcolo del codice di controllo nell'intestazione IPv4 dell'esempio. Una volta compilato, il programma visualizza correttamente il valore atteso: $B4CA_{16}$.

```
#include <stdint.h>
#include <stdio.h>
int
main (void)
{
  uint32_t sum = 0;
  uint16_t carry;
  uint16_t checksum;
  //
  sum += 0x4500;
  sum += 0x002D;
  sum += 0x23C8;
  sum += 0x4000;
  sum += 0x4006;
  sum += 0x0000;
  sum += 0xAC15;
  sum += 0x0B0F;
  sum += 0xAC15;
  sum += 0xFEFE;
  //
  carry     = sum >> 16;
  checksum  = sum & 0x0000FFFF;
  checksum += carry;
  //
  checksum = ~checksum;
  //
  printf ("0x%04x\n", checksum);
  //
  return 0;
}
```

Listato 83.178. Calcolo del codice di controllo nell'intestazione TCP dell'esempio che tiene conto della pseudo-intestazione. Una volta compilato, il programma visualizza il valore: $EE94_{16}$.

```
#include <stdint.h>
#include <stdio.h>
```

```
int
main (void)
{
  uint32_t sum = 0;
  uint16_t carry;
  uint16_t checksum;
  //
  sum += 0x2B67;
  sum += 0x3039;
  sum += 0x2237;
  sum += 0x6A7C;
  sum += 0x221F;
  sum += 0x83AD;
  sum += 0x5018;
  sum += 0x0201;
  sum += 0x0000;
  sum += 0x0000;
  sum += 0x6369;
  sum += 0x616F;
  sum += 0x0A00;

  sum += 0xac15;
  sum += 0x0b0f;
  sum += 0xac15;
  sum += 0xfefe;
  sum += 0x0006;
  sum += 0x0019;
  //
  carry     = sum >> 16;
  checksum  = sum & 0x0000FFFF;
  checksum += carry;
  //
  checksum = ~checksum;
  //
  printf ("0x%04x\n", checksum);
  //
  return 0;
}
```

## 83.13 Riferimenti


- *Intel® 64 and IA-32 Architectures Software Developer's Manuals*, http://developer.intel.com/products/processor/manuals/index.htm
- David Brackeen, *256-Color VGA Programming in C*, http://www.brackeen.com/vga/
- Brandon Friesen, *Bran's kernel development tutorial*, http://www.osdever.net/bkerndev/Docs/title.htm
- Wikipedia, *Global Descriptor Table*, http://en.wikipedia.org/wiki/Global_Descriptor_Table
- *Higher Half With GDT*, http://wiki.osdev.org/Higher_Half_With_GDT
- Weqaar A. Janjua, *IA-32 Boot sector code*, http://sites.google.com/site/weqaar/Home/files
- Gergor Brunmar, *The world of Protected mode*, http://www.osdever.net/tutorials/pdf/gb_pmode.pdf
- John Fine, *Descriptor Tables: GDT, IDT and LDT*, http://www.osdever.net/tutorials/pdf/descriptors.pdf , http://www.osdever.net/tutorials/view/descriptor-tables-gdt-idt-ldt
- Jochen Liedtke, *Segments, Intel's IA-32 from a system architecture view*, http://wayback.archive.org/web/2004/http://i30ww30w.ira.uka.de/teaching/coursedocuments/48/segments.pdf
- Allan Cruse, *CS 630: Advanced Microcomputer Programming*, http://www.cs.usfca.edu/~cruse/cs630f06/
- Wikipedia, *Interrupt descriptor table*, http://en.wikipedia.org/wiki/Interrupt_descriptor_table
- *Interrupt Descriptor Table*, http://wiki.osdev.org/Interrupt_Descriptor_Table
- Alexander Blessing, *Programming the PIC*, http://www.osdever.net/tutorials/pdf/pic.pdf





