Parte vii

Progetto

Capitolo 83

Primi passi verso un sistema per hardware x86-32

«

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

`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'
```

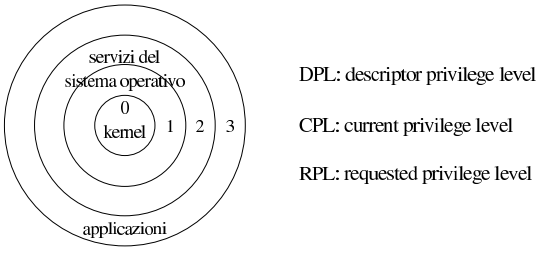
```
#!/bin/sh
qemu -fda floppy.img -boot a
```

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.



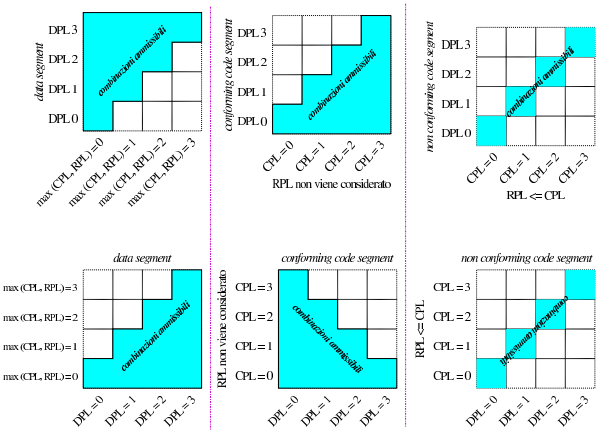
I microprocessori x86-32 definiscono precisamente quattro anelli, numerati da zero a tre e vi si attribuiscono convenzionalmente delle competenze: al livello zero, corrispondente all'anello centrale, competono i privilegi più importanti, ovvero quelli del kernel; al livello tre, corrispondente all'anello 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.

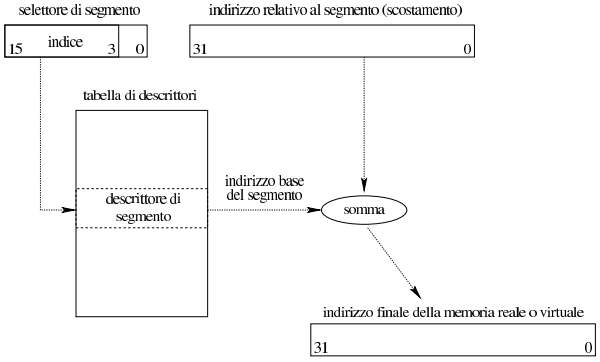
Quando il processo elaborativo raggiunge del codice «conforme» a partire da un livello di privilegio attuale meno importante di quello del codice in questione, il valore di CPL non cambia, quindi non cambia nemmeno la pila dei dati relativa al processo elaborativo.

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.

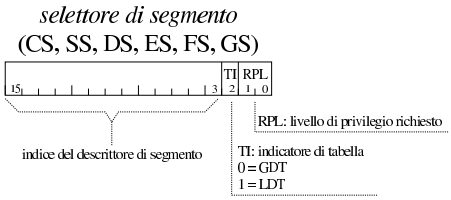
Per individuare un indirizzo di memoria (reale o virtuale), si parte da un ***selettore***, contenuto in un registro di segmento appropriato al contesto, dal quale si ottiene un indice per selezionare una voce da una tabella di descrittori. Attraverso l'indice si individua il descrittore di un segmento, 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);
```

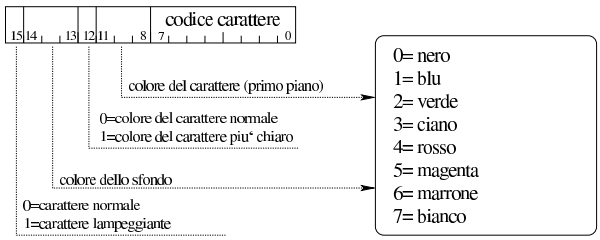

83.3 Utilizzo dello schermo VGA a caratteri

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.

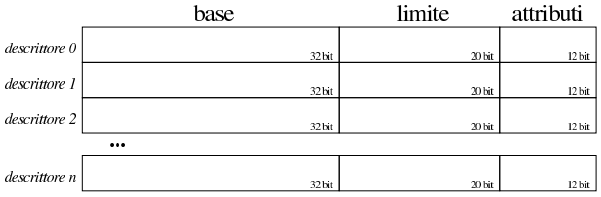
Negli esempi che appaiono nelle sezioni successive, si fa riferimento alla predisposizione di una tabella GDT, a partire da un sistema che è già in modalità protetta, essendo consentito di accedere a tutta la memoria, linearmente, senza alcuna limitazione.

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

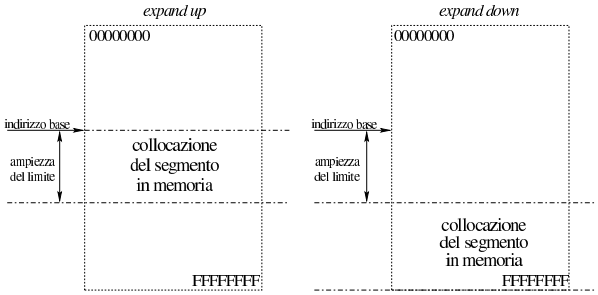
Inizialmente conviene considerare la tabella GDT in modo semplificato, organizzata a righe e colonne, come si vede nello schema successivo, dal momento che nella realtà i dati sono spezzettati e sparpagliati nello spazio a disposizione. Le righe di questa tabella possono essere al massimo 8192 e ogni 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 $FFFFF000_{16}$.

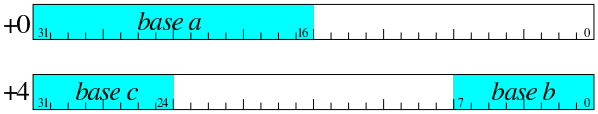
Figura 83.15. Confronto tra un limite da interpretare in modalità normale (a sinistra), rispetto a un limite da interpretare secondo un attributo di espansione in basso.



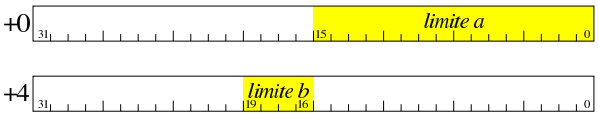
La direzione di espansione serve a determinare come si colloca l'area del segmento; si distinguono due casi: da ***base***, fino a ***base***+(***limite***×***granularità***) incluso; oppure da ***base***+(***limite***×***granularità***)+1 a $FFFFFFFF_{16}$. Il concetto è illustrato dalla figura già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

83.4.3 Struttura effettiva della tabella GDT

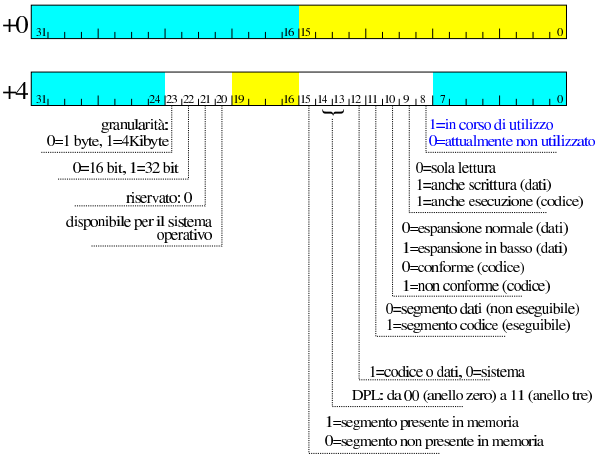
Nella realtà, la tabella GDT è formata da un array di descrittori, ognuno dei quali è composto da 8 byte, rappresentati qui in due blocchi da 32 bit, come nello schema successivo, dove viene evidenziata la porzione che riguarda l'indicazione dell'indirizzo iniziale del segmento di memoria:



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**:

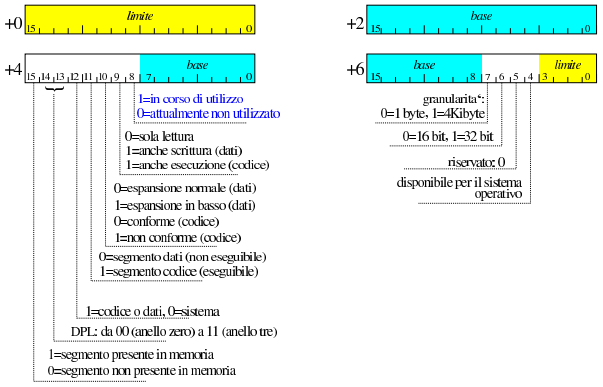


A proposito del bit che rappresenta il tipo di espansione o la conformità, in generale va usato con il valore zero, a indicare che il limite rappresenta l'espansione del segmento, a partire dalla sua ori-

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.

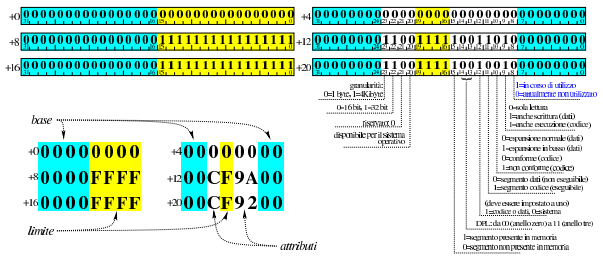
Quando si deve predisporre una tabella GDT prima di essere passati al funzionamento in modalità protetta, ovvero quando non ci si può avvalere di un sistema di avvio che offre una modalità protetta provvisoria, occorre ragionare a blocchi da 16 bit, non essendoci la possibilità di usare istruzioni a 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);
}
```

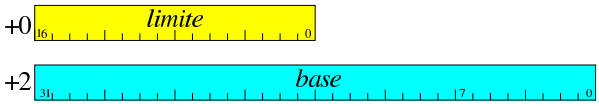
Per usare questa funzione occorre prima dichiarare l'array di descrittori di segmento. L'esempio seguente serve a riprodurre la tabella elementare della figura 83.20:

```
...
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:

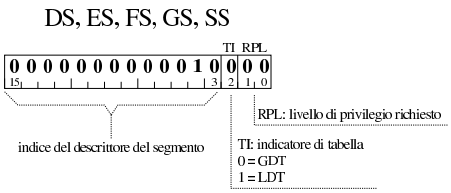
```
…
    lgdt (%eax)  # EAX contiene l'indirizzo della struttura.
…
```

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.

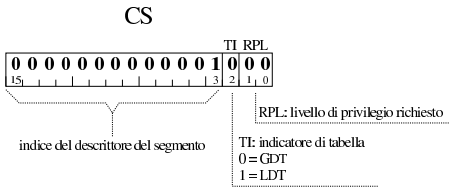


Come si può vedere nel disegno, il valore 10_{16} assegnato ai registri destinati ai segmenti di dati, contiene l'indice 2_{10} per la tabella GDT, con la richiesta di privilegi pari a zero (ovvero il valore più importante). Il descrittore con indice due della tabella GDT è esattamente quello che è stato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

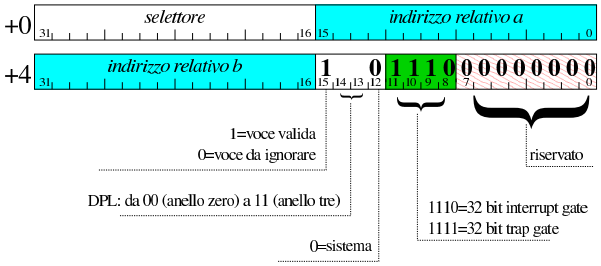
83.5 IDT

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

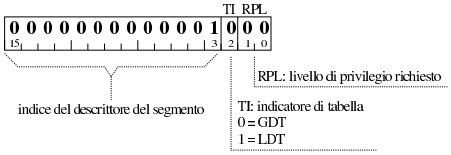
La tabella IDT è costituita da un array di descrittori di interruzione, ognuno dei quali occupa 64 bit. I descrittori possono essere al massimo 256 (da 0 a 255). Nel disegno successivo, viene mostrata la struttura di un descrittore della tabella IDT, prevedendo un accesso a blocchi da 32 bit:



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);
}
```

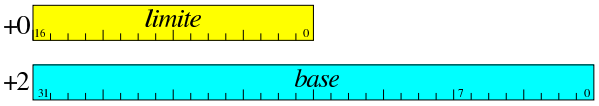

Per poter usare questa funzione occorre dichiarare prima l'array che rappresenta la tabella IDT. Di norma viene creata con tutti 256 descrittori possibili, assicurandosi che inizialmente siano azzerati effettivamente, anche se sarebbe sufficiente azzerare il bit di validità (il bit 15 del secondo blocco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

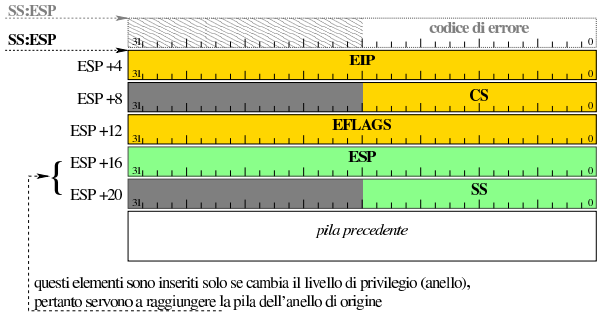
```
idt_load (void *idtr);
```

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**'.

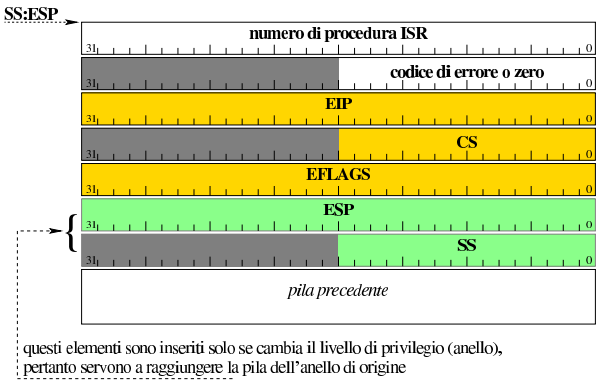
83.5.5 Bozza di un gestore di interruzioni

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.

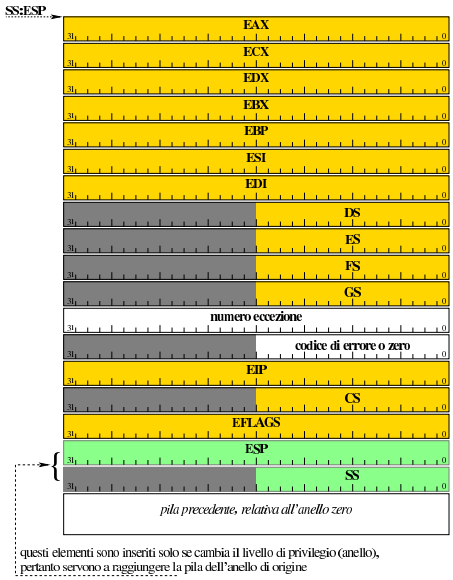
```
.extern interrupt_handler
#
.globl isr_0
.globl isr_1
…
.globl isr_254
.globl isr_255
#
isr_0:          # division by zero exception
    cli
    push $0     # Codice di errore fittizio.
    push $0     # Numero di procedura ISR.
    jmp isr_common
#
isr_1:          # debug exception
    cli
    push $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.

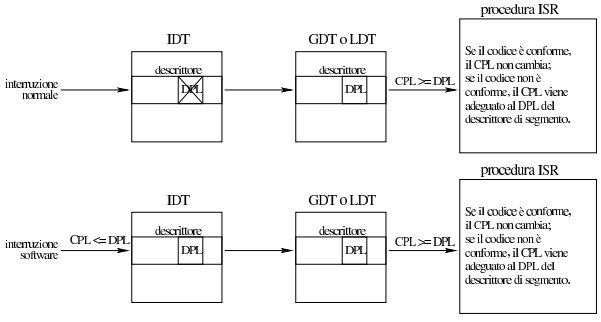
La prima cosa da considerare è il livello di privilegio del descrittore (DPL) del segmento codice in cui si trova la procedura ISR, il quale deve essere numericamente inferiore o uguale al livello corrente (CPL) precedente all'interruzione. Di conseguenza, è normale attendersi che le interruzioni comuni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

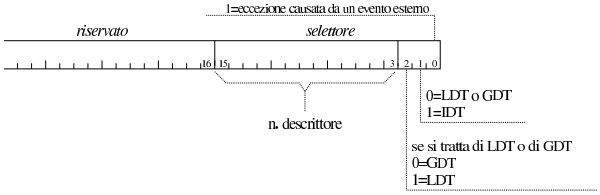
Va ricordato che in presenza di alcuni tipi di eccezione, il microprocessore accumula nella pila un codice di errore, pertanto, per uniformare le procedure ISR (*interrupt service routine*), occorre tenere conto dei casi in cui tale informazione è già inserita nella pila, rispetto a quelli dove questa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

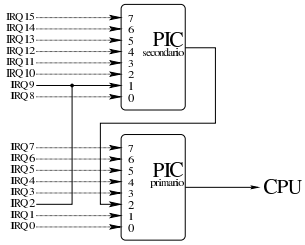
Quando il codice di errore è completamente a zero, almeno nei primi 16 bit meno significativi, vuol dire che non riguarda un problema

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*.

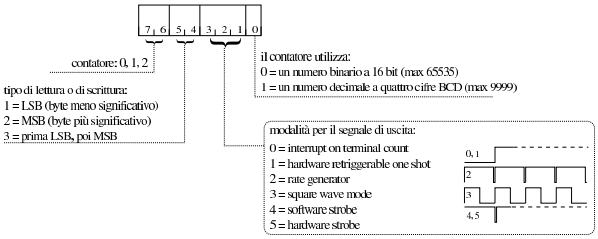
Questo integrato, o comunque ciò che ne fa la funzione, conta degli impulsi provenienti a una frequenza stabilita e, a seconda di come viene programmato, produce un risultato differente nelle sue tre uscite. Per esempio può generare un'onda quadra a una frazione della frequenza ricevuta in ingresso,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

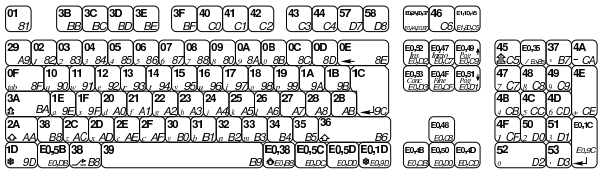
Il funzionamento della tastiera può essere configurato, inviando, a porte differenti, dei comandi che qui non vengono trattati; tuttavia la documentazione annotata nella bibliografia riporta tali informazioni.

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*.

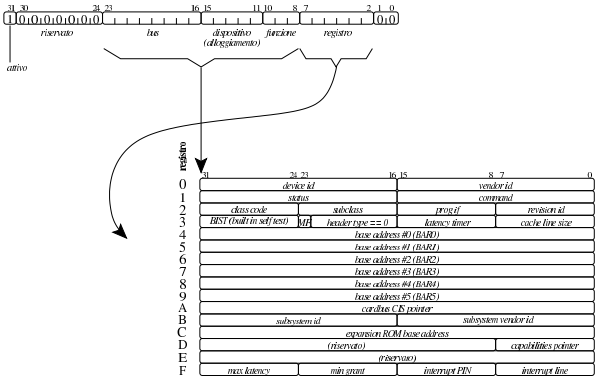
Le informazioni che il bus PCI ha da offrire sono strutturate in «tabelle» (ma nella documentazione standard si parla di «intestazioni»), suddivise in registri da 32 bit, suddivisi a loro volta in campi non omogenei tra loro. L'accesso a queste tabelle avviene registro per registro, attraverso la porta $0CFC_{16}$ (*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 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 23..16 15..8	*class code* *subclass* *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* *BAR1* *BAR2* *BAR3* *BAR4* *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.

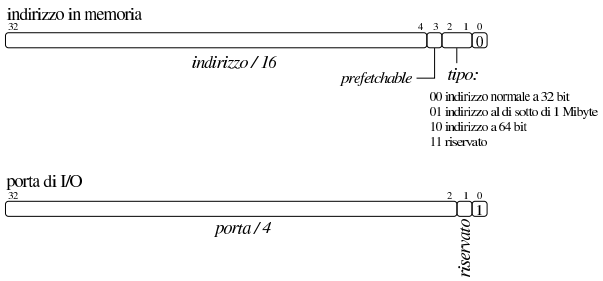


Tabella 83.106. Interpretazione della classe del dispositivo (*class code*).

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*.

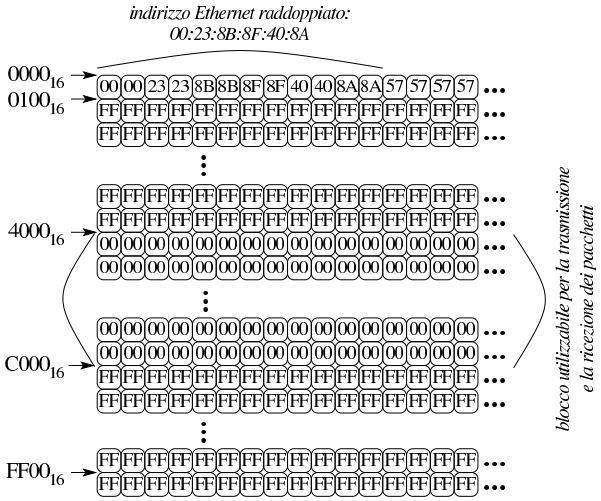
83.11.1 Memoria interna

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.

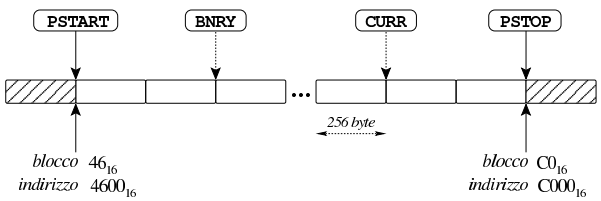


Per un'interfaccia di rete NE2000, il contenuto di un pacchetto, esclusa l'intestazione Ethernet, può andare da un minimo di 46 byte a un massimo di 1500 byte. Data l'organizzazione della memoria interna dell'interfaccia, un pacchetto utilizza da uno a sei blocchi da 256 byte (se un pacchetto è molto 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 $C0_{16}$.

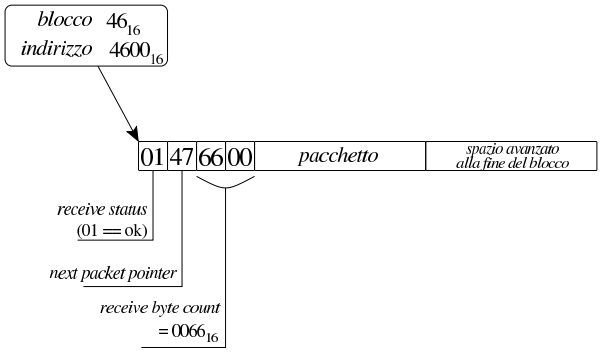
Mano a mano che la scrittura nella coda procede, viene incrementato un indice rappresentato dal registro ***CURR*** (*current page*), il quale rappresenta il prossimo blocco di memoria da utilizzare per la scrittura. Nell'esempio, il registro ***CURR*** contiene il valore BE_{16}, corrispondente al penultimo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



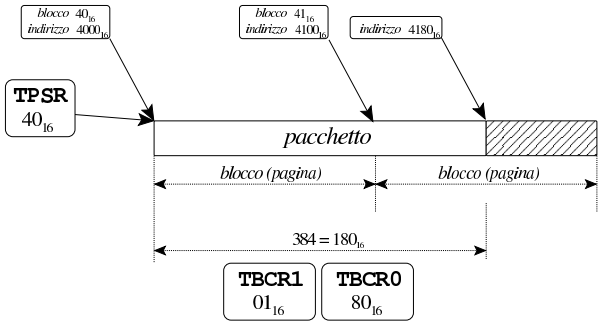
La figura ipotizza che nel blocco di memoria 46_{16}, pari all'indirizzo 4600_{16}, sia stato annotato un pacchetto ricevuto, i cui primi quattro byte indicano una ricezione corretta e una dimensione di 0066_{16} byte. Se successivamente a questo pacchetto ne è stato ricevuto un altro, questo lo si trova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

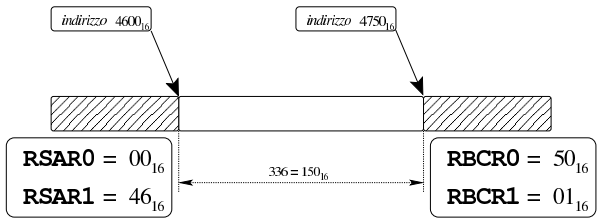


Si osservi che il pacchetto da trasmettere contiene solo ciò che serve per una trama Ethernet; pertanto, quei quattro byte di intestazione che si trovano in ricezione, non ci sono affatto in trasmissione.

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*.

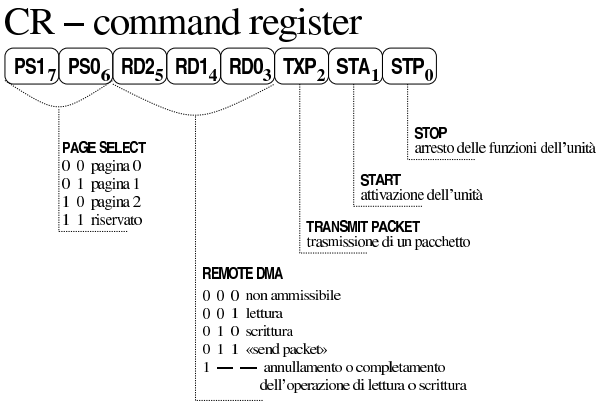


Figura 83.119. Sintesi dell'utilizzo del registro ***ISR***, *interrupt status register*.

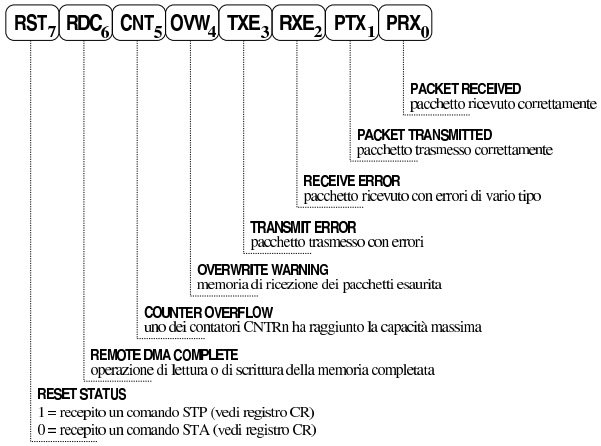


Figura 83.120. Sintesi dell'utilizzo del registro ***DCR***, *data configuration register*.

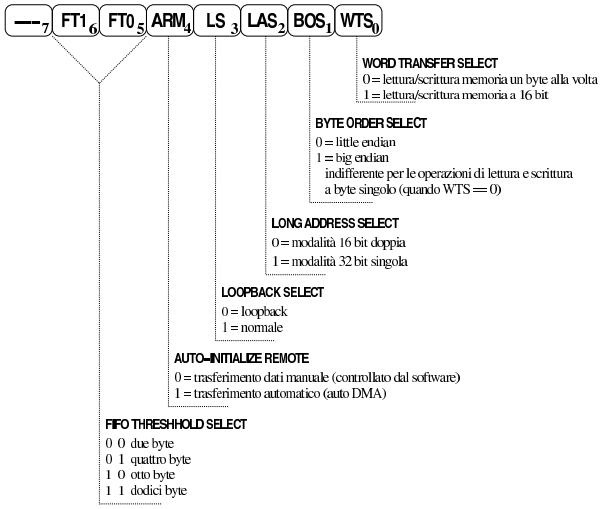


Figura 83.121. Sintesi dell'utilizzo del registro ***TCR***, *transmit configuration register*.

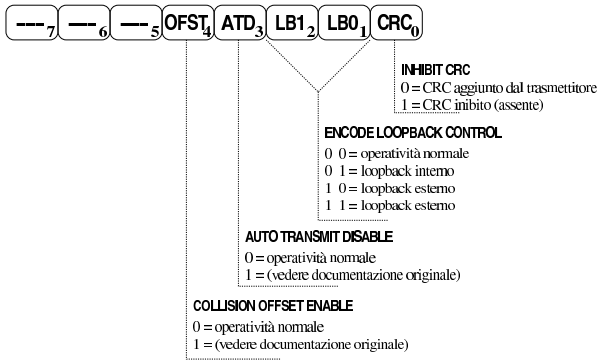


Figura 83.122. Sintesi dell'utilizzo del registro ***TSR***, *transmit status register*.

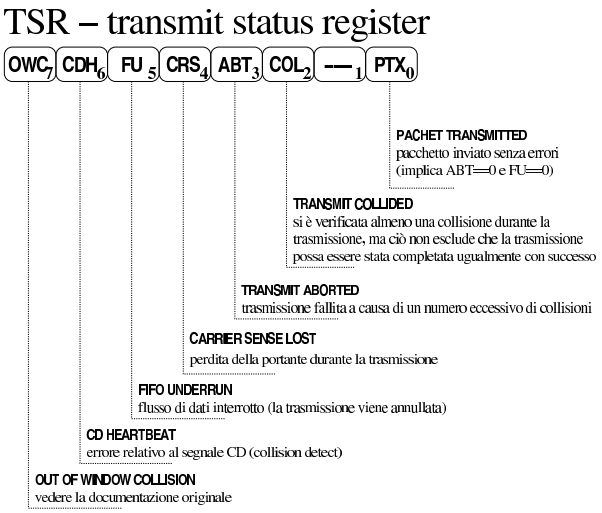


Figura 83.123. Sintesi dell'utilizzo del registro ***RCR***, *receive configuration register*.

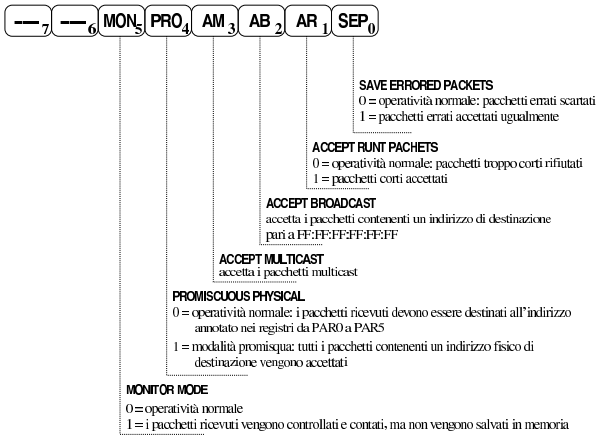
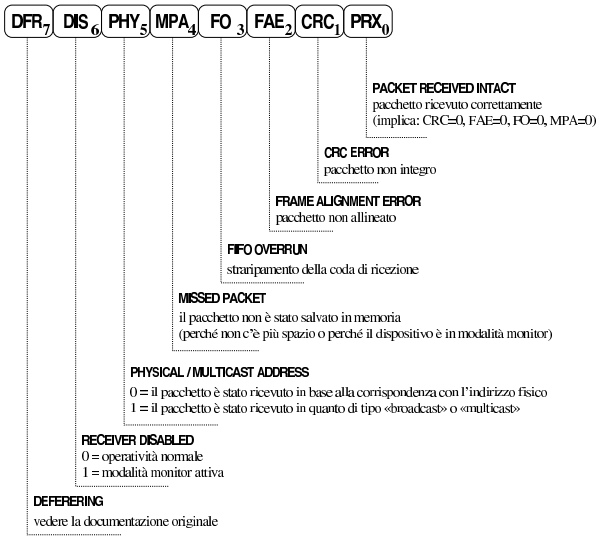


Figura 83.124. Sintesi dell'utilizzo del registro ***RSR***, *receive status register*.



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;
      }
  }
```

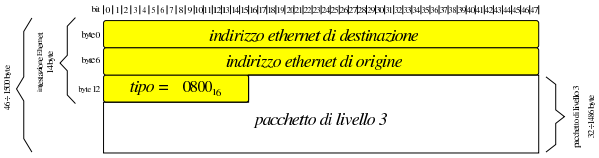
83.12 IPv4 in una rete Ethernet

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.

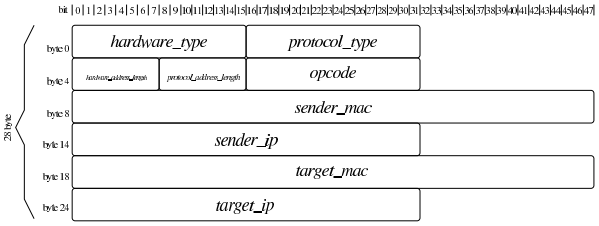


Tabella 83.131. Campi di un pacchetto del protocollo ARP, relativo a una rete fisica Ethernet e al protocollo IPv4.

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.

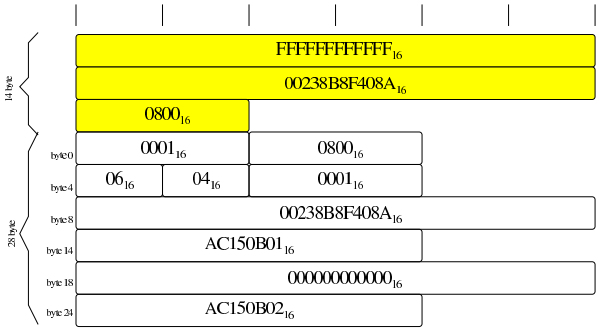
Va però osservato che il protocollo ARP serve a collegare il protocollo fisico con quello di rete; pertanto, per l'analisi dei pacchetti ARP occorre considerare anche il loro involucro a livello fisico.

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.

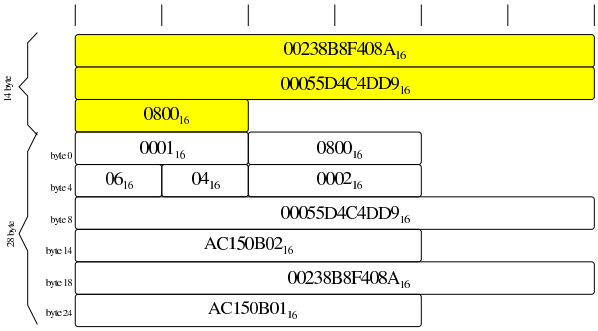
Il nodo «A» invia un pacchetto di richiesta nella rete fisica locale. Questo pacchetto deve essere diretto a tutti i nodi fisici raggiungibili, pertanto è contenuto in un pacchetto Ethernet «circolare», ovvero *broadcast*.

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.

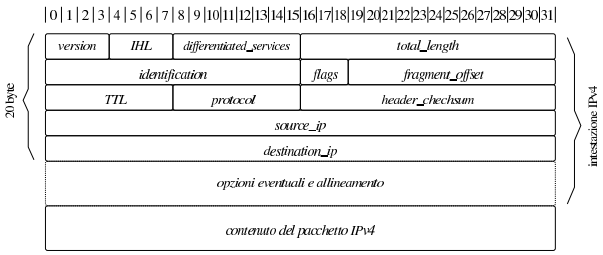


Tabella 83.137. Campi dell'intestazione di un pacchetto del protocollo IPv4. Le opzioni, se presenti, devono occupare uno spazio multiplo di 32 bit, riempiendo eventualmente i bit mancanti con valori a zero.

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: 01_{16} = ICMP; 06_{16} = TCP; 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;
};
```