- *8259 Programmable Interrupt Controller (PIC)*, http://www.pklab.net/index.php?id=94
- *Write your own Operating System*: *Interrupt Service Routines*, http://wiki.osdev.org/Interrupt_Service_Routines
- Wikipedia, *Intel 8253*, http://en.wikipedia.org/wiki/Intel_8253
- *Write your own Operating System*: *Programmable Interval Timer*, http://wiki.osdev.org/Programmable_Interval_Timer
- Mark Feldman, *Programming the Intel 8253 Programmable Interval Timer*, http://www.nondot.org/sabre/os/files/MiscHW/PIT.txt
- Salvatore D'Angelo, *Keyboard Driver*, http://opencommunity.altervista.org/samples/openjournal/keyboard.html
- Adam Chapweske, *The PS/2 Keyboard Interface*, http://www.burtonsys.com/ps2_chapweske.htm , http://www.tayloredge.com/reference/Interface/atkeyboard.pdf
- OSDev Wiki, *ATA PIO mode*, http://wiki.osdev.org/ATA_PIO_Mode
- T13, *AT Attachment with Packet Interface - 6 (ATA/ATAPI-6)*, http://bos.asmhackers.net/docs/ata/docs/ata-atapi-6-3b.pdf
- LDP, *PCI*, http://tldp.org/LDP/tlk/dd/pci.html
- OSDev Wiki, *PCI*, http://wiki.osdev.org/PCI
- PLX technology, *pci 9054*, http://www.nikhef.nl/~peterj/datasheets/9054db54-1C.pdf
- *PCI and AGP Vendors, Devices and Subsystems identification file*, http://wayback.archive.org/web/2009*/http://members.datafast.net.au/~dft0802/ , http://wayback.archive.org/web/2009*/http://members.datafast.net.au/~dft0802/downloads.htm
- OSDev Wiki, *NE2000*, http://wiki.osdev.org/Ne2000
- National semiconductor, *DP8390D/NS32490D NIC network interface controller*, http://www.national.com/pf/DP/DP8390D.html
- National semiconductor, *Writing drivers for DP8390 NIC family of ethernet controllers*, http://www.datasheetarchive.com/Indexer/Datasheet-017/DSA00296168.html
- Realtek, *RTL8019AS, Realtek Full-Duplex Ethernet Controller with Plug and Play Function (RealPNP), SPECIFICATION*, http://www.ethernut.de/pdf/8019as19ds.pdf
- Network Working Group, *RFC 826: An Ethernet Address Resolution Protocol*, 1982, http://www.ietf.org/rfc/rfc826.txt
- Information Sciences Institute, University of Southern California, *RFC 791: Internet protocol*, 1981, http://www.ietf.org/rfc/rfc791.txt
- J. Postel, *RFC 792: Internet control message protocol*, 1981, http://www.ietf.org/rfc/rfc792.txt
  J. Mogul, J. Postel, *RFC 950: Internet standard subnetting procedure*, 1985, http://www.ietf.org/rfc/rfc950.txt
- J. Postel, *RFC 768: User datagram protocol*, 1980, http://www.ietf.org/rfc/rfc768.txt
- Information science institute, *RFC 793: Transmission control protocol*, 1981, http://www.ietf.org/rfc/rfc793.txt


[1] La modalità protetta è quella che consente di accedere alla memoria oltre il limite di 1 Mibyte.

[2] Alla tabella GDT possono essere collegate delle tabelle LDT, ovvero *local description table*, con il compito di individuare delle porzioni di memoria per conto di processi elaborativi singoli.

[3] Per accesso lineare alla memoria si intende che l'indirizzo relativo del segmento corrisponde anche all'indirizzo reale della memoria stessa. In inglese si usa il termine *flat memory*.

[4] Per rappresentare i numeri da 0 a 8191 servono precisamente 13 bit. Nei selettori di segmento si usano i 13 bit più significativi per individuare un descrittore.