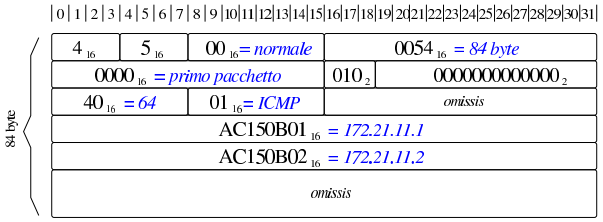
big endian

```
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).

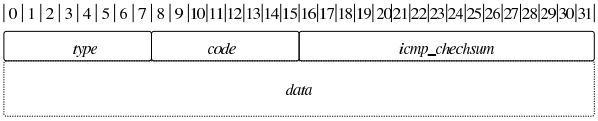


Tabella 83.146. Campi dell'intestazione comune di un pacchetto del protocollo ICMP.

Campo	Lunghezza	Descrizione
type	8 bit	Numero del tipo di messaggio: 01_{16} *echo reply* 03_{16} *destination unreachable* 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*.

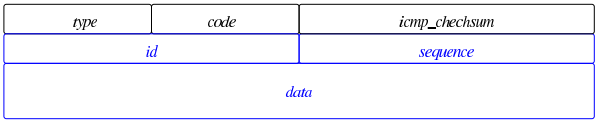


Tabella 83.148. Campi specifici dei pacchetti ICMP di eco.

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.

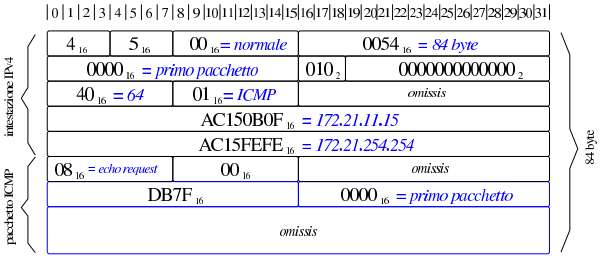
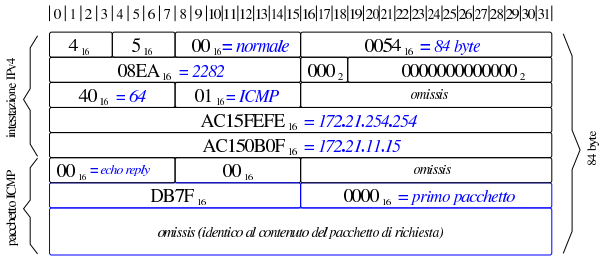


Figura 83.152. Esempio di pacchetto ICMP *echo reply* (pong), restituito da 172.21.254.254, completo di intestazione IPv4.



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.

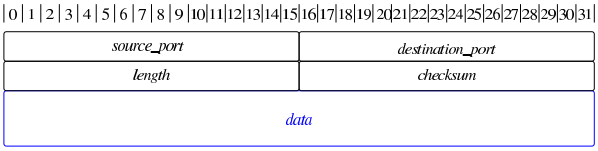


Figura 83.154. Pseudo-intestazione IPv4, utile per il calcolo del codice di controllo UDP, ma non facente parte del pacchetto.

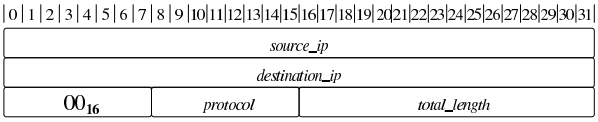


Tabella 83.155. Campi dell'intestazione UDP e della pseudo-intestazione relativa al protocollo IPv4.

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.

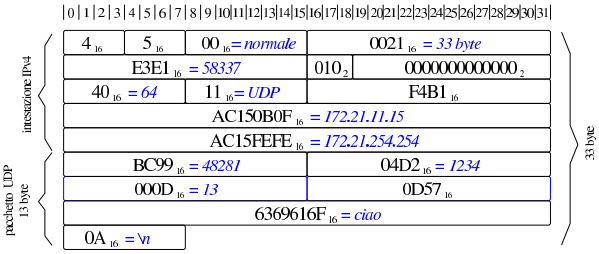
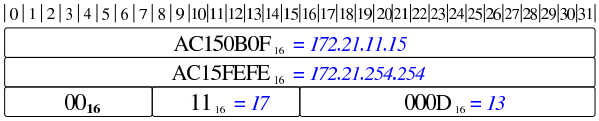


Figura 83.159. Pseudo-intestazione relativa al pacchetto di esempio della figura precedente.



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.

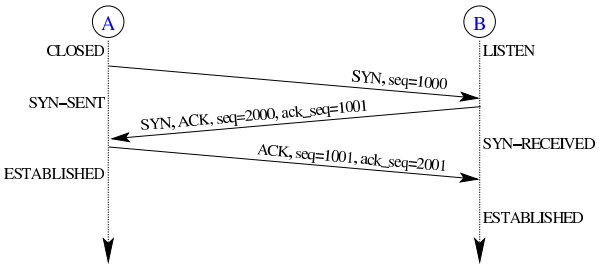
Una connessione TCP prevede undici stati, descritti dalla tabella successiva:

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

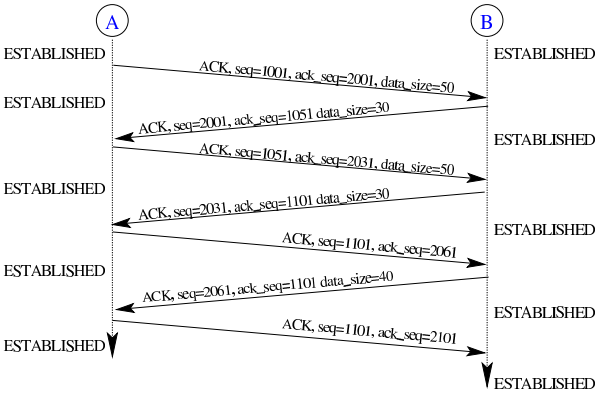
che nella creazione della connessione e nella sua conclusione, c'è un momento in cui il numero di sequenza viene incrementato di una unità, senza che ciò sia dovuto alla trasmissione effettiva di un byte.

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.

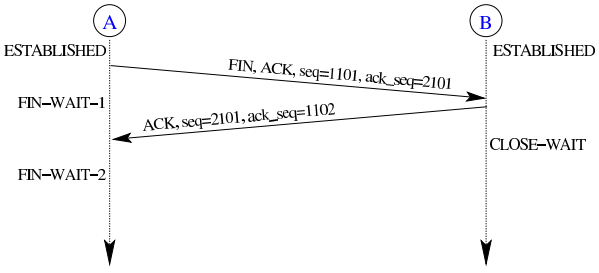


Figura 83.167. Dopo la chiusura dal lato «A», si ipotizza che il lato «B» continui a trasmettere in tutto altri 100 byte di dati.

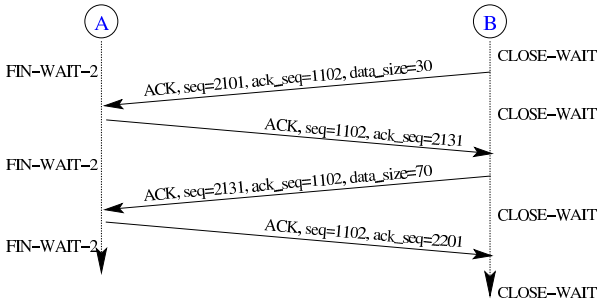
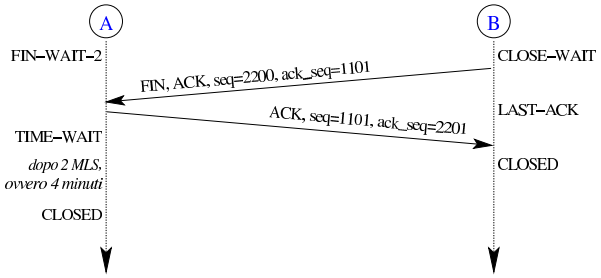
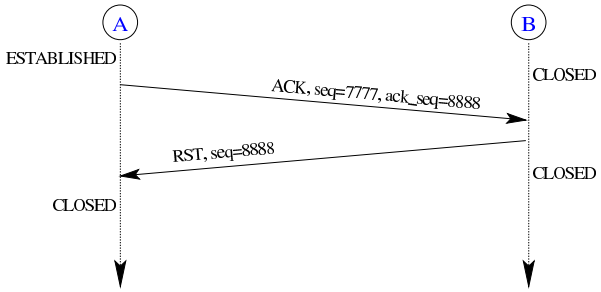


Figura 83.168. Dopo la chiusura dal lato «A», e dopo che il lato «B» ha finito con la propria trasmissione, anche questo decide di chiudere e si arriva alla conclusione della connessione. Tuttavia, il lato «A» che alla fine non può sapere se il lato «B» ha ricevuto effettivamente la conferma, rimane 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.

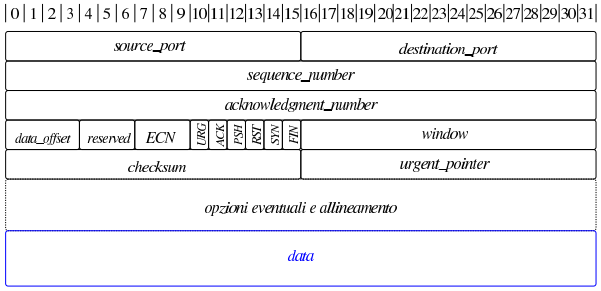


Figura 83.171. Pseudo-intestazione IPv4, utile per il calcolo del codice di controllo TCP, ma non facente parte del pacchetto.

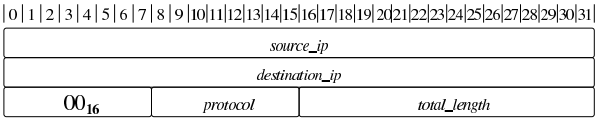


Tabella 83.172. Campi dell'intestazione TCP e della pseudo-intestazione relativa al protocollo IPv4.

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* Il contenuto di questo campo è definito dal RFC 3160. In condizioni normali, questo campo viene lasciato a zero.
URG	1 bit	*urgent* 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* 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* 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* 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* Se l'indicatore è attivo significa che si sta tentando di instaurare una connessione.
FIN	1 bit	*finalization* 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 ***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.

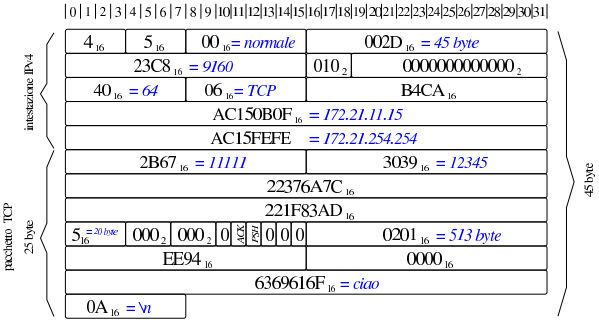
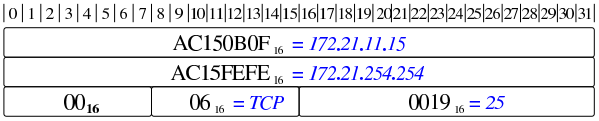


Figura 83.176. Pseudo-intestazione relativa al pacchetto di esempio della figura precedente.



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.

Studio per un sistema a 32 bit

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

84.1 Introduzione a os32

os32 è uno studio che applica qualche rudimento relativo ai sistemi operativi, basandosi sull'architettura x86-32 IBM AT, utilizzando come strumenti di sviluppo GNU C, GNU AS e GNU LD, su un sistema GNU/Linux. Il risultato non è un sistema operativo utilizzabile, ma una struttura su cui poter fare esperimenti e di cui è possibile mostrare (in termini tipografici) ed eventualmente descrivere ogni riga di codice.

os32 contiene uno schedulatore banale e molto limitato, un'organizzazione dei processi ad albero e una funzionalità di amministrazione dei segnali, una gestione del file system Minix 1 e delle partizioni in stile Dos (ma solo di quelle primarie), una shell banale e qualche programma di servizio di esempio.

La differenza più importante di os32, rispetto a os16, sta nel fatto che non si utilizzano più le funzioni del BIOS tradizionale, dal momento che si opera in modalità protetta.

84.1.1 Organizzazione

Tutti i file di os32 dovrebbero essere disponibili a partire da *allegati/os32* . In particolare, il file 'disk.hda' è l'immagine di un vecchio disco PATA, suddiviso in due partizioni: la prima partizione con un file system Dos-FAT, contenente SYSLINUX e il kernel di os32; la seconda contenente un file system Minix 1 con il sistema.

Gli script preparati per os32 prevedono che i file system contenuti nel file-immagine che rappresentano l'unità PATA, in fase di sviluppo si trovino innestati nelle directory '`/mnt/disk.hda.1/`' e '`/mnt/disk.hda.2/`'. Pertanto, se si ricompila os32, tali directory vanno predisposte (oppure vanno modificati gli script con l'organizzazione che si preferisce attuare). Tuttavia, considerato che non è facile lavorare con file-immagine suddivisi in partizioni, altri script aiutano nelle operazioni di manutenzione: '`fdisk`', '`format`', '`mount`', '`umount`'.

Per la verifica del funzionamento del sistema, è previsto l'uso equivalente di Bochs o di Qemu. Per questo scopo sono disponibili gli script '`bochs`' e '`qemu`' (rispettivamente i listati 94.1.2 e 94.1.9), con le opzioni necessarie a operare correttamente.

Per la compilazione del lavoro si usano due script alternativi: '`makeit.mer`' o '`makeit.sep`' (listato 94.1.8). Lo script '`makeit.mer`' conduce una compilazione in cui i file eseguibili degli applicativi sono tali da condividere lo stesso segmento di memoria, sia per il codice, sia per i dati; al contrario, lo script '`makeit.sep`' fa sì che codice e dati siano distinti (il kernel si compila solo in formato ELF). I due script ricreano ogni volta i file-make, basandosi sui file presenti effettivamente nelle varie directory previste; inoltre, alla fine della compilazione, copiano il kernel nella prima partizione del file-immagine del disco PATA (purché risulti innestata come previsto nella directory '`/mnt/disk.hda.1/`') e nella seconda partizione copiano gli applicativi.

Va osservato che il lavoro si basa su un file system Minix 1 (sezione 68.7) perché è molto semplice, ma soprattutto, la prima versione è quella che può essere utilizzata facilmente in un sistema operativo GNU/Linux (sul quale avviene lo sviluppo di os32). È bene sottolineare che si tratta della versione con nomi da 14 caratteri, ovvero quella tradizionale del sistema operativo Minix, mentre nei sistemi GNU/Linux, la creazione predefinita di un file system del genere produce una versione particolare, con nomi da 30 caratteri.

84.1.2 Le directory

Gli script descritti nella sezione precedente, si trovano all'inizio della gerarchia prevista per os32. Le directory successive dividono in modo molto semplice le varie componenti per la compilazione:

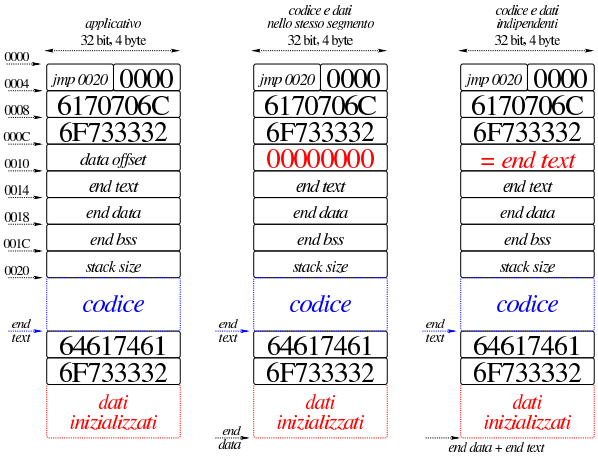
Directory	Contenuto
'`applic/`'	File delle applicazioni da usare con os32.
'`kernel/`'	File per la realizzazione del kernel, inclusi i file di intestazione specifici.
'`lib/`'	File di intestazione generali, file della libreria C per le applicazioni e, per quanto possibile, anche per il kernel.
'`skel/`'	Scheletro del file system complessivo, con i file di configurazione e le pagine di manuale.

La libreria C non è completa, limitandosi a contenere ciò che serve per lo stato di avanzamento attuale del lavoro. Si osservi che nella directory '`lib/_gcc/`' si collocano file contenenti una libreria di funzioni in linguaggio C, necessaria al compilatore GNU C per compiere il proprio lavoro correttamente con valori da 64 bit.

84.1.3 La struttura degli eseguibili

Nell'ottica della massima semplicità, gli eseguibili degli applicativi di os32 hanno una struttura propria, schematizzata dalla figura successiva. Tale struttura viene ottenuta attraverso i file sorgenti '`crt0.mer.s`' o '`crt0.sep.s`', e i file di configurazione di GNU LD '`applic.mer.ld`' o '`applic.sep.ld`'. La sigla '`*.mer.*`' individua la compilazione in un solo segmento, sia per il codice, sia per i dati, mentre la sigla '`*.sep.*`' riguarda la situazione opposta, in cui codice e dati si trovano divisi.

Figura 84.2. Struttura dei file eseguibili degli applicativi di os32.

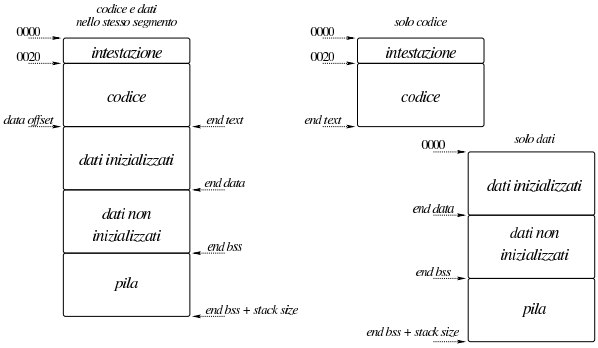


Nella figura si mettono a confronto la struttura dell'eseguibile di un applicativo compilato per avere codice e dati nello stesso segmento, rispetto al caso in cui questi sono separati. Nei primi quattro byte c'è un'istruzione di salto al codice che si trova subito dopo l'intestazione, quindi appare un'impronta di riconoscimento che occupa complessivamente otto byte. Tale impronta è la rappresentazione esadecimale della stringa «os32appl», ma spezzata in due e rovesciata a causa dell'architettura *little endian* (se si legge il file in esadecimale, si vede la sequenza dei caratteri «lppa23so»).

Dopo l'impronta di riconoscimento si trovano, rispettivamente, lo scostamento del segmento dati, espresso in byte, gli indirizzi conclusivi dell'area del codice, dei dati inizializzati e di quelli non inizializzati. Alla fine viene indicata la dimensione richiesta per la pila dei dati. Per distinguere se l'eseguibile è fatto per gestire in un solo segmento codice e dati, oppure se questi devono essere separati, va osservato il valore di ***data_offset***: se questo è zero, significa che il segmento dati parte dall'indirizzo zero, esattamente come il segmento codice, pertanto si trovano nello stesso spazio di indirizzamento. Se invece il valore di ***data_offset*** è diverso da zero, allora deve coincidere con il valore di ***end_text***, in quanto i dati inizializzati si trovano nel file a partire dalla fine del codice, ma devono poi collocarsi in un segmento separato, per cui il valore di ***end data*** è da intendere riferito all'indirizzo iniziale della zona dei dati, ovvero zero.

Nel file eseguibile, la porzione che contiene i dati inizializzati, parte con un'impronta ulteriore, costituita da «os32data», con gli stessi problemi di inversione già descritti per l'intestazione. Lo scopo di questa impronta è semplicemente quello di evitare che ci possano essere dati che iniziano precisamente dall'indirizzo zero, essendo questo riservato per il puntatore nullo.

Figura 84.3. Immagine in memoria dei processi generati dagli eseguibili di os32: a sinistra quelli in cui codice e dati condividono lo stesso segmento di memoria; a destra quelli che usano segmenti distinti.



Il kernel è in formato ELF, ma nella prima parte del codice vie-

ne piazzata un'impronta, secondo le specifiche multiboot (sezione 65.5.1).

Riquadro 84.4. Compilazione degli applicativi con codice e dati separati.

La compilazione degli applicativi che in memoria separano il segmento codice da quello dei dati avviene in modo più complicato rispetto all'altro metodo, perché codice e dati iniziano formalmente dall'indirizzo zero e non è possibile procedere a una compilazione «binaria» normale. Pertanto, in questo caso si crea inizialmente un formato ELF provvisorio; poi, attraverso lo script '`elf-to-os32`' che a sua volta si avvale di '**`objdump`**', vengono individuate le componenti che servono dal file ELF e ricomposte secondo il formato che si aspetta os32.

84.1.4 Tabelle

Nel codice del kernel si utilizzano spesso delle informazioni organizzate in memoria in forma di tabella. Si tratta precisamente di array, le cui celle sono costituite generalmente da variabili strutturate. Queste tabelle, ovvero gli array che le rappresentano, sono dichiarate come variabili pubbliche; tuttavia, per facilitare l'accesso ai rispettivi elementi e per uniformità di comportamento, viene abbinata loro una funzione, con un nome terminante per '`..._reference()`', con cui si ottiene il puntatore a un certo elemento della tabella, fornendo gli argomenti appropriati. Per esempio, la tabella degli inode in corso di utilizzazione viene dichiarata così nel file '`kernel/fs/inode_table.c`':

```
inode_t inode_table[INODE_MAX_SLOTS];
```

Successivamente, la funzione ***inode_reference()*** offre il puntatore a un certo inode:

```
inode_t *inode_reference (dev_t device, ino_t ino);
```

84.1.5 Guida di stile

Per cercare di dare un po' di uniformità al codice del kernel e a quello della libreria, dove possibile, i nomi delle variabili seguono una certa logica, riassunta dalla tabella successiva.

Tipo	Nome	Utilizzo
`inode_t *`	`inode` `inode_…`	Puntatore a un inode (puntatore a un elemento della tabella di inode).
`ino_t`	`ino` `ino_…`	Numero di inode, nell'ambito di un certo super blocco (ammesso che sia abbinato effettivamente a un dispositivo).
`int`	`fdn` `fdn_…`	Numero del descrittore di un file (indice all'interno della tabella dei descrittori).
`fd_t *`	`fd` `fd_…`	Puntatore a un descrittore di file (puntatore a un elemento della tabella di descrittori).
`int`	`fno` `fno_…`	Numero del file di sistema (indice all'interno della tabella dei file di sistema).
`zno_t`	`zone` `zone_…`	Numero assoluto di una «zona» del file system Minix.
`zno_t`	`fzone` `fzone_…`	Numero relativo di una «zona» del file system Minix. In questo caso, il numero della zona è relativo al file, dove la prima zona del file ha il numero zero.

`off_t`	`offset`
	`offset_…`
	`off_…`
		Scostamento, secondo il significato del tipo derivato '**`off_t`**'.

Tipo	Nome	Utilizzo
`size_t` `ssize_t`	`size` `size_…`	Dimensione, secondo il significato dei tipi derivati '**`size_t`**' o '**`ssize_t`**'.
`size_t` `ssize_t`	`count` `count_…`	Quantità, quando il tipo '**`size_t`**' è appropriato.
`blkcnt_t`	`blkcnt` `blkcnt_…`	Quantità espressa in blocchi del file system (in questo caso, trattandosi di un file system Minix 1, si intendono zone).
`blksize_t`	`blksize` `blksize_…`	Dimensione del blocco del file system, espressa in byte (in questo caso, trattandosi di un file system Minix 1, si intende la dimensione della zona).
`int`	`fno` `fno_…`	Numero di file system.
`int`	`oflags` `oflags_…`	Opzioni relative all'apertura di un file, annotate nella tabella dei file di sistema: indicatori di sistema.
`int`	`status` `status_…`	Valore intero restituito da una funzione, quando la risposta contiene solo l'indicazione di un successo o di un insuccesso.
`void *`	`pstatus`	Puntatore restituito da una funzione, quando interessa sapere solo se si tratta di un esito valido.
`char *`	`path` `path_…`	Percorso del file system.
`dev_t`	`device` `device_…`	Numero di dispositivo, contenente sia il numero primario, sia quello secondario (*major*, *minor*).
`int`	`n` `n_…`	Dimensione di qualcosa, di tipo '**`int`**'.
`char *`	`string` `string_…`	Area di memoria da considerare come stringa.
`void *`	`buffer` `buffer_…`	Area di memoria destinata ad accogliere un'informazione di tipo imprecisato.
`int`	`n` `n_…`	Dimensione o quantità di qualcosa, espressa attraverso il tipo '**`int`**'.
`int`	`c` `c_…`	Un carattere senza segno trasformato nel tipo '**`int`**'.
`struct stat`	`st` `st_…`	Variabile strutturata usata per rappresentare lo stato di un file, secondo il tipo '**`struct stat`**'.
`FILE *`	`fp` `fp_…`	Puntatore che rappresenta un flusso di file.
`DIR *`	`dp` `dp_…`	Puntatore che rappresenta un flusso relativo a una directory.

Tipo	Nome	Utilizzo
`struct dirent`	`dir` `dir_…`	Variabile strutturata contenente le informazioni su una voce di una directory.
`struct password`	`pws` `pws_…`	Variabile strutturata contenente le informazioni di una voce del file ‘`/etc/passwd`’.
`struct tm`	`tms` `tms_…`	Variabile strutturata contenente le componenti di un orario.
`struct tm *`	`timeptr` `timeptr_…`	Puntatore a una variabile strutturata contenente le componenti di un orario.

84.1.6 Tipi derivati speciali

Nel codice del kernel e nella libreria specifica di os32 si usano dei tipi derivati speciali, riassunti nella tabella successiva.

File di intestazione	Tipo speciale	Descrizione
‘`kernel/fs.h`’	`zno_t`	Variabile scalare, per rappresentare un numero di una zona, secondo la terminologia del file system Minix.
‘`kernel/fs.h`’	`sb_t`	Variabile strutturata, adatta a contenere tutte le informazioni di un super blocco, relativo a un dispositivo di memorizzazione innestato.
‘`kernel/fs.h`’	`inode_t`	Variabile strutturata, adatta a contenere tutte le informazioni di un inode aperto nel sistema.
‘`kernel/fs.h`’	`file_t`	Variabile strutturata, adatta a contenere i dati di un file di sistema.
‘`kernel/fs.h`’	`fd_t`	Variabile strutturata, adatta a contenere i dati di un descrittore di file, ovvero del file di un certo processo elaborativo.
‘`kernel/fs.h`’	`directory_t`	Variabile strutturata, adatta a contenere una voce di una directory.
‘`kernel/ibm_i386.h`’	`gdt_t`	Variabile strutturata, adatta a contenere le informazioni di una voce della tabella GDT.
‘`kernel/ibm_i386.h`’	`idt_t`	Variabile strutturata, adatta a contenere le informazioni di una voce della tabella IDT.
‘`kernel/ibm_i386.h`’	`idtr_t`	Variabile strutturata, adatta a rappresentare il registro IDTR.
‘`kernel/memory.h`’	`addr_t`	Variabile scalare, in grado di rappresentare un indirizzo efficace di memoria (un indirizzo che vada da 00000000_{16} a $FFFFFFFF_{16}$).
‘`kernel/multiboot.h`’	`multiboot_t`	Variabile strutturata che riproduce la scomposizione delle informazioni ricevute dal kernel dal sistema di avvio multiboot.
‘`kernel/proc.h`’	`proc_t`	Variabile strutturata per rappresentare un elemento della tabella dei processi.

File di intestazione	Tipo speciale	Descrizione
‘`kernel/proc.h`’	`header_t`	Variabile strutturata che riproduce la suddivisione delle informazioni contenute nella parte iniziale degli eseguibili di os32.
‘`kernel/driver/ata.h`’	`ata_t`	Variabile strutturata per rappresentare un elemento della tabella delle unità ATA.
‘`kernel/driver/ata.h`’	`ata_sector_t`	Variabile strutturata per rappresentare un settore di dati relativo alla gestione ATA.
‘`kernel/driver/kbd.h`’	`kbd_t`	Variabile strutturata per rappresentare complessivamente lo stato della tastiera, incorporando anche la modalità di interpretazione corretta della mappa attuale.
‘`kernel/driver/screen.h`’	`screen_t`	Variabile strutturata per rappresentare il contenuto di uno schermo e la collocazione del cursore; tale variabile strutturata compone un elemento della tabella degli schermi gestiti simultaneamente.
‘`kernel/driver/tty.h`’	`tty_t`	Variabile strutturata, adatta a contenere le informazioni e lo stato di un terminale, che costituisce in pratica un elemento della tabella dei terminali.
‘`kernel/blk.h`’	`blk_cache_t`	Variabile strutturata, adatta a contenere un blocco di memoria (per i dispositivi a blocchi) e le informazioni che lo riguardano, come elemento della tabella che costituisce la memoria tampone per l’accesso alle unità di memorizzazione.
‘`lib/sys/os32.h`’	`h_addr_t`	Variabile scalare a 32 bit, contenente un indirizzo IPv4 in *host byte order*, ovvero rappresentato secondo l’achitettura della CPU, per ciò che riguarda l’ordine dei byte.

84.2 Caricamento ed esecuzione del kernel

Il kernel di os32 è compilato in formato ELF, secondo le specifiche multiboot, in modo da poter essere avviato da un sistema come GRUB 1 o SYSLINUX. Il codice del kernel inizia nel file ‘`crt0.s`’, dove a un certo punto viene eseguita la funzione ***kmain()***, nella quale si sintetizza il funzionamento del kernel stesso. Il sistema di avvio colloca il kernel a partire dall’indirizzo 100000_{16} (1 Mibyte), già in modalità protetta, di conseguenza il codice è organizzato per iniziare da tale posizione.

84.2.1 Multiboot

os32 è conforme alle specifiche multiboot per consentirne l’avvio attraverso GRUB 1 o SYSLINUX, senza doversi prendere carico dei problemi relativi al passaggio alla modalità protetta. Perché il file del kernel sia riconosciuto come aderente a tali specifiche, contiene un’impronta di riconoscimento, definita *multiboot header*, collocata nella parte iniziale, come dichiarato nel file ‘`kernel/main/crt.s`’, entro i primi 8 Kibyte.

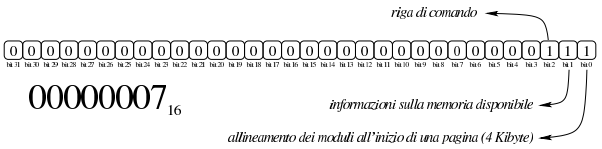
Figura 84.9. La prima parte obbligatoria dell'intestazione multiboot.

Il primo campo da 32 bit, definito ***magic***, contiene l'impronta di riconoscimento vera e propria, costituita precisamente dal numero $1BADB002_{16}$. Il secondo campo da 32 bit, definito ***flags***, contiene degli indicatori con i quali si richiede un certo comportamento al sistema di avvio. Il terzo campo da 32 bit, definito ***checksum***, contiene un numero calcolato in modo tale che la somma tra i numeri contenuti nei tre campi da 32 bit porti a ottenere zero, senza considerare i riporti.

I nomi indicati sono quelli definiti dallo standard e, come si vede, il campo ***checksum*** si ottiene calcolando −(***magic*** + ***flags***), dove si deve intendere che i calcoli avvengono con valori interi senza segno e si ignorano i riporti.

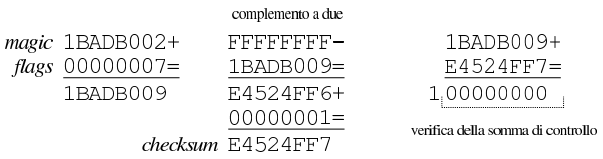
Dal momento che il kernel da avviare è in formato ELF, le informazioni che il sistema di avvio necessita per piazzarlo correttamente in memoria e per passare il controllo allo stesso, sono già disponibili e non c'è la necessità di occuparsi di altri campi facoltativi che possono seguire i tre già descritti. Stante questa semplificazione, per quanto riguarda il campo ***flags***, os32 utilizza precisamente il valore 00000003_{16}, con il significato che si vede nella figura successiva.

Figura 84.10. Il campo ***flags*** e il suo utilizzo fondamentale.



Il bit meno significativo del campo ***flags***, se impostato a uno, serve a richiedere il caricamento in memoria dei moduli eventuali (assieme al file-immagine principale) in modo che risultino allineati all'inizio di una «pagina» (ovvero all'inizio di un blocco da 4 Kibyte). os32 non prevede moduli, tuttavia richiede ugualmente questa opzione. Il secondo bit del campo ***flags*** serve a richiedere al sistema di avvio di passare le informazioni disponibili sulla memoria, le quali poi vengono rese disponibili a partire da un'area a cui punta inizialmente il registro ***EBX***.

Figura 84.11. Calcolo del campo ***checksum***.



Quando il sistema di avvio passa il controllo al kernel, dopo averlo caricato in memoria: il microprocessore è in modalità protetta; il registro ***EAX*** contiene il numero $2BADB002_{16}$; il registro ***EBX*** contiene l'indirizzo fisico, a 32 bit, di una sequenza di campi contenenti informazioni passate dal sistema di avvio (*multiboot information structure*), come si vede nella figura successiva.

Figura 84.12. Inizio della struttura di informazioni offerta da un sistema di avvio aderente alle specifiche *multiboot*.

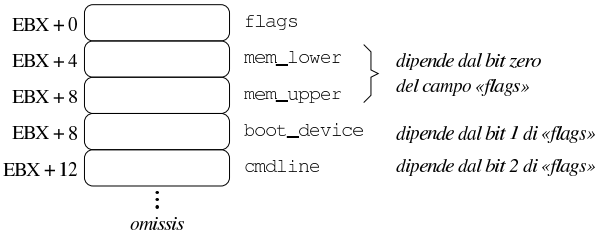


Tabella 84.13. Descrizione dei primi campi della struttura informativa fornita dal sistema di avvio *multiboot*.

Nome mnemonico del campo	bit del campo '**flags**' da cui dipende	Descrizione
`flags`		Il primo campo definisce degli indicatori, con i quali si dichiara se una certa informazione, successiva, viene fornita ed è valida.
`mem_lower` `mem_upper`	0	Se è attivo il bit meno significativo del campo '**flags**', i campi '**mem_lower**' e '**mem_upper**' contengono la dimensione della memoria bassa (da zero a un massimo di 640 Kibyte) e della memoria alta (quella che si trova a partire da un mebibyte). La dimensione è da intendersi in kibibyte (simbolo Kibyte) e, per quanto riguarda la memoria alta, viene indicata solo la dimensione continua fino al primo «buco».
`boot_device`	1	Se è attivo il secondo bit, partendo dal lato meno significativo, il campo '**boot_device**' dà informazioni sull'unità di avvio. L'informazione è divisa in quattro byte, come descritto nelle specifiche *multiboot*.
`cmdline`	2	Se è attivo il terzo bit, partendo dal lato meno significativo, il campo '**cmdline**' contiene l'indirizzo iniziale di una stringa che riproduce la riga di comando passata al kernel.

Come si può intuire leggendo la tabella che descrive i primi cinque campi, il significato dei bit del campo '**flags**' viene attribuito, mano a mano che l'aggiornamento delle specifiche prevede l'espansione della struttura informativa. Per esempio, un campo '**flags**' con il valore 100_2 sta a significare che esistono i campi fino a '**cmdline**' e il contenuto di quelli precedenti non è valido, ma i campi successivi, non esistono affatto. La comprensione di questo concetto dovrebbe rendere un po' più semplice la lettura delle specifiche.

Tabella 84.14. Funzioni per la gestione delle specifiche multiboot all'interno di os32.

Funzione	Descrizione
`void` `mboot_save (multiboot_t *`***`mboot_data`***`);`	Salva le informazioni multiboot all'interno della variabile strutturata pubblica ***multiboot***. Listati 94.11, 94.11.2 e 94.11.3.

Funzione	Descrizione

`char **`
`mboot_cmdline_opt (const char *`***opt***`,`
`const char *`***delim***`);`
	Scandisce la stringa delle opzioni salvata all'interno di ***multiboot.cmdline***, alla ricerca di un'opzione il cui nome corrisponda alla stringa. Dopo il nome dell'opzione deve apparire il segno '`=`' e dopo devono trovarsi i valori associati all'opzione, separati da ***delim***. Questi valori vengono restituiti in forma di array di stringhe, dove l'ultima stringa si riconosce perché vuota. Listati 94.11, 94.11.2 e 94.11.1.

84.2.2 File «kernel.ld», «kernel/main/crt0.s» e «kernel/main/stack.s»

Listati 94.1.7, 94.9.2 e 94.9.6.

Il codice del kernel inizia dal file '`crt.s`'; tuttavia, per la sua corretta interpretazione, va considerato prima il file di configurazione di GNU LD (il collegatore, ovvero il *linker*), costituito dal file '`kernel.ld`', sintetizzabile così:

```
ENTRY (kstartup)
SECTIONS {
    . = 0x00100000;
    …
}
```

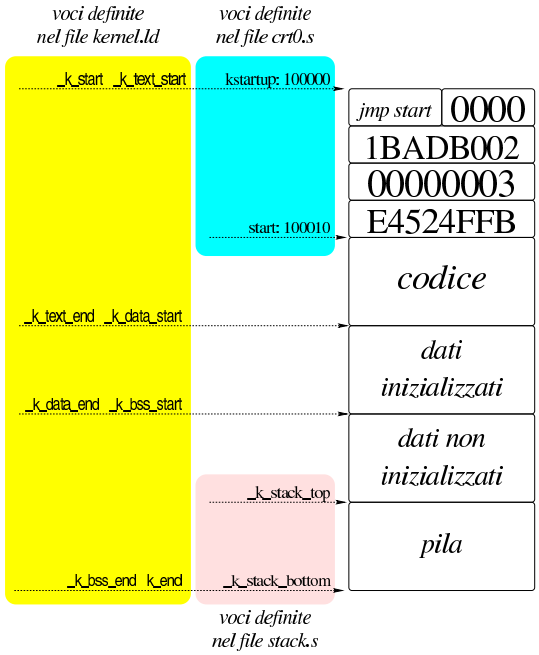
Si osserva subito che il punto di inizio del codice, descritto successivamente dal file '`crt.s`', deve corrispondere alla posizione dell'etichetta '**`kstartup`**' e che quel punto deve trovarsi all'indirizzo 100000_{16}, ovvero quello in cui il sistema di avvio lo colloca.

Nel file '`crt.s`', dopo il preambolo in cui si dichiarano i simboli esterni e quelli interni da rendere pubblici, si parte proprio con l'etichetta '**`kstartup`**', e da lì si salta a un'altra posizione ('**`start`**'), per lasciare spazio all'intestazione multiboot.

```
…
.section .text
kstartup:
    jmp start
.align 4
multiboot_header:
    .int 0x1BADB002                     # magic
    .int 0x00000003                     # flags
    .int -(0x1BADB002 + 0x00000003)     # checksum
start:
    …
```

L'immagine del kernel in memoria utilizza un solo segmento per codice e dati, suddividendosi nel modo consueto: codice, dati inizializzati, dati non inizializzati e pila. Per individuare le varie componenti, il file '`kernel.ld`' inserisce dei nomi a cui è possibile fare riferimento nel codice; inoltre, viene utilizzato il file '`stack.s`' per definire lo spazio usato per la pila dei dati.

Figura 84.17. Immagine del kernel in memoria, a partire dall'indirizzo 100000_{16}, evidenziando le etichette dichiarate nei file '`kernel.ld`', '`crt0.s`' e '`stack.s`'.



A partire dall'indirizzo corrispondente all'etichetta '**`start`**', nel file '`crt0.s`' inizia il lavoro preliminare del kernel. Per prima cosa viene attivata la pila dei dati, collocando nel registro ***ESP*** l'indirizzo corrispondente alla fine della stessa, ovvero '**`_k_stack_bottom`**':

```
    movl $_k_stack_bottom, %esp
```

Quindi si azzera il registro EFLAGS, sfruttando per questo la pila appena attivata:

```
    pushl $0
    popf
```

Infine si chiama la funzione ***kmain()*** (del file '`kmain.c`'), fornendo come argomenti la firma di riconoscimento del sistema multiboot, contenuta nel registro EAX, e il puntatore alla struttura contenente le informazioni fornite dal sistema multiboot, contenuto nel registro EBX:

```
    pushl %ebx  # multiboot_t *info;
    pushl %eax  # uint32_t     magic;
    call kmain  # void kmain (uint32_t magic,
                #             multiboot_t *info);
```

Se ci dovesse essere una conclusione della funzione ***kmain()***, si passerebbe al codice successivo, il quale si limita a mettere a riposo la CPU:

```
halt:
    hlt
    jmp halt
```

84.2.3 File «kernel/main.h» e «kernel/main/*»

Listato 94.9 e successivi.

Tutto il lavoro del kernel di os32 si sintetizza nella funzione ***kmain()***, contenuta nel file '`kernel/main/kmain.c`'. Per poter dare un significato a ciò che vi appare al suo interno, occorre conoscere tutto il resto del codice, ma inizialmente è utile avere un'idea di ciò che succede, se poi si vuole compilare ed eseguire il sistema operativo.

La funzione si chiama ***kmain()*** (e non ***main()***), perché non è conforme allo schema che dovrebbe avere la prima funzione di un programma per sistemi POSIX. Come già accennato a proposito del file '`crt0.s`', la funzione ***kmain()*** prevede come parametri un codi-

ce di riconoscimento e il puntatore a delle informazioni, forniti dal sistema di avvio.

```
…
void
kmain (uint32_t magic, multiboot_t *mboot_data)
{
  …
  tty_init ();
  if (magic == 0x2BADB002)
    {
      mboot_save (mboot_data);
      k_printf ("os32 build %s ram %i Kibyte\n",
                BUILD_DATE, (int) multiboot.mem_upper);
      mb_size (multiboot.mem_upper * 1024);
      kbd_load ();
      blk_cache_init ();
      fs_init ();
      proc_init ();
    }
  else
    {
      …
      k_exit ();
    }
  menu ();
  …
}
```

Dopo la dichiarazione delle variabili si inizializza la gestione del video della console (funzione ***tty_init()***), si verifica che il codice sia stato avviato da un sistema di avvio multiboot e se ne salvano le informazioni (funzione ***mboot_save()***), quindi si mostra un messaggio iniziale, si imposta la dimensione massima della memoria disponibile in base ai dati ottenuti dal sistema multiboot (funzione ***mb_size()***), si configura la tastiera (funzione ***kbd_load()***), si inizializza la gestione della memoria tampone (funzione ***blk_cache_init()***), del file system (funzione ***fs_init()***) e dei processi elaborativi (***proc_init()***). Fatto tutto questo appare un menù (funzione ***menu()***) e si passa a una fase successiva.

```
…
void
kmain (uint32_t magic, multiboot_t *mboot_data)
{
  …
  menu ();
  for (exit = 0; exit == 0;)
    {
      sys (SYS_0, NULL, 0);
      …
      if …
        …
      else if (strncmp (command, "h", MAX_CANON) == 0)
        {
          menu ();
        }
      else if (strncmp (command, "x", MAX_CANON) == 0)
        …
      else if (strncmp (command, "q", MAX_CANON) == 0)
        {
          k_printf ("System halted!\n");
          return;
        }
    }
}
```

A questo punto il kernel ha concluso le sue attività preliminari e, per motivi diagnostici, mostra un menù, quindi inizia un ciclo in cui ogni volta esegue una chiamata di sistema nulla e poi legge un comando dalla tastiera, costituito però da un solo carattere: se risulta selezionato un comando previsto, il kernel esegue quanto richiesto e poi riprende il ciclo. La chiamata di sistema nulla serve a far sì che lo schedulatore ceda il controllo a un altro processo, ammesso che questo esista, consentendo l'avvio di processi ancor prima di avere messo in funzione quel processo che deve svolgere il ruolo di '**`init`**'.

In generale le chiamate di sistema sono fatte per essere usate solo dalle applicazioni; tuttavia, in pochi casi speciali il kernel le deve utilizzare come se fosse proprio un'applicazione. Qui si rende necessario l'uso della chiamata nulla, perché quando è in funzione il codice del kernel non ci possono essere interruzioni esterne e quindi nessun altro processo verrebbe messo in condizione di funzionare.

Le funzioni principali disponibili in questa modalità diagnostica sono riassunte nella tabella successiva:

Comando	Risultato
`h`	Mostra il menù di funzioni disponibili.
`t`	Mostra i valori gestiti internamente dell'orologio del kernel.
`f`	Esegue una biforcazione del kernel, nella quale, il processo figlio si limita a mostrare ripetutamente il proprio numero di processo.
`g` `G`	Mostra le prime voci della tabella GDT, in binario.
`i` `I`	Mostra le prime voci della tabella IDT, in binario.
`m`	Mostra la mappa della memoria, elencando le aree continue utilizzate.
`p`	Mostra la situazione dei processi e altre informazioni.
`s`	Mostra delle informazioni sul super blocco.
`n`	Mostra l'elenco degli inode attivi.
`1`	Invia il segnale '**`SIGKILL`**' al processo numero uno.
`2`|`3`|`4`|`5`| `6`|`7`|`8`|`9`|`A`| `B`|`C`|`D`|`E`|`F`	Invia il segnale '**`SIGTERM`**' al processo con il numero corrispondente, da 2 a 15.
`a`|`b`|`c`	Avvia il programma '`/bin/aaa`', '`/bin/bbb`' o '`/bin/ccc`'.
`x`	Termina il ciclo e successivamente si passa all'avvio di '`/bin/init`'.
`q`	Ferma il sistema.

Premendo [*x*], il kernel avvia '`/bin/init`', quindi si mette in un altro ciclo, dove si limita a passare ogni volta il controllo allo schedulatore, attraverso la chiamata di sistema nulla.

```
      else if (strncmp (command, "x", MAX_CANON) == 0)
        {
          exec_argv[0] = "/bin/init";
          exec_argv[1] = NULL;
          pid = run ("/bin/init", exec_argv, NULL);
          while (1)
            {
              sys (SYS_0, NULL, 0);
            }
        }
```

Figura 84.26. Aspetto di os32 in funzione mentre visualizza la tabella dei processi avviati e la mappa della memoria.

```
c
abaabaaba
p
pp  p pg                            T * 0x1000 D * 0x1000 stack
id id rp  tty  uid euid suid usage s addr  size addr  size pointer  name
 0  0  0 0000    0    0    0 00.03 R 00000 028e 00000 0000 028eb2c os32 kernel
 0  1  0 0000    0    0    0 00.09 r 0051e 000e 0052c 002d 002cf88 /bin/ccc
 1  2  0 0000   10   10   10 00.00 s 002bc 000e 002ca 002d 002cf34 /bin/aaa
 1  3  0 0000   11   11   11 00.00 s 002f7 000e 00305 002d 002cf34 /bin/bbb
ab
m
Hex mem map, blocks of 1000: 0-28f 2bc-332 51e-559
aabaab_
```

Figura 84.27. Aspetto di os32 in funzione con il menù in evidenza, dopo aver dato il comando '**x**' per avviare '**init**'.

```
os32 build 20AAMMGGHHmm ram 130048 Kibyte
[ata_init] ATA drive 0 size 8064 Kib
[ata_drq] ERROR: drive 2 error
[dm_init] ATA drive=0 total sectors=16128
[dm_init] partition type=0c start sector=63 total sectors=2961
[dm_init] partition type=81 start sector=3024 total sectors=13104
.--------------------------------------------------------------------.
| h    show this menu                              .--------------.|
| t    show internal timer values                  | all commands ||
| f    fork the kernel                             | followed by  ||
| m    memory map (HEX)                            | [Enter]      ||
| g|G  show GDT table first 21+21 items            '--------------'|
| i|I  show IDT table first 21+21 items                             |
| p    process status list                                        |
| s    super block list                                           |
| n    list of active inodes                                      |
| 1..9 kill process  1 to 9                                       |
| A..F kill process 10 to 15                                      |
| a..c run programs '/bin/aaa' to '/bin/ccc' in parallel          |
| x    exit interaction with kernel and start '/bin/init'         |
| q    quit kernel                                                |
'--------------------------------------------------------------------'
x
init
os32: a basic os. [Ctrl q], [Ctrl r], [Ctrl s], [Ctrl t] to change
console.
This is terminal /dev/console0
Log in as "root" or "user" with password "ciao" :-)
login:
```

84.3 Funzioni interne legate all'hardware, nei file «kernel/ibm_i386.h» e «kernel/ibm_i386/...»

Listato 94.6 e successivi.

Il file '`kernel/ibm_i386.h`' e quelli contenuti nella directory '`kernel/ibm_i386/`', raccolgono il codice del kernel che è legato strettamente all'hardware, escludendo però la gestione dei dispositivi. Tra le altre spiccano particolarmente le funzioni per la gestione dei segmenti di memoria (la tabella GDT), delle interruzioni (la tabella IDT) e l'attivazione delle routine associate alle interruzioni (ISR).

Alcune delle funzioni scritte in linguaggio assemblatore hanno nomi che iniziano con un trattino basso, ma a fianco di queste sono disponibili delle macroistruzioni, con nomi equivalenti senza il trattino basso iniziale, per garantire che gli argomenti della chiamata abbiano il tipo corretto, restituendo un valore intero «normale», quando qualcosa deve essere restituito.

84.3.1 Funzioni per l'input e l'output con le porte interne

Alcune funzioni e macroistruzioni di questo gruppo sono destinate a facilitare l'input e l'output con le porte interne dell'architettura x86.

Tabella 84.28. Funzioni e macroistruzioni per l'input e l'output con le porte interne x86.

Funzione o macroistruzione	Descrizione
`uint32_t _in_8 (uint32_t port);` `unsigned int in_8 (port);`	Legge un valore a 8 bit da una porta. Listati 94.6 e 94.6.3.
`uint32_t _in_16 (uint32_t port);` `unsigned int in_16 (port);`	Legge un valore a 16 bit da una porta. Listati 94.6 e 94.6.1.
`void _out_8 (uint32_t port, uint32_t value);` `void out_8 (port, value);`	Scrive un valore a 8 bit in una porta. Listati 94.6 e 94.6.6.
`void _out_16 (uint32_t port, uint32_t value);` `void out_16 (port, value);`	Scrive un valore a 16 bit in una porta. Listati 94.6 e 94.6.4.

84.3.2 Funzioni accessorie alla gestione delle interruzioni hardware

Alcune funzioni di questo gruppo sono destinate a facilitare il controllo delle interruzioni hardware che raggiungono la CPU.

Tabella 84.29. Funzioni accessorie per il controllo delle interruzioni hardware.

Funzione o macroistruzione	Descrizione
`void cli (void);`	Disabilita le interruzioni hardware attraverso l'azzeramento dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.7.
`void sti (void);`	Abilita le interruzioni hardware attraverso l'attivazione dell'indicatore relativo nel registro EFLAGS. Listati 94.6 e 94.6.27.
`void irq_on (unsigned int irq);`	Abilita selettivamente l'interruzione hardware indicata per numero (da zero a 16). Listati 94.6 e 94.6.20.
`void irq_off (unsigned int irq);`	Disabilita selettivamente l'interruzione hardware indicata per numero (da zero a 16). Listati 94.6 e 94.6.19.

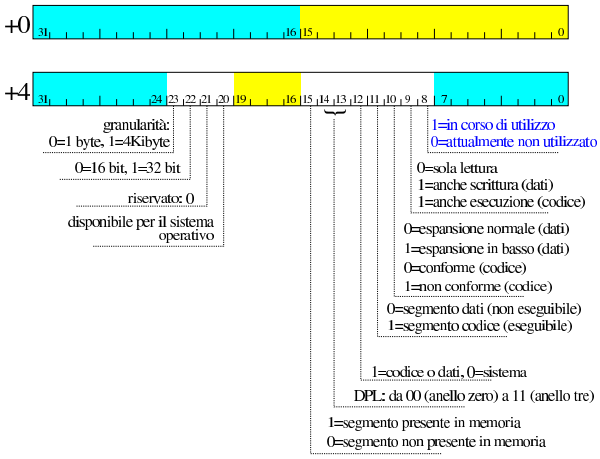
84.3.3 Gestione della tabella GDT

Nel momento in cui il codice del kernel prende il controllo, il microprocessore si trova già a funzionare in modalità protetta, attraverso una tabella GDT già impostata per gestire la memoria in modo lineare, senza particolari accorgimenti. Quando il kernel inizializza la gestione dei processi (funzione ***proc_init()***) costruisce una nuova tabella GDT, nella quale, per ogni processo gestibile, predispone due elementi, per descrivere rispettivamente il segmento codice e il segmento dati di un processo. In pratica, la nuova tabella GDT è composta da una prima voce nulla, obbligatoria, da una coppia di voci che descrivono il segmento codice e dati del kernel, da altre coppie di voci, modificate poi durante il funzionamento, per descrivere i segmenti dei processi.

Tutti i processi vedono la memoria con un indirizzamento che corrisponde a quello reale; tuttavia, disponendo ognuno di una propria coppia di voci nella tabella GDT, è possibile controllarne l'uso in modo da impedire che possano raggiungere aree al di fuori della propria competenza.

La tabella GDT è rappresentata in C dall'array ***gdt_table[]*** dichiarato nel file '`kernel/ibm_i386/gdt_public.c`' (listato 94.6.11), composto da elementi di tipo '**gdt_t**' (listato 94.6).

```
typedef struct {
    uint32_t  limit_a              : 16,
              base_a               : 16;
    uint32_t  base_b               : 8,
              accessed             : 1,
              write_execute        : 1,
              expansion_conforming : 1,
              code_or_data         : 1,
              code_data_or_system  : 1,
              dpl                  : 2,
              present              : 1,
              limit_b              : 4,
              available            : 1,
              reserved             : 1,
              big                  : 1,
              granularity          : 1,
              base_c               : 8;
} gdt_t;
```



La tabella viene creata con una quantità di elementi pari al valore della macro-variabile ***GDT_ITEMS***. Sapendo che la prima voce è obbligatoriamente nulla, che se ne usano altre due per il kernel e che ogni processo utilizza due voci della tabella, si possono gestire al massimo (***GDT_ITEMS***−3)/2 processi.

La struttura di ogni elemento della tabella GDT è molto complessa, pertanto, per scriverci un nuovo valore si usa la funzione ***gdt_segment()*** che si occupa di spezzettare e ricollocare i dati come richiesto dal microprocessore (listato 94.6.12)

Tabella 84.32. Funzioni per la gestione della tabella GDT.

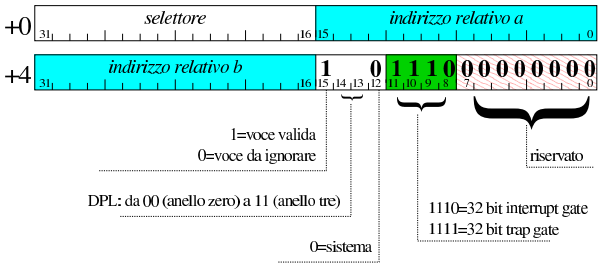
Funzione	Descrizione
`void gdt_segment (int segment, uint32_t base, uint32_t limit, bool present, bool code, unsigned char dpl);`	Scrive una voce della tabella GDT, sezionando e ricomponendo i dati come richiesto dal microprocessore. Listato 94.6.12.
`void gdt (void);`	Predispone e attiva la tabella GDT; per questo si avvale in modo particolare delle funzioni ***gdt_segment()*** e di ***gdt_load()***. Listato 94.6.8.
`void gdt_load (void *gdtr);`	Fa sì che il microprocessore carichi la tabella GDT, a partire dal puntatore al registro GDTR; registro che contiene l'informazione della collocazione in memoria della tabella GDT e della sua estensione. Listato 94.6.9.
`void gdt_print (void *gdtr, unsigned int first unsigned int last);`	Funzione diagnostica, usata per visualizzare il contenuto della tabella GDT in binario. Listato 94.6.10.

84.3.4 Gestione della tabella IDT

La tabella IDT serve al microprocessore per conoscere quali procedure avviare al verificarsi delle interruzioni. La funzione ***idt()*** si occupa di predisporre la tabella e di attivarla, ma prima di ciò si prende cura di posizionare le interruzioni hardware a partire dalla voce 32 (la 33-esima). Le procedure a cui fa riferimento la tabella IDT creata con la funzione ***idt()*** sono dichiarate nel file '`kernel/ibm_i386/isr.s`', descritto però nella sezione successiva.

La tabella IDT è rappresentata in C dall'array ***idt_table[]*** dichiarato nel file '`kernel/ibm_i386/idt_public.c`' (listato 94.6.18), composto da elementi di tipo '**`idt_t`**' (listato 94.6).


```
typedef struct {
    uint32_t  offset_a : 16,
              selector : 16;
    uint32_t  filler   : 8,
              type     : 4,
              system   : 1,
              dpl      : 2,
              present  : 1,
              offset_b : 16;
} idt_t;
```



La tabella viene creata con 129 elementi, anche se più della metà non vengono usati; tuttavia, proprio l'ultimo, corrispondente all'interruzione 128_{10}, ovvero 80_{16}, serve per le chiamate di sistema.

La struttura di ogni elemento della tabella IDT è un po' complicata, pertanto, per scriverci un nuovo valore si usa la funzione ***idt_descriptor()*** che si occupa di spezzettare e ricollocare i dati come richiesto dal microprocessore (listato 94.6.14)

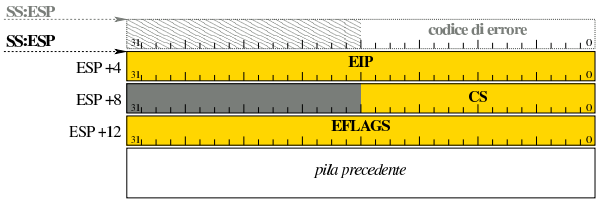
Tabella 84.35. Funzioni per la gestione della tabella IDT.

Funzione	Descrizione
`void idt_descriptor (int desc, void *isr, uint16_t selector, bool present, char type, char dpl);`	Scrive una voce della tabella IDT, sezionando e ricomponendo i dati come richiesto dal microprocessore. Listato 94.6.14.
`void idt (void);`	Predispone e attiva la tabella IDT; per questo si avvale in modo particolare delle funzioni ***idt_descriptor()*** e di ***idt_load()***. Listato 94.6.13.
`void idt_load (void *idtr);`	Fa sì che il microprocessore carichi la tabella IDT, a partire dal puntatore al registro IDTR; registro che contiene l'informazione della collocazione in memoria della tabella IDT e della sua estensione. Listato 94.6.16.
`void idt_irq_remap (unsigned int offset_1, unsigned int offset_2);`	Modifica la mappatura delle interruzioni hardware (IRQ) spostando il primo gruppo a partire dal valore di ***offset_1*** e il secondo gruppo a partire da ***offset_2***. Listato 94.6.15.
`void idt_print (void *idtr, unsigned int first unsigned int last);`	Funzione diagnostica, usata per visualizzare il contenuto della tabella IDT in binario. Listato 94.6.17.

84.3.5 Gestione delle interruzioni

Le interruzioni che individua il microprocessore (eccezioni, interruzioni software e interruzioni hardware) fanno interrompere l'attività normale dello stesso, costringendolo ad accumulare nella pila attuale dei dati lo stato di alcuni registri ed eventualmente di un codice di

errore, saltando poi alla posizione di codice indicata nella voce corrispondente nella tabella IDT. Va osservato che, per semplicità, os32 fa lavorare i propri processi nell'anello zero, come il kernel, per cui i dati accumulati nella pila si limitano a quelli della figura successiva, perché non c'è mai un passaggio da un livello di privilegio a un altro.



Le posizioni del codice a cui il microprocessore deve saltare, secondo le indicazioni della tabella IDT, sono contenute tutte nel file '`kernel/ibm_i386/isr.s`', mentre nel file '`kernel/ibm_i386.h`' vi si fa riferimento attraverso dei prototipi di funzione, benché non si tratti propriamente di funzioni.

Tabella 84.37. Funzioni per la gestione delle interruzioni.

Funzione	Descrizione
`void isr_`***n*** `(void);`	Si tratta di procedure attivate dalle interruzioni, dove per esempio ***isr_33()*** viene eseguita a seguito del verificarsi dell'interruzione numero 33, la quale ha origine da IRQ 1, ovvero dalla tastiera. L'indicazione di quale procedura attivare per ogni interruzione dipende dalla configurazione della tabella IDT. Listato 94.6.21.
`void isr_exception_unrecoverable (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` ***interrupt***`, uint32_t` ***error***`, uint32_t` ***eip***`, uint32_t` ***cs***`, uint32_t` ***eflags***`);`	Questa funzione viene usata all'interno del file '`isr.s`' per segnalare il verificarsi di un'eccezione non risolvibile, come nel caso di una divisione per zero. Pertanto, la funzione ha soprattutto un significato diagnostico. Listato 94.6.23.
`char *isr_exception_name (int` ***exception***`);`	Restituisce il puntatore alla stringa contenente il nome dell'eccezione corrispondente al numero di interruzione fornito. Viene usata da ***isr_exception_unrecoverable()*** per dare delle indicazioni comprensibili sull'eccezione che si è verificata. Listato 94.6.22.

Funzione	Descrizione
`void isr_irq_clear (uint32_t` ***idtn***`);`	Avvisa il PIC (*programmable interrupt controller*) che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Tuttavia, essendoci due PIC, la funzione stabilisce quale dei due è coinvolto direttamente e di conseguenza come procedere. Listato 94.6.24.
`void isr_irq_clear_pic1 (void);`	Avvisa il PIC1 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.25.
`void isr_irq_clear_pic2 (void);`	Avvisa il PIC2 che l'interruzione hardware emessa è stata recepita e se ne possono ricevere altre. Listato 94.6.26.

84.4 Gestione dei processi

Listato 94.6.21; listato 94.14 e successivi.

La gestione dei processi è raccolta nei file '`kernel/proc.h`' e '`kernel/proc/…`'; tuttavia, dal file '`kernel/ibm_i386/isr.s`' hanno origine le procedure attivate dalle interruzioni e dalle chiamate di sistema: le chiamate di sistema e le interruzioni provenienti dal temporizzatore interno provocano l'attivazione dello schedulatore.

Con os32, quando un processo viene interrotto per lo svolgimento del compito dell'interruzione, si passa sempre a utilizzare la pila dei dati del kernel. Per annotare la posizione in cui si trova l'indice della pila del kernel si usa la variabile ***_ksp***, accessibile anche dal codice in linguaggio C.

Il codice del kernel può essere interrotto dagli impulsi del temporizzatore, ma in tal caso non viene coinvolto lo schedulatore per lo scambio con un altro processo, così che dopo l'interruzione è sempre il kernel che continua a funzionare; pertanto, nella funzione ***kmain()*** è il kernel che cede volontariamente il controllo a un altro processo (ammesso che ci sia) con una chiamata di sistema nulla.

84.4.1 File «kernel/ibm_i386/isr.s»

Il file '`kernel/ibm_i386/isr.s`' contiene il codice per la gestione delle interruzioni dei processi. Nella parte iniziale del file, vengono dichiarate delle variabili, alcune delle quali sono pubbliche e accessibili anche dal codice in C.

```
.section .data
proc_syscallnr:              .int   0x00000000
proc_msg_offset:             .int   0x00000000
proc_msg_size:               .int   0x00000000
proc_instruction_pointer:    .int   0x00000000
proc_back_address:           .int   0x00000000
_ksp:                        .int   0x00000000
syscall_working:             .int   0x00000000
_clock_kernel:
kticks_lo:                   .int   0x00000000
kticks_hi:                   .int   0x00000000
_clock_time:
tticks_lo:                   .int   0x00000000
tticks_hi:                   .int   0x00000000
```

Si tratta di variabili scalari da 32 bit, tenendo conto che: i simboli '**`kticks_lo`**' e '**`kticks_hi`**' compongono assieme la variabile ***_clock_kernel*** a 64 bit per il linguaggio C; i simboli '**`tticks_lo`**' e '**`tticks_hi`**' compongono assieme la variabile ***_clock_time*** a 64 bit per il linguaggio C.

Dopo la dichiarazione delle variabili inizia il codice vero e proprio,

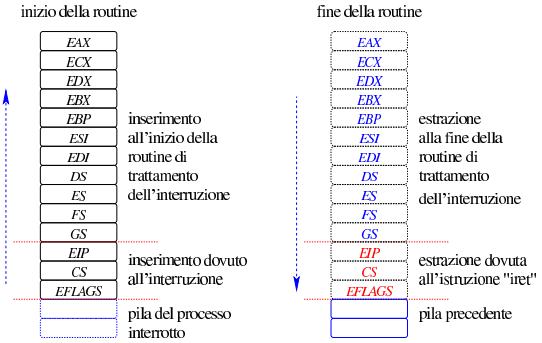
dove i simboli '**isr_*n***' si riferiscono al codice da usare in presenza dell'interruzione ***n***. Tra tutte, le interruzioni più importanti sono quelle del temporizzatore (***isr_32()***), il quale produce un impulso a circa 100 Hz; quelle della tastiera (***isr_33()***) e delle chiamate di sistema (***isr_128()***).

Il codice per la gestione dei tre tipi di interruzione più importanti ha delle similitudini che conviene analizzare simultaneamente. os32 non cambia mai anello, nel senso che il livello di privilegio dei processi è pari a quello del kernel; pertanto, nel momento in cui si verifica un'interruzione, la pila e il segmento dati in essere sono quelli del processo interrotto. Le procedure che gestiscono le tre interruzioni principali iniziano con il salvataggio dei registri nella pila attuale e il passaggio al segmento dei dati del kernel, lasciando temporaneamente la pila nel segmento dati del processo interrotto; nello stesso modo, terminano con il ripristino del segmento dati originario (al momento dell'interruzione) e il ripristino successivo dei registri, estraendone i valori dalla pila:

```
    #
    # Save into process stack:
    #
    pushl %gs
    pushl %fs
    pushl %es
    pushl %ds
    pushl %edi
    pushl %esi
    pushl %ebp
    pushl %ebx
    pushl %edx
    pushl %ecx
    pushl %eax
    #
    # Set the data segments to the kernel data segment,
    # so that the following variables can be accessed.
    #
    mov  $16, %ax # DS, ES, FS and GS.
    mov  %ax, %ds
    mov  %ax, %es
    mov  %ax, %fs
    mov  %ax, %gs
    …
    …
    #
    # Restore from process stack.
    #
    popl %eax
    popl %ecx
    popl %edx
    popl %ebx
    popl %ebp
    popl %esi
    popl %edi
    popl %ds
    popl %es
    popl %fs
    popl %gs
    …
    iret
```

Il segmento dati del kernel si trova nella terza voce della tabella GDT (la prima è nulla, la seconda è per il codice del kernel, la terza è per i dati del kernel). Sapendo che ogni voce occupa 8 byte (64 bit), per raggiungere l'inizio della terza voce occorre indicare il valore 16 nel registro di segmento.

Figura 84.40. Inserimento nella pila del processo interrotto.



Durante l'elaborazione di un'interruzione proveniente dal temporizzatore o dalla tastiera, è necessario sapere se è già in corso l'elaborazione di una chiamata di sistema. Se ciò accade, l'impulso del temporizzatore viene recepito, incrementando i contatori, ma non viene fatto altro, mentre l'impulso della tastiera viene semplicemente ignorato.

```
    #
    # Check if a system call is already working: if so,
    # just leave (go to L2).
    #
    cmpl $1, syscall_working
    je L2
```

In pratica, quando si presenta una chiamata di sistema, inizialmente viene assegnato il valore uno alla variabile ***syscall_working***, mentre alla fine del suo compito questa variabile viene azzerata:

```
    #
    # Tell that it is a system call.
    #
    movl $1, syscall_working
    …
    #
    # End of system call.
    #
    movl $0, syscall_working
```

Quando l'interruzione proviene dal temporizzatore e non è in corso l'esecuzione di una chiamata di sistema, oppure quando l'interruzione deriva proprio da una chiamata di sistema, viene attivato lo schedulatore (direttamente o indirettamente, attraverso la funzione che svolge il lavoro richiesto dalla chiamata di sistema), ma per fare questo, è necessario passare alla pila dei dati del kernel, per poi ripristinarla successivamente:

```
    #
    # Save process stack registers into kernel data segment.
    #
    mov %ss,  proc_stack_segment_selector
    mov %esp, proc_stack_pointer
    …
    #
    # Switch to kernel stack.
    #
    mov $16, %ax
    mov %ax, %ss
    mov _ksp, %esp
    …
    …
    #
    # Restore process stack registers from kernel data
    # segment.
    #
    mov proc_stack_segment_selector, %ss
    mov proc_stack_pointer, %esp
```

Figura 84.44. Scambi delle pile: prima fase.

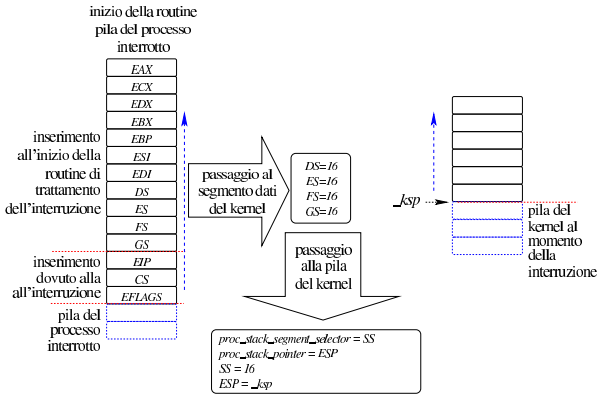
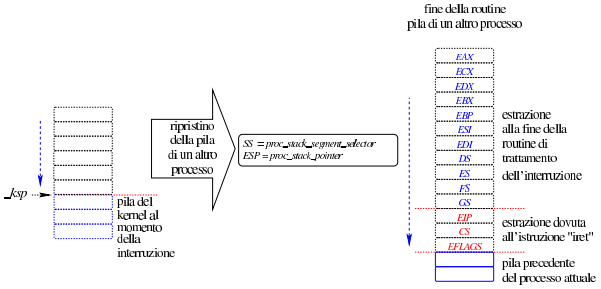


Figura 84.45. Scambi delle pile: seconda fase.



84.4.1.1 Particolarità della routine «isr_32», ovvero «irq_timer»

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, la routine di gestione delle interruzioni del temporizzatore si occupa di incrementare i contatori degli impulsi. Gli impulsi giungono alla frequenza di 100 Hz circa, per cui non c'è la necessità di fare alcun tipo di conversione:

```
…
isr_32:         # IRQ 0: «timer»
    cli
    jmp irq_timer
…
irq_timer:
    …
    add $1, kticks_lo    # Kernel ticks counter.
    adc $0, kticks_hi    #
    #
    add $1, tticks_lo    # Clock ticks counter.
    adc $0, tticks_hi    #
    …
```

A questo punto, se l'interruzione è avvenuta mentre era in corso l'elaborazione di una chiamata di sistema, tutto si conclude con il ripristino dei registri e del PIC1, in modo da consentire la ripresa delle interruzioni. Se invece l'interruzione è avvenuta in una situazione differente, si verifica ancora che non sia stato interrotto il funzionamento del kernel stesso, perché se così fosse, anche in questo caso la procedura termina con il solito ripristino dei registri e del PIC1.

```
    #
    # Check if it is already in kernel mode: the kernel has
    # PID 0. If so, just leave (go to L2).
    #
    mov proc_current, %edx       # Interrupted PID.
    mov $0,           %eax       # Kernel PID.
    cmp %eax, %edx
    je L2
```

Se non è stato interrotto il codice del kernel, viene chiamata la funzione ***proc_scheduler()***, la quale può cambiare i valori delle variabili pubbliche ***proc_stack_segment_selector*** e ***proc_stack_pointer***, provocando così la sostituzione del processo interrotto, quando subito dopo si ripristina la pila a cui queste due variabili fanno riferimento.

84.4.1.2 Particolarità della routine «isr_128»

La pila dei dati al momento dell'interruzione dovuta a una chiamata di sistema, contiene anche le informazioni necessarie a conoscere il tipo di funzione richiesta e gli argomenti di questa, in forma di variabile strutturata, di cui viene trasmesso il puntatore.

Dopo il salvataggio dei registri principali e dopo il cambiamento del segmento dati, rimanendo ancora sulla pila dei dati del processo interrotto, si recuperano dalla pila le informazioni necessarie a ricostruire la funzione richiesta, salvandole in variabili locali:

```
    #
    # Save some more data, from the system call.
    #
    .equ SYSCALL_NUMBER,      60
    .equ MESSAGE_OFFSET,      64
    .equ MESSAGE_SIZE,        68
    #
    mov %esp, %ebp
    mov SYSCALL_NUMBER(%ebp), %eax
    mov %eax, proc_syscallnr
    mov MESSAGE_OFFSET(%ebp), %eax
    mov %eax, proc_msg_offset
    mov MESSAGE_SIZE(%ebp), %eax
    mov %eax, proc_msg_size
```

A questo punto, in modo simile a quanto avviene per le interruzioni del temporizzatore, si verifica se la chiamata di sistema è avvenuta durante il funzionamento del kernel, cosa che os32 consente. Tuttavia, la chiamata di sistema viene eseguita ugualmente, solo che si salva l'indice della pila nella variabile ***_ksp***; pertanto, è proprio attraverso una prima chiamata di sistema nulla che os32 inizializza la gestione delle interruzioni.

```
    #
    # Check if it is already in kernel mode: the kernel has
    # PID 0.
    #
    mov proc_current, %edx       # Interrupted PID.
    mov $0,           %eax       # Kernel PID.
    cmp %eax, %edx
    jne L3
    #
    mov %esp, _ksp
L3:
```

A questo punto viene eseguito il passaggio alla pila del kernel, indipendentemente dal fatto che serva o meno, quindi viene chiamata la funzione ***sysroutine()***, inserendo nella pila attuale i parametri richiesti e salvati precedentemente all'interno di variabili locali:

```
    push proc_msg_size
    push proc_msg_offset
    push proc_syscallnr
    call sysroutine
    add $4, %esp
    add $4, %esp
    add $4, %esp
```

I passi successivi includono il ripristino della pila precedente, secondo quanto annotato nelle variabili globali ***proc_stack_segment_selector*** e ***proc_stack_pointer***, e lo stato dei registri dalla nuova pila.

Va osservato che la funzione ***sysroutine()*** oltre che prendersi carico di eseguire il compito della chiamata di sistema richiesta, provvede poi a sostituire il processo interrotto, avvalendosi a sua volta della funzione ***proc_scheduler()***.

84.4.2 La tabella dei processi

Listato 94.14.

Nel file '`kernel/proc.h`' viene definito il tipo '**`proc_t`**', con il quale, nel file '`kernel/proc/proc_public.c`' si definisce la tabella dei processi, rappresentata dall'array ***proc_table[]***.

Listato 84.51. Struttura del tipo '**proc_t**', corrispondente agli elementi dell'array ***proc_table[]***.

```
typedef struct {
    pid_t             ppid;          // Parent PID.
    pid_t             pgrp;          // Process group ID.
    uid_t             uid;           // Real user ID
    uid_t             euid;          // Effective user ID.
    uid_t             suid;          // Saved user ID.
    gid_t             gid;           // Real group ID
    gid_t             egid;          // Effective group ID.
    gid_t             sgid;          // Saved group ID.
    dev_t             device_tty;    // Controlling terminal.
    char              path_cwd[PATH_MAX];
                                     // Working directory path.
    inode_t          *inode_cwd;     // Working directory inode.
    int               umask;         // File creation mask.
    unsigned long int sig_status;    // Active signals.
    unsigned long int sig_ignore;    // Signals to be ignored.
    uintptr_t         sig_handler[MAX_SIGNALS];
                                     // Opt. sig. handlers.
    uintptr_t         sig_handler_wrapper;
                                     // Special wrapper.
    clock_t           usage;         // Clock ticks CPU
                                     // time usage.
    unsigned int      status;
    int               wakeup_events; // Wake up for something.
    int               wakeup_signal; // Signal waited.
    unsigned int      wakeup_timer;  // Seconds to wait for.
    inode_t          *wakeup_inode;  // Inode waited.
    addr_t            address_text;
    size_t            domain_text;
    addr_t            address_data;
    size_t            domain_data;
    size_t            domain_stack;  // Included inside the
                                     // data.
    size_t            extra_data;    // Extra data for 'brk()'.
    uint32_t          sp;
    int               ret;
    char              name[PATH_MAX];
    fd_t              fd[FOPEN_MAX];
} proc_t;
```

La tabella successiva descrive il significato dei vari membri previsti dal tipo '**proc_t**'. Va osservato che la cosiddetta «u-area» (*user area*) non viene gestita come un sistema Unix tradizionale e tutti i dati dei processi sono raccolti nella tabella gestita dal kernel. Di conseguenza, dal momento che i processi non dispongono di una tabella personale con i dati della u-area, devono avvalersi sempre di chiamate di sistema per leggere i dati del proprio processo.

Tabella 84.52. Membri del tipo '**proc_t**'.

Membro	Contenuto
`ppid`	Numero del processo genitore: *parent process id*.
`pgrp`	Numero del gruppo di processi a cui appartiene quello della voce corrispondente: *process group*. Si tratta del numero del processo a partire dal quale viene definito il gruppo.
`uid`	Identità reale del processo della voce corrispondente: *user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro '**euid**'.
`euid`	Identità efficace del processo della voce corrispondente: *effective user id*. Si tratta del numero dell'utente, secondo la classificazione del file '`/etc/passwd`', per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quell'utente.
`suid`	Identità salvata: *saved user id*. Si tratta del valore che aveva ***euid*** prima di cambiare identità.

Membro	Contenuto
`gid`	Gruppo reale del processo della voce corrispondente: *group id*. Si tratta del numero del gruppo, secondo la classificazione del file '`/etc/group`', per conto del quale il processo è stato avviato. Tuttavia, i privilegi del processo dipendono dall'identità efficace, definita dal membro '**egid**'.
`egid`	Gruppo efficace del processo della voce corrispondente: *effective group id*. Si tratta del numero del gruppo, secondo la classificazione del file '`/etc/group`', per conto del quale il processo è in funzione; pertanto, il processo ha i privilegi di quel gruppo.
`sgid`	Gruppo salvato: *saved group id*. Si tratta del valore che aveva ***egid*** prima di cambiare identità.
`device_tty`	Terminale di controllo, espresso attraverso il numero del dispositivo.
`path_cwd` `inode_cwd`	Entrambi i membri rappresentano la directory corrente del processo: nel primo caso in forma di percorso, ovvero di stringa, nel secondo in forma di puntatore a inode rappresentato in memoria.
`umask`	Maschera dei permessi associata al processo: i permessi attivi nella maschera vengono tolti in fase di creazione di un file o di una directory.
`sig_status`	Segnali inviati al processo e non ancora trattati: ogni segnale si associa a un bit differente del valore del membro ***sig_status***; un bit a uno indica che il segnale corrispondente è stato ricevuto e non ancora trattato.
`sig_ignore`	Segnali che il processo ignora: ogni segnale da ignorare si associa a un bit differente del valore del membro ***sig_ignore***; un bit a uno indica che quel segnale va ignorato.
`sig_handler[]`	Array di funzioni da eseguire al ricevimento del segnale rispettivo.
`sig_handler_wrapper[]`	Array di funzioni da usare per avvolgere quelle da eseguire al ricevimento di un certo segnale. Si tratta in pratica della funzione dichiarata nel file '`lib/signal/_signal_handler_wrapper.s`', ma è riferita al codice dell'applicazione di origine. Queste funzioni hanno il compito di sistemare la pila dopo l'esecuzione della funzione attivata da un segnale.
`usage`	Tempo di utilizzo della CPU, da parte del processo, espresso in impulsi del temporizzatore, il quale li produce alla frequenza di circa 100 Hz.
`status`	Stato del processo, rappresentabile attraverso una macro-variabile simbolica, definita nel file '`proc.h`'. Per os32, gli stati possibili sono: «inesistente», quando si tratta di una voce libera della tabella dei processi; «creato», quando un processo è appena stato creato; «pronto», quando un processo è pronto per essere eseguito, «in esecuzione», quando il processo è in funzione; «sleeping», quando un processo è in attesa di qualche evento; «zombie», quando un processo si è concluso, ha liberato la memoria, ma rimangono le sue tracce perché il genitore non ha ancora recepito la sua fine.
`wakeup_events`	Eventi attesi per il risveglio del processo, ammesso che si trovi nello stato si attesa. Ogni tipo di evento che può essere atteso corrisponde a un bit e si rappresenta con una macro-variabile simbolica, dichiarata nel file '`lib/sys/os32.h`'.

Membro	Contenuto
`wakeup_signal`	Ammesso che il processo sia in attesa di un segnale, questo membro esprime il numero del segnale atteso.
`wakeup_timer`	Ammesso che il processo sia in attesa dello scadere di un conto alla rovescia, questo membro esprime il numero di secondi che devono ancora trascorrere.
`wakeup_inode`	Ammesso che il processo sia in attesa di poter accedere a un inode, questo membro esprime il puntatore a un inode che deve rendersi disponibile.
`address_text` `domain_text`	Il valore di questi membri descrive la memoria utilizzata dal processo per le istruzioni (il segmento codice). La voce ***domain_text*** rappresenta la dimensione occupata a partire da ***address_text***.
`address_data` `domain_data`	Il valore di questi membri descrive la memoria utilizzata dal processo per i dati; tuttavia l'informazione è utile solo se i dati sono distinti dal segmento codice (gli eseguibili di os32 possono essere compilati in modo da condividere codice e dati nello stesso segmento, oppure in modo da tenerli separati).
`domain_stack`	Dimensione della memoria usata per la pila dei dati, la quale, a seconda del tipo di eseguibile, può collocarsi nel segmento dati, oppure nell'unico segmento che include codice e dati; in ogni caso, si tratta della porzione finale della memoria in questione.
`extra_data`	Dimensione della memoria usata per l'allocazione dinamica della memoria. Questo spazio, ammesso che sia utilizzato, si colloca dopo la pila e può essere modificato con la funzione ***brk()***.
`sp`	Indice della pila dei dati, nell'ambito del segmento dati del processo. Il valore è significativo quando il processo è nello stato di pronto o di attesa di un evento. Quando invece un processo era attivo e viene interrotto, questo valore viene aggiornato.
`ret`	Rappresenta il valore restituito da un processo terminato e passato nello stato di «zombie».
`name[]`	Il nome del processo, rappresentato dal nome del programma avviato.
`fd[]`	Tabella dei descrittori dei file relativi al processo.

L'indice della tabella dei processi corrisponde al numero del processo, ovvero il PID, che infatti non è rappresentato al suo interno. Tuttavia, per accedervi più agevolmente, viene usata la funzione ***proc_reference()***, la quale, fornendo il numero PID desiderato, fornisce il puntatore alla voce della tabella che lo descrive(listato 94.14.6).

84.4.3 Chiamate di sistema

I processi eseguono una chiamata di sistema attraverso la funzione ***sys()***, dichiarata nel file '`lib/sys/os32/sys.s`'(listato 95.21.7). La funzione in sé, per come è dichiarata, potrebbe avere qualunque parametro, ma in pratica ci si attende che il suo prototipo sia il seguente:

```
void sys (int syscallnr, void *message, size_t size);
```

Il numero della chiamata di sistema, richiesto come primo parametro, si rappresenta attraverso una macro-variabile simbolica, definita nel file '`lib/sys/os32.h`'.

Per fornire dei dati a quella parte di codice che deve svolgere il compito richiesto, si usa una variabile strutturata, di cui viene trasmesso il puntatore (riferito al segmento dati del processo che esegue la chiamata) e la dimensione complessiva.

Nel file '`lib/sys/os32.h`' sono definiti dei tipi derivati, riferiti a variabili strutturate, per ogni tipo di chiamata(listato 95.21). Per esempio, per la chiamata di sistema usata per cambiare la directory corrente del processo, si usa un messaggio di tipo '**`sysmsg_chdir_t`**':

```
typedef struct {
    char         path[PATH_MAX];
    int          ret;
    int          errno;
    int          errln;
    char         errfn[PATH_MAX];
} sysmsg_chdir_t;
```

In realtà, la funzione ***sys()***, si limita a produrre un'interruzione software, da cui viene attivata la routine che inizia al simbolo '**`isr_128`**' nel file '`kernel/ibm_i386/isr.s`', la quale estrapola le informazioni salienti dalla pila dei dati e poi le fornisce alla funzione ***sysroutine()***:

```
void sysroutine (uint32_t syscallnr,
                 uint32_t msg_off, uint32_t msg_size);
```

I parametri della funzione ***sysroutine()*** corrispondono in pratica agli argomenti della chiamata della funzione ***sys()***, con la differenza che nei vari passaggi hanno perso l'identità originaria e giungono come numeri puri e semplici. A questo proposito, il secondo parametro cambia nome, in quanto ciò che prima era il puntatore a un'area di memoria, qui va interpretato come lo scostamento rispetto al segmento dati del processo (*segment offset*).

84.4.4 Funzione «proc_init()»

Listato 94.14.3.

```
void proc_init (void);
```

La funzione ***proc_init()*** viene chiamata dalla funzione ***kmain()***, una volta sola, per attivare la gestione dei processi elaborativi. Si occupa di compiere le azioni seguenti:

- predisporre la tabella GDT, attraverso la chiamata della funzione ***gdt()***;
- impostare il temporizzatore in modo da fornire impulsi alla frequenza dichiarata nella macro-variabile ***CLOCKS_PER_SEC***, pari a 100 Hz;
- predisporre la tabella IDT, attraverso la chiamata della funzione ***idt()***;
- azzerare la tabella dei processi, inserendovi però i dati relativi al kernel (il processo zero);
- allocare la memoria già utilizzata dal kernel;
- attivare selettivamente le interruzioni hardware desiderate;
- attivare la gestione delle unità PATA;
- innestare il file system principale.

84.4.5 Funzione «sysroutine()»

Listato 94.14.28.

La funzione ***sysroutine()*** viene chiamata esclusivamente dalla routine attivata dalle chiamate di sistema (tale routine è introdotta dal simbolo '**`isr_128`**' nel file '`kernel/ibm_i386/isr.s`') e ha i parametri che si possono vedere dal prototipo:

```
void sysroutine (uint32_t syscallnr,
                 uint32_t msg_off, uint32_t msg_size);
```

Il primo parametro è il numero della chiamata di sistema che ha provocato l'interruzione; gli altri due danno la posizione e la dimensione del messaggio inviato attraverso la chiamata di sistema.

All'inizio della funzione viene dichiarato un puntatore a un'unione di tutti i tipi di messaggio gestibili:

```
    union {
        sysmsg_brk_t       brk;
        sysmsg_chdir_t     chdir;
        sysmsg_chmod_t     chmod;
        …
    } *msg;
```

Viene quindi calcolata la collocazione del messaggio originale, per poi poter assegnare a ***msg*** il puntatore a tale messaggio.

Le chiamate di sistema sono fatte per le applicazioni, ma al kernel è consentito di eseguirne alcune, per motivi particolari. Se però il kernel tenta di eseguire una chiamata differente, si ottiene un messaggio di avvertimento, ma si tenta ugualmente l'esecuzione della richiesta.

Disponendo del puntatore ***msg***, sapendo di quale chiamata di sistema si tratta, il messaggio può essere letto come:

```
msg->tipo_chiamata
```

Per esempio, per la chiamata di sistema '**SYS_CHDIR**', si deve fare riferimento al messaggio ***msg->chdir***; pertanto, per raggiungere il membro ***ret*** del messaggio si usa la notazione ***msg->chdir.ret***.

Per distinguere il tipo di chiamata si usa una struttura di selezione:

```
    switch (syscallnr)
      {
        case SYS_0:
            break;
        case SYS_BRK:
            msg->brk.ret = s_brk (pid, msg->brk.address);
            sysroutine_error_back (&msg->brk.errno,
                                   &msg->brk.errln,
                                   msg->brk.errfn);
            break;
        case SYS_CHDIR:
            msg->chdir.ret = s_chdir (pid, msg->chdir.path);
            sysroutine_error_back (&msg->chdir.errno,
                                   &msg->chdir.errln,
                                   msg->chdir.errfn);
            break;
```

Il messaggio usato per trasmettere i dati della chiamata, può servire anche per restituire dei dati al mittente, pertanto, spesso alcuni contenuti vengono modificati. Ciò succede particolarmente con il membro ***ret*** che generalmente rappresenta il valore restituito dalla chiamata di sistema.

Al termine del lavoro, viene chiamata la funzione ***proc_scheduler()***.

84.4.6 Funzione «proc_scheduler()»

Listato 94.14.11.

La funzione ***proc_scheduler()*** non prevede parametri e riceve le informazioni che le possono servire attraverso variabili pubbliche: ***_ksp***, ***proc_stack_pointer***, ***proc_stack_segment_selector*** e ***proc_current***. A sua volta, la funzione aggiorna i valori di queste variabili, per mettere in pratica uno scambio di processi.

```
void proc_scheduler (void);
```

La prima cosa che fa la funzione consiste nel verificare che il valore dell'indice della pila del processo interrotto non superi lo spazio disponibile per la pila stessa. Diversamente il processo viene eliminato forzatamente, con una segnalazione adeguata sul terminale attivo. Si ottiene comunque una segnalazione se l'indice si avvicina pericolosamente al limite.

Successivamente la funzione svolge delle operazioni che riguardano tutti i processi: aggiorna i contatori dei processi che attendono lo scadere di un certo tempo; verifica la presenza di segnali e predispone le azioni relative; raccoglie l'input dai terminali.


```
    proc_sch_timers ();
    …
    proc_sch_signals ();
    …
    proc_sch_terminals ();
```

A quel punto aggiorna il tempo di utilizzo della CPU del processo appena interrotto:

```
    current_clock   = s_clock ((pid_t) 0);
    proc_table[prev].usage += current_clock - previous_clock;
    previous_clock  = current_clock;
```

Quindi inizia la ricerca di un altro processo, candidato a essere ripreso al posto di quello interrotto. La ricerca inizia dal processo successivo a quello interrotto, senza considerare alcun criterio di precedenza. Il ciclo termina se la ricerca incontra di nuovo il processo di partenza.

```
    for (next = prev+1; next != prev; next++)
      {
        if (next >= PROCESS_MAX)
          {
            next = -1; // At the next loop, 'next'
                       // will be zero.
            continue;
          }
        …
      }
```

All'interno di questo ciclo di ricerca, se si incontra un processo pronto per essere messo in funzione, lo si scambia con quello interrotto: in pratica si salva il valore attuale dell'indice della pila, si scambiano gli stati e si aggiornano i valori di ***proc_current***, ***proc_stack_segment_selector*** e ***proc_stack_pointer***, in modo da ottenere effettivamente lo scambio all'uscita dalla funzione:

```
        else if (proc_table[next].status == PROC_READY)
          {
            if (proc_table[prev].status == PROC_RUNNING)
              {
                proc_table[prev].status = PROC_READY;
              }
            proc_table[prev].sp     = proc_stack_pointer;
            proc_table[next].status = PROC_RUNNING;
            proc_table[next].ret    = 0;
            proc_current            = next;
            proc_stack_segment_selector
                = gdt_pid_to_segment_data (next) * 8;
            proc_stack_pointer      = proc_table[next].sp;
            break;
          }
```

Alla fine del ciclo, occorre verificare se esiste effettivamente un processo successivo attivato, perché in caso contrario, si lascia il controllo direttamente al kernel. In fine, si salva il valore accumulato in precedenza dell'indice della pila del kernel, nella variabile ***_ksp***.

84.4.7 Programmazione dei segnali

Un processo può ricevere un segnale, a seguito del quale può essere interrotto per compiere una certa azione. La maggior parte dei segnali può essere inibita, in modo tale che ricevendoli il processo non venga a essere disturbato, oppure si può associare loro una funzione, da eseguire al momento del ricevimento del tale segnale. Diversamente, in mancanza di tale associazione, il ricevimento di un segnale comporta un'azione predefinita.

L'associazione di una funzione allo scattare di un segnale si ottiene, nel codice dell'applicazione, con la funzione ***signal()*** (listato 95.17.3), la quale attraverso una chiamata di sistema fornisce al kernel tutti i dati necessari per la programmazione del segnale.

La vera difficoltà sta nell'esecuzione effettiva della funzione, nel momento in cui scatta il segnale previsto per il processo.

```
sighandler_t signal (int sig, sighandler_t handler);
```

La funzione ***signal()*** richiede l'indicazione del numero del segnale da programmare e di un puntatore rappresentato da una funzione che si vuole azionare nel momento in cui scatta il segnale in questione. Il tipo '`sighandler_t`' rappresenta il puntatore a una funzione che richiede un parametro di tipo intero, costituito dal numero del segnale ricevuto, e non restituisce alcunché; pertanto, la funzione che si passa come secondo parametro della funzione ***signal()*** deve avere la forma seguente:

```
void handler (int sig);
```

La funzione ***signal()***, a sua volta, esegue finalmente la chiamata di sistema, ma oltre al numero del segnale e al puntatore della funzione da azionare, invia il puntatore di un'altra funzione, denominata ***_sighandler_wrapper()***, il cui scopo è quello di avvolgere la chiamata della funzione da azionare, per sistemare in modo appropriato la pila dei dati (listato 95.17.1). In questa fase della descrizione del problema, va osservato che la funzione ***_sighandler_wrapper()*** si trova nel codice del processo che riceve il segnale.

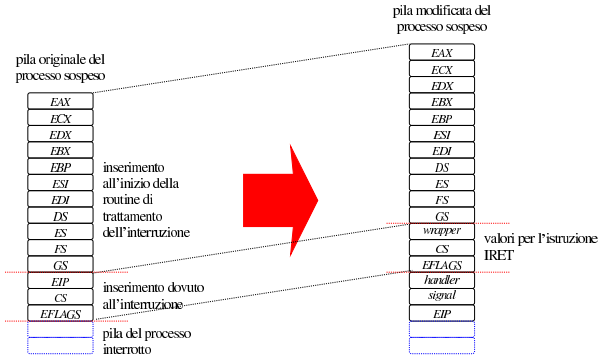
La chiamata di sistema, quando raggiunge il kernel, comporta l'aggiornamento dei dati del processo, annotando sia la funzione da azionare, sia la funzione che deve avvolgerla.

Quando arriva un segnale a un processo che prevede l'azionamento di una funzione, attraverso la funzione ***proc_sch_signals()***, chiamata a sua volta dalla funzione ***proc_scheduler()***, attraverso altri passaggi si arriva alla funzione ***proc_sig_handler()*** (listato 94.14.15).

```
void proc_sig_handler (pid_t pid, int sig);
```

La funzione ***proc_sig_handler()*** ha lo scopo di modificare la pila dei dati del processo ***pid***, in modo da far sì che, nel momento in cui fosse selezionato, prima di riprendere con l'attività sospesa originariamente, esegua la funzione attivata dal segnale ***sig***.

Figura 84.60. Modifica della pila attraverso la funzione ***proc_sig_handler()***.



Come si può vedere nella figura, i valori che servono all'istruzione IRET per concludere l'interruzione, vengono modificati in modo da ripartire iniziando con la funzione che avvolge quella da azionare, ***wrapper***, ovvero quella che dal lato dell'applicazione è chiamata ***_sighandler_wrapper()***. D'altro canto, quando quella funzione viene messa in azione, si trova nella pila dei valori che le servono per poter chiamare a sua volta la funzione da azionare effettivamente.

Il codice di ***_sighandler_wrapper()*** non corrisponde propriamente a una funzione, in quanto ciò che si trova nella pila non è quello che si prevede di solito. Le figure successive mostrano i cambiamenti della pila del processo, prima e dopo l'esecuzione della funzione incaricata di gestire il segnale ricevuto.

Figura 84.61. Dall'avvio di ***_sighandler_wrapper()*** fino alla chiamata della funzione di gestione del segnale ricevuto.

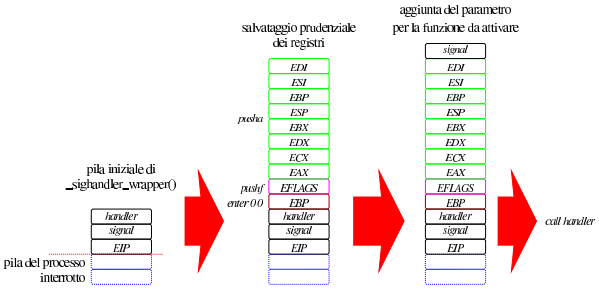
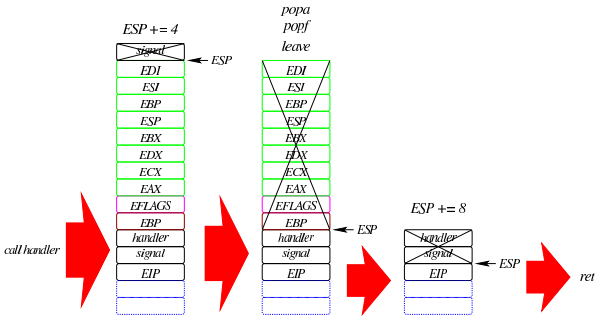


Figura 84.62. Dopo la conclusione della funzione di gestione del segnale ricevuto, fino alla restituzione del controllo al termine di ***_sighandler_wrapper()***.



Lo scopo della funzione ***_sighandler_wrapper()*** è quello di garantire che sia preservato completamente l'ambiente di lavoro del processo nel momento dell'interruzione, perché non è possibile fare affidamento sul rispetto delle convenzioni di chiamata, dato che la funzione da azionare in corrispondenza dell'interruzione, viene iniettata in una posizione arbitraria del codice.

Riquadro 84.63. Programmazione ripetuta dei segnali.

Il sistema tradizionale con cui si programma una funzione in corrispondenza di un segnale, richiede che lo scattare del segnale riporti la gestione di questo allo stato predefinito, perché altrimenti la stessa funzione azionata potrebbe essere interrotta da un segnale che ne aziona un'altra. Se così fosse, la pila dei dati potrebbe riempirsi velocemente, portando il processo al collasso. Di conseguenza, se il programmatore desidera ripristinare una funzione associata a un segnale, lo deve richiedere alla fine della funzione stessa (chiamando ***signal()*** di nuovo), ma in tal caso c'è la possibilità che quel segnale raggiunga il processo nell'intervallo di tempo tra l'azionamento della funzione e il ripristino della programmazione della stessa. Per esempio, ciò significa che se si vuole controllare il segnale ***SIG_TERM*** per impedire che questo porti alla conclusione il processo, anche se la funzione azionata dal segale richiama ***signal()*** per programmarsi nuovamente dopo la chiamata, in modo da non perdere il controllo del segnale, se il processo viene interessato da una raffica si segnali ***SIG_TERM***, prima o poi il processo viene concluso, perché un segnale lo raggiunge quando quella funzione non ha ancora fatto in tempo a chiamare ***signal()***.

84.4.8 Salvataggio e recupero della pila per i «salti non locali»

Il linguaggi C prevede la disponibilità di due funzioni, attraverso le quali è possibile salvare il contesto della pila dei dati per poterne recuperare lo stato in un momento successivo. Tuttavia, tale recupero può avvenire solo se dopo il salvataggio il contenuto della pila precedente rimane valido, in quanto il recupero avviene solo in una situazione in cui la pila sia stata incrementata ulteriormente.

Il salvataggio si ottiene con la funzione ***setjmp()*** e il recupero con ***longjmp()***. L'effetto della chiamata della funzione ***longjmp()*** comporta il riportare il processo alla situazione in cui si trovava dopo l'esecuzione della funzione ***setjmp()***, con la differenza che nel secondo caso, la funzione ***setjmp()*** restituisce un valore differente.

Si tratta di un modo pessimo di programmare, tuttavia fa parte dello standard del linguaggio C.

Generalmente, la realizzazione delle funzioni ***setjmp()*** e ***longjmp()*** avviene nella libreria, senza coinvolgere il kernel in alcun modo. Ma os32 procede diversamente e si avvale invece di chiamate di sistema. Si tratta comunque di una scelta motivata esclusivamente da una più semplice comprensione del codice, facendo rientrare il meccanismo in quello più generale della gestione dei processi.

La funzione ***setjmp()*** è realizzata dal file '`lib/setjmp/setjmp.s`' (listato 95.16.2). La funzione svolge sostanzialmente il compito che si può vedere tradotto in linguaggio C nel codice seguente, se il compilatore gestisse la pila dei dati nella forma più compatta e prevedibile:

```
#include <sys/os32.h>
#include <setjmp.h>
int
setjmp (jmp_buf env)
{
  sysmsg_jmp_t msg;
  msg.env = env;
  msg.ret = 0;
  sys (SYS_SETJMP, &msg, sizeof msg);
  return (msg.ret);
}
```

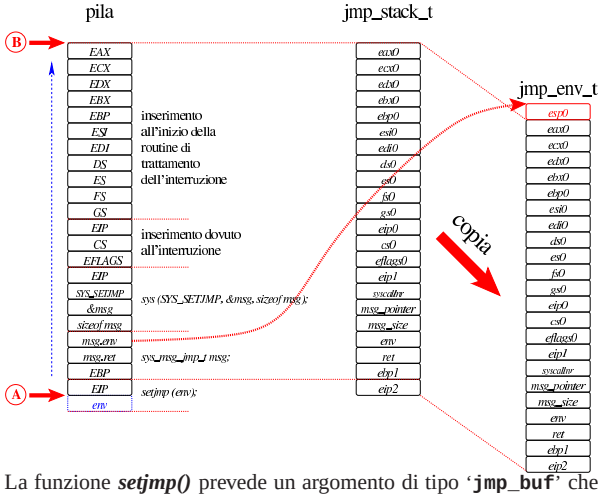
La funzione ***longjmp()*** è realizzata invece in C, nel file '`lib/setjmp/longjmp.c`' (listato 95.16.1), perché non c'è la necessità di conoscere esattamente la struttura della sua pila.

La struttura corrispondente al tipo '**`sysmsg_jmp_t`**' si limita a due campi: un puntatore che deve fare riferimento alla memoria in cui viene salvato il contenuto della pila e il valore che deve restituire ***setjmp()*** quando rivive attraverso la chiamata di ***longjmp()***.

```
typedef struct {
    void  *env;
    int    ret;
} sysmsg_jmp_t;
```

Le due chiamate di sistema raggiungono, rispettivamente, le funzioni ***s_setjmp()*** e ***s_longjmp()*** del kernel (listati 94.8.38 e 94.8.22). La funzione ***s_setjmp()*** salva lo stato della pila, a partire dalla chiamata della funzione ***setjmp()***, mentre ***s_longjmp()*** lo ripristina, rimettendo anche l'indice della pila allo stato che aveva al momento della chiamata di ***setjmp()***.

Figura 84.66. Lo stato della pila durante le varie fasi che riguardano la chiamata di ***setjmp()***, a confronto con i tipi '**`jmp_stack_t`**' e '**`jmp_env_t`**'.



La funzione ***setjmp()*** prevede un argomento di tipo '**`jmp_buf`**' che lo standard prescrive sia come un array:

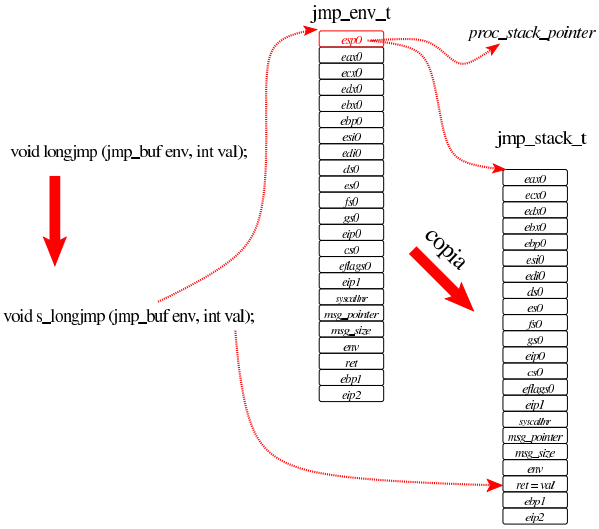
```
int setjmp (jmp_buf env);
```

In pratica, l'array serve solo a occupare lo spazio necessario a rappresentare il tipo '**`jmp_env_t`**', i cui membri si vedono rappresentati nella figura già apparsa. La funzione ***s_setjmp()*** si occupa si salvare lo stato della pila, dal punto «A» al punto «B» della figura, all'interno di ***env***, secondo la struttura di '**`jmp_env_t`**', mettendo, oltre al contenuto della pila, il valore del suo indice attuale.

La funzione ***longjmp()*** deve portare al ripristino della pila, in una posizione antecedente rispetto a quella attuale.

```
void longjmp (jmp_buf env, int val);
```

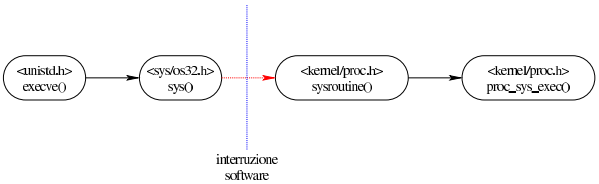
Figura 84.67. La chiamata di ***longjmp()*** ricostruisce la vecchia pila di ***setjmp()***, nella posizione in cui si trovava, ricollocando l'indice della pila e modificando il valore che poi ***setjmp()*** rediviva va a restituire.



84.5 Caricamento ed esecuzione delle applicazioni

Caricare un programma e metterlo in esecuzione è un processo delicato che parte dalla funzione ***execve()*** della libreria standard e viene svolto dalla funzione ***proc_sys_exec()*** del kernel.

Figura 84.68. Da ***execve()*** a ***proc_sys_exec()***.



84.5.1 Caricamento in memoria

La funzione ***proc_sys_exec()*** (listato 94.14.22) del kernel è quella che svolge il compito di caricare un processo in memoria e di annotarlo nella tabella dei processi.

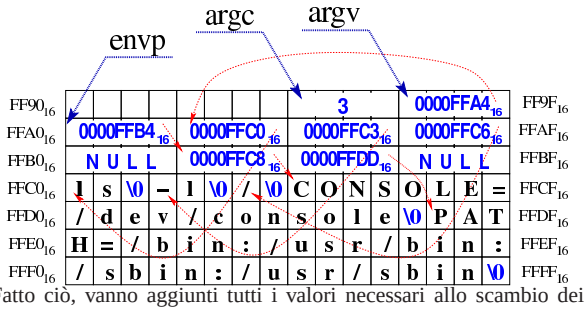
La funzione, dopo aver verificato che si tratti di un file eseguibile valido e che ci siano i permessi per metterlo in funzione, procede all'allocazione della memoria, dividendo se necessario l'area codice da quella dei dati, quindi legge il file e copia opportunamente le componenti di questo nelle aree di memoria allocate.

La realizzazione attuale della funzione ***proc_sys_exec()*** non è in grado di verificare se un processo uguale sia già in memoria, quindi carica la parte del codice anche se questa potrebbe essere già disponibile.

Terminato il caricamento del file viene aggiornata la tabella GDT e quindi viene ricostruita in memoria la pila dei dati del processo.

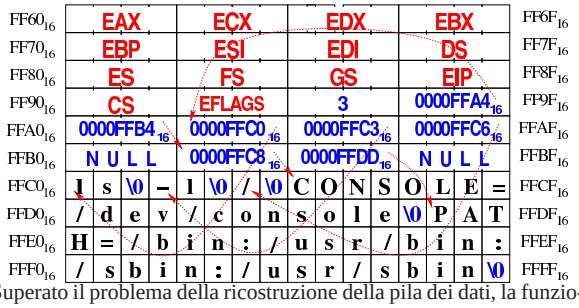
Prima si mettono sul fondo le stringhe delle variabili di ambiente e quelle degli argomenti della chiamata, quindi si aggiungono i puntatori alle stringhe delle variabili di ambiente, ricostruendo così l'array noto convenzionalmente come '**envp[]**', continuando con l'aggiunta dei puntatori alle stringhe degli argomenti della chiamata, per riprodurre l'array '**argv[]**'. Per ricostruire gli argomenti della chiamata della funzione ***main()*** dell'applicazione, vanno però aggiunti ancora: il puntatore all'inizio dell'array delle stringhe che descrivono le variabili di ambiente, il puntatore all'array delle stringhe che descrivono gli argomenti della chiamata e il valore che rappresenta la quantità di argomenti della chiamata.

Figura 84.69. Caricamento degli argomenti della chiamata della funzione ***main()***.



Fatto ciò, vanno aggiunti tutti i valori necessari allo scambio dei processi, costituiti dai vari registri da rimpiazzare.

Figura 84.70. Completamento della pila con i valori dei registri.



Superato il problema della ricostruzione della pila dei dati, la funzione ***proc_sys_exec()*** predispone i descrittori di standard input, standard output e standard error, quindi libera la memoria usata dal processo chiamante e ne rimpiazza i dati nella tabella dei processi con quelli del nuovo processo caricato.

84.5.2 Il codice iniziale dell'applicativo

I programmi iniziano con il codice che si trova nel file '`applic/crt0.mer.s`', oppure '`applic/crt0.sep.s`', a seconda che si compilino in modo da avere codice e dati nello stesso segmento, oppure in segmenti di memoria differenti. Questo file è abbastanza diverso da '`kernel/main/crt0.s`' del kernel; in particolare va osservato che, a differenza del kernel, il codice delle applicazioni viene eseguito in un momento in cui l'indice della pila è già collocato correttamente; inoltre, se la funzione ***main()*** delle applicazioni termina e restituisce il controllo a '`crt0.*.s`', un ciclo senza fine esegue continuamente una chiamata di sistema per la conclusione del processo elaborativo corrispondente.

Figura 84.71. Codice iniziale degli applicativi e variabile strutturata di tipo '**header_t**'.

```
.section .text
startup:
    jmp startup_code
filler:
    .space (0x0004 - (filler - startup))
magic:
    .quad 0x6F7333326170706C   # os32appl
doffset:
    .int _text_start|_text_end
etext:
    .int _text_end
edata:
    .int _data_end
ebss:
    .int _bss_end
stack_size:
    .int 0x8000
.align 4
startup_code:
…
```

```
typedef struct {
  uint32_t filler0;
  uint64_t magic;
  uint32_t data_offset;
  uint32_t etext;
  uint32_t edata;
  uint32_t ebss;
  uint32_t ssize;
} header_t;
```

La figura mostra il confronto tra il codice iniziale contenuto nel file '`applic/crt0.*.s`', senza preamboli e senza commenti, con la dichiarazione del tipo derivato '**header_t**', presente nel file '`kernel/proc.h`' (nel codice si può notare la differenza tra '`crt0.mer.s`' e '`crt0.sep.s`', relativa al valore assegnato alla variabile ***doffset***). Attraverso questa struttura, la funzione ***proc_sys_exec()*** è in grado di estrapolare dal file le informazioni necessarie a caricarlo correttamente in memoria.

Come già accennato, quando viene eseguito il codice di un programma applicativo, la pila dei dati è già operativa. Pertanto, dopo il simbolo '**startup_code**' si può già lavorare con questa.

```
    pop %eax            # argc
    pop %ebx            # argv
    pop %ecx            # envp
    mov %ecx, environ   # Variable 'environ' comes
                        # from <unistd.h>.
    push %ecx
    push %ebx
    push %eax
```

Per prima cosa, viene estratto dalla pila il puntatore all'array noto come ***envp[]***, per poter assegnare tale valore alla variabile ***environ***, come richiede lo standard della libreria POSIX. Tuttavia, per poter gestire poi le variabili di ambiente, si rende necessario utilizzare un array più «comodo», quando le stringhe vanno sostituite. A tale proposito, nel file '`lib/stdlib/environment.c`', si dichiarano ***_environment_table[][]*** e ***_environment[]***. Il primo è semplicemente un array di caratteri, dove, utilizzando due indici di accesso, si conviene di allocare delle stringhe, con una dimensione massima prestabilita. Il secondo, invece, è un array di puntatori, per localizzare l'inizio delle stringhe contenute nel primo. In pratica, alla fine ***_environment[]*** e ***environ[]*** devono essere equivalenti. Ma per attuare questo, occorre utilizzare la funzione ***_environment_setup()*** che sistema tutti i puntatori necessari.

```
    push %ecx
    call _environment_setup
    add  $4, %esp

    mov  $_environment, %eax
    mov  %eax, environ

    pop %eax            # argc
    pop %ebx            # argv[][]
    pop %ecx            # envp[][]
    mov $_environment, %ecx
    push %ecx
    push %ebx
    push %eax
```

Come si vede dall'estratto del file '`applic/crt0.*.s`', si vede l'uso della funzione ***_environment_setup()*** (il registro ECX contiene già il puntatore a ***envp[]***, e viene inserito nella pila proprio come argomento per la funzione). Successivamente viene riassegnata anche

la variabile ***environ*** in modo da coincidere con ***_environment***. Alla fine, viene ricostruita la pila per gli argomenti della chiamata della funzione ***main()***, ma prima di procedere con quella chiamata, si utilizzano delle funzioni, per inizializzare la gestione dei flussi di file e delle directory, sempre in forma di flussi, e per predisporre la tabella delle funzioni da eseguire alla conclusione del processo.

```
    call _stdio_stream_setup
    call _dirent_directory_stream_setup
    call _atexit_setup

    call main

    mov  %eax, exit_value
    …
.align 4
.section .data
exit_value:
    .int 0x00000000
.align 4
.section .bss
```

La funzione ***_stdio_stream_setup()***, contenuta nel file ‘`lib/stdio/FILE.c`’, associa i descrittori standard ai flussi di file standard (standard input, standard output e standard error); la funzione ***_dirent_directory_stream_setup()***, contenuta nel file ‘`lib/dirent/DIR.c`’, compie un lavoro analogo, limitandosi però a inizializzare un array di flussi di directory; la funzione ***_atexit_setup()***, contenuta nel file ‘`lib/stdlib/atexit.c`’ azzera l’array ***_atexit_table[]***, destinato a contenere l’elenco di funzioni da eseguire alla conclusione del processo.

Dopo queste preparazioni, viene chiamata la funzione ***main()***, la quale riceve regolarmente i propri argomenti previsti. Il valore restituito dalla funzione viene poi salvato in corrispondenza del simbolo ‘**`exit_value`**’.

```
halt:
    pushl $2            # Size of message.
    pushl $exit_value   # Pointer to the message.
    pushl $6            # SYS_EXIT
    call sys
    add  $4, %esp
    add  $4, %esp
    add  $4, %esp
    jmp halt
```

All’uscita dalla funzione ***main()***, dopo aver salvato quanto restituito dalla funzione stessa, ci si introduce nel codice successivo al simbolo ‘**`halt`**’, nel quale si chiama la funzione ***sys()*** (chiamata di sistema), per produrre la chiusura formale del processo. Ciò che si vede è comunque l’equivalente di ‘**`_exit (`*exit_value*`);`**’.

84.6 Gestione della memoria

Dal punto di vista del kernel di os32, l’allocazione della memoria riguarda la collocazione dei processi elaborativi nella stessa. Per semplicità si utilizza una mappa di bit per indicare lo stato dei blocchi di memoria, dove un bit a uno indica un blocco di memoria occupato.

Nel file ‘`memory.h`’ viene definita la dimensione di un blocco di memoria e, di conseguenza, la quantità massima che possa essere gestita. Attualmente i blocchi sono da 4096 byte, pertanto, sapendo che la memoria può arrivare solo fino a 4 Gibyte, si gestiscono al massimo 1048576 blocchi.

Per la scansione della mappa si utilizzano interi da 32 bit, pertanto tutta la mappa si riduce a 32768 di questi interi, ovvero 128 Kibyte. Nell’ambito di ogni intero da 32 bit, il bit più significativo rappresenta il primo blocco di memoria di sua competenza. Per esempio, per indicare che si stanno utilizzando i primi 28672 byte, pari ai primi 7 blocchi di memoria, si rappresenta la mappa della memoria come «FE0000000...».

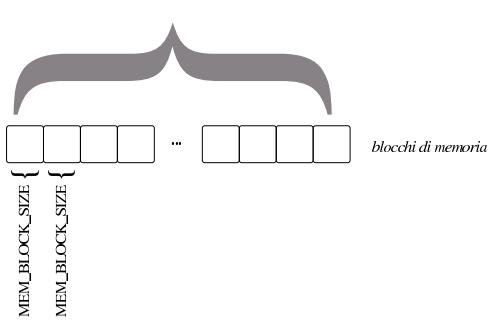
Il fatto che la mappa della memoria vada scandito a ranghi di 32 bit va tenuto in considerazione, perché se invece si andasse con ranghi differenti, si incapperebbe nel problema dell’inversione dei byte.

84.6.1 File «kernel/memory.h» e «kernel/memory/...»

Listato 94.10 e successivi.

Il file ‘`kernel/memory.h`’, oltre ai prototipi delle funzioni usate per la gestione della memoria, definisce la dimensione del blocco minimo di memoria e la quantità massima di questi, rispettivamente con le macro-variabili ***MEM_BLOCK_SIZE*** e ***MEM_MAX_BLOCKS***; inoltre predispone il tipo derivato ‘**`addr_t`**’, corrispondente a un indirizzo di memoria reale.

Figura 84.76. Mappa della memoria in blocchi: la dimensione minima di un’area di memoria è di ***MEM_BLOCK_SIZE*** byte.



Nei file della directory ‘`kernel/memory/`’ viene dichiarata la mappa della memoria, corrispondente a un array di interi a 32 bit, denominato ***mb_table[]***. L’array è pubblico, tuttavia è disponibile anche una funzione che ne restituisce il puntatore: ***mb_reference()***. Tale funzione sarebbe perfettamente inutile, ma rimane per uniformità rispetto alla gestione delle altre tabelle.

Tabella 84.77. Funzioni per la gestione della mappa della memoria, dichiarate nel file di intestazione ‘`kernel/memory.h`’ e realizzate nella directory ‘`kernel/memory/`’.

Funzione	Descrizione
`uint32_t *mb_reference (void);`	Restituisce il puntatore alla tabella dei blocchi di memoria, per uniformare l’accesso alla tabella dalle funzioni che non fanno parte del gruppo contenuto nella directory ‘`kernel/memory/`’.
`ssize_t mb_alloc (addr_t` ***address***`,` `size_t` ***size***`);`	Alloca la memoria a partire dall’indirizzo indicato, per la quantità di byte richiesta. L’allocazione ha termine anticipatamente se si incontra un blocco già utilizzato. La funzione restituisce la dimensione allocata effettivamente.
`void mb_free (addr_t` ***address***`,` `size_t` ***size***`);`	Libera la memoria a partire dall’indirizzo indicato, per la quantità di byte richiesta. Lo spazio viene liberato in ogni caso, anche se risulta già libero; tuttavia viene prodotto un avvertimento a video se si verifica tale ipotesi.

`int mb_reduce (addr_t` ***address***`,`
`size_t` ***new***`,`
`size_t` ***previous***`);`
	Riduce un’area di memoria già utilizzata. Restituisce zero se l’operazione si conclude con successo, oppure `-1` in caso contrario, aggiornando la variabile ***errno*** di conseguenza.

`void mb_clean (addr_t` ***address***`,` `size_t` ***size***`);`	Azzera l’area di memoria specificata.

Funzione	Descrizione
`addr_t mb_alloc_size (size_t size);`	Alloca un'area di memoria della dimensione richiesta, restituendone l'indirizzo. La funzione conclude con successo il proprio lavoro se il valore restituito è diverso da zero; se invece l'indirizzo ottenuto è pari a zero si è verificato un errore che può essere verificato analizzando il contenuto della variabile ***errno***.
`void mb_size (size_t size);`	Questa funzione, usata una sola volta all'interno di ***kmain()***, serve a definire la dimensione massima della memoria disponibile in blocchi. In pratica, le si fornisce la dimensione effettiva della memoria che viene così divisa per la dimensione del blocco, ignorando il resto. Questa informazione viene conservata nella variabile ***mb_max***.
`void mb_print (void);`	Funzione diagnostica che visualizza gli intervalli di memoria utilizzati, esprimendoli però in blocchi.

84.6.2 Scansione della mappa di memoria

Listato 94.10 e successivi.

La mappa della memoria si rappresenta (a sua volta in memoria), con un array di interi a 32 bit, dove ogni bit individua un blocco di memoria. Pertanto, l'array si compone di una quantità di elementi pari al valore di ***MEM_MAX_BLOCKS*** diviso 32.

Il primo elemento di questo array, ovvero ***mb_table[0]***, individua i primi 32 blocchi di memoria, dove il bit più significativo si riferisce precisamente al primo blocco. Per esempio, se ***mb_table[0]*** contiene il valore $F8000000_{16}$, ovvero 1111100000000000_2, significa che i primi cinque blocchi di memoria sono occupati, mentre i blocchi dal sesto al trentaduesimo sono liberi.

Dal momento che i calcoli per individuare i blocchi di memoria e per intervenire nella mappa relativa, possono creare confusione, queste operazioni sono raccolte in funzioni statiche separate, anche se sono utili esclusivamente all'interno del file in cui si trovano. Tali funzioni statiche hanno una sintassi comune:

```
int mb_block_set1   (int block)
int mb_block_set0   (int block)
int mb_block_status (int block)
```

Le funzioni ***mb_block_set1()*** e ***mb_block_set0()*** servono rispettivamente a impegnare o liberare un certo blocco di memoria, individuato dal valore dell'argomento. La funzione ***mb_block_status()*** restituisce uno nel caso il blocco indicato risulti allocato, oppure zero in caso contrario.

Queste tre funzioni usano un metodo comune per scandire la mappa della memoria: il valore che rappresenta il blocco a cui si vuole fare riferimento, viene diviso per 32, ovvero il rango degli elementi dell'array che rappresenta la mappa della memoria. Il risultato intero della divisione serve per trovare quale elemento dell'array considerare, mentre il resto della divisione serve per determinare quale bit dell'elemento trovato rappresenta il blocco desiderato. Trovato ciò, si deve costruire una maschera, nella quale si mette a uno il bit che rappresenta il blocco; per farlo, si pone inizialmente a uno il bit più

significativo della maschera, quindi lo si fa scorrere verso destra di un valore pari al resto della divisione.

Per esempio, volendo individuare il terzo blocco di memoria, pari al numero 2 (il primo blocco corrisponderebbe allo zero), si avrebbe che questo è descritto dal primo elemento dell'array (in quanto 2/32 dà zero, come risultato intero), mentre la maschera necessaria a trovare il bit corrispondente è $00100000000000000000000000000000_2$, la quale si ottiene spostando per due volte verso destra il bit più significativo (due volte, pari al resto della divisione).

Una volta determinata la maschera, per segnare come occupato un blocco di memoria, basta utilizzare l'operatore OR binario:

```
mb_table[i] = mb_table[i] | mask;
```

Se invece si vuole liberare un blocco di memoria, si utilizza un AND binario, invertendo però il contenuto della maschera:

```
mb_table[i] = mb_table[i] & ~mask;
```

Va osservato che la rappresentazione dei blocchi nella mappa è invertita rispetto ad altri sistemi operativi, in quanto non sarebbe tanto logico il fatto che il bit più significativo si riferisca invece alla parte più bassa del proprio insieme di blocchi di memoria. La scelta è dovuta al fatto che, volendo rappresentare la mappa numericamente, la lettura di questa sarebbe più vicina a quella che è la percezione umana del problema.

84.7 Dispositivi

La gestione dei dispositivi fisici, da parte di os32, è limitata ed essenziale. Tutte le operazioni di lettura e scrittura di dispositivi, passano attraverso la gestione comune della funzione ***dev_io()***.

Nel file '`lib/sys/os32.h`' (listato 95.21), disponibile sia al kernel, sia alle applicazioni, sono elencate le macro-variabili che descrivono tutti i dispositivi previsti in forma numerica. Queste macro-variabili hanno nomi prefissati dalla sigla ***DEV_***…. Per esempio, ***DEV_DM_MAJOR*** corrisponde al numero primario (*major*) per le unità di memorizzazione di massa, ***DEV_DM00*** corrisponde al numero primario e secondario (*major* e *minor*), in un valore unico, della prima unità di memorizzazione di massa complessiva, mentre ***DEV_DM01*** corrisponde alla prima partizione della stessa.

84.7.1 File «kernel/dev.h» e «kernel/dev/…»

Listati 94.3 e successivi.

Il file '`kernel/dev.h`' incorpora il file '`lib/sys/os32/os32.h`', per acquisire le macro-variabili della gestione dei dispositivi che sono disponibili anche agli applicativi. Successivamente dichiara la funzione ***dev_io()***, la quale sintetizza tutta la gestione dei dispositivi. Questa funzione utilizza il parametro ***rw***, per specificare l'azione da svolgere (lettura o scrittura). Per questo parametro vanno usate le macro-variabili ***DEV_READ*** e ***DEV_WRITE***, così da non dover ricordare quale valore numerico corrisponde alla lettura e quale alla scrittura.

```
ssize_t dev_io (pid_t pid, dev_t device, int rw, off_t offset,
                void *buffer, size_t size, int *eof);
```

Sono comunque descritte anche altre funzioni, ma utilizzate esclusivamente da ***dev_io()***.

La funzione ***dev_io()*** si limita a estrapolare il numero primario dal numero del dispositivo complessivo, quindi lo confronta con i vari tipi gestibili. A seconda del numero primario seleziona una funzione appropriata per la gestione di quel tipo di dispositivo, passando praticamente gli stessi argomenti già ricevuti.

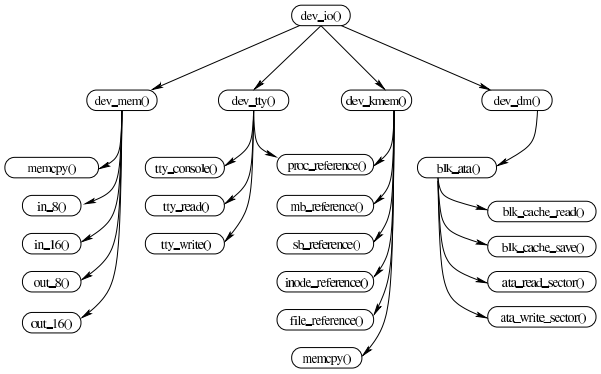
Va osservato il caso particolare dei dispositivi ***DEV_KMEM_***…. In un sistema operativo Unix comune, attraverso ciò che fa capo al file di dispositivo '`/dev/kmem`', si ha la possibilità di accedere all'immagine in memoria del kernel, lasciando a un programma con privilegi adeguati la facoltà di interpretare i simboli che consentono di

individuare i dati esistenti. Nel caso di os32, non ci sono simboli nel risultato della compilazione, quindi non è possibile ricostruire la collocazione dei dati. Per questa ragione, le informazioni che devono essere pubblicate, vengono controllate attraverso un dispositivo specifico. Quindi, il dispositivo ***DEV_KMEM_PS*** consente di leggere la tabella dei processi, ***DEV_KMEM_MMAP*** consente di leggere la mappa della memoria, e così vale anche per altre tabelle.

Per quanto riguarda la gestione dei terminali, attraverso la funzione ***dev_tty()***, quando un processo vuole leggere dal terminale, ma non risulta disponibile un carattere, questo viene messo in pausa, in attesa di un evento legato ai terminali.

os32 gestisce virtualmente tutti i dispositivi come se fossero a caratteri. Tuttavia, nel caso delle unità di memorizzazione di massa il flusso di caratteri, in lettura o in scrittura, viene scomposto in blocchi, sfruttando anche una memoria (*cache*) per questi. Pertanto, la funzione ***dev_dm()*** si avvale di ***blk_ata()***.

Figura 84.80. Interdipendenza tra la funzione ***dev_io()*** e le altre. I collegamenti con le funzioni ***major()*** e ***minor()*** sono omesse.



84.7.2 File «kernel/blk.h» e «kernel/blk/...»

Listati 94.2 e successivi.

I file contenuti nella directory '`kernel/blk/`' riguardano specificatamente la gestione della memoria *cache* per i blocchi di dati usati più di frequente, relativamente ai dispositivi di memorizzazione. In pratica, tale gestione riguarda esclusivamente le unità PATA.

La tabella ***blk_table()*** è composta da elementi '**`blk_cache_t`**', ognuno dei quali rappresenta un blocco singolo, con l'indicazione del dispositivo (dell'unità intera e non di una singola partizione) e del numero di blocco a cui si riferisce, assieme a un numero che ne rappresenta l'«età».

Inizialmente, la funzione ***blk_cache_init()***, usata una volta sola all'interno di ***kmain()***, si azzerano le informazioni sul numero di dispositivo e sul numero del blocco di ogni elemento della tabella, quindi si assegna l'età attraverso un numero progressivo, da 0 a ***BLK_CACHE_MAX_AGE***. Il numero più basso rappresenta l'ultimo blocco letto o modificato, mentre quello più alto riguarda il blocco che da più tempo non è stato utilizzato.

Quando la funzione ***blk_ata()*** deve leggere un blocco da un'unità PATA, prima, attraverso la funzione ***blk_cache_read()***, controlla all'interno della tabella ***blk_table()*** esiste già una copia del blocco; questo viene trovato, la funzione ***blk_cache_read()*** ne azzera l'età, incrementando conseguentemente l'età dei blocchi che avevano prima un valore inferiore al suo. Se il blocco viene trovato nella tabella, la funzione non interpella l'hardware PATA e conclude il suo lavoro, altrimenti provvede alla lettura necessaria e al suo salvataggio nella tabella dei blocchi, con l'aiuto di ***blk_cache_save()***, la quale aggiorna il blocco se questo era già presente nella tabella, oppure rimpiazza il blocco di età maggiore, aggiornando di conseguenza l'età, come nel caso della lettura.

Quando la funzione ***blk_ata()*** deve scrivere un blocco, la scrittura hardware avviene in ogni caso, seguita dal salvataggio nella tabella dei blocchi.

In pratica, la memoria *cache* viene usata solo per le letture, pertanto tutte le scritture sono sincrone.

84.7.3 Numero primario e numero secondario

I dispositivi, secondo la tradizione dei sistemi Unix, sono rappresentati dal punto di vista logico attraverso un numero intero, senza segno, a 16 bit. Tuttavia, per organizzare questa numerazione in modo ordinato, tale numero viene diviso in due parti: la prima parte, nota come *major*, ovvero «numero primario», si utilizza per individuare il tipo di dispositivo; la seconda, nota come *minor*, ovvero «numero secondario», si utilizza per individuare precisamente il dispositivo, nell'ambito del tipo a cui appartiene.

In pratica, il numero complessivo a 16 bit si divide in due, dove gli 8 bit più significativi individuano il numero primario, mentre quelli meno significativi danno il numero secondario. L'esempio seguente si riferisce al dispositivo che genera il valore zero, il quale appartiene al gruppo dei dispositivi relativi alla memoria:

`DEV_MEM_MAJOR`	01_{16}
`DEV_ZERO`	0104_{16}

In questo caso, il valore che rappresenta complessivamente il dispositivo è 0104_{16} (pari a 260_{10}), ma si compone di numero primario 01_{16} e di numero secondario 04_{16} (che coincidono nella rappresentazione in base dieci). Per estrarre il numero primario si deve dividere il numero complessivo per 256 (0100_{16}), trattenendo soltanto il risultato intero; per filtrare il numero secondario si può fare la stessa divisione, ma trattenendo soltanto il resto della stessa. Al contrario, per produrre il numero del dispositivo, partendo dai numeri primario e secondario separati, occorre moltiplicare il numero primario per 256, sommando poi il risultato al numero secondario.

84.7.4 Dispositivi previsti

L'astrazione della gestione dei dispositivi, consente di trattare tutti i componenti che hanno a che fare con ingresso e uscita di dati, in modo sostanzialmente omogeneo; tuttavia, le caratteristiche effettive di tali componenti può comportare delle limitazioni o delle peculiarità. Ci sono alcune questioni fondamentali da considerare: un tipo di dispositivo potrebbe consentire l'accesso in un solo verso (lettura o scrittura); l'accesso al dispositivo potrebbe essere ammesso solo in modo sequenziale, rendendo inutile l'indicazione di un indirizzo; la dimensione dell'informazione da trasferire potrebbe assumere un significato differente rispetto a quello comune.

Tabella 84.82. Classificazione dei dispositivi di os32.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
`DEV_MEM`	r/w	diretto	Permette l'accesso alla memoria, in modo indiscriminato; tuttavia, solo al kernel è permessa la scrittura.
`DEV_NULL`	r/w	nessuno	Consente la lettura e la scrittura, ma non si legge e non si scrive alcunché.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
DEV_PORT	r/w	sequenziale	Consente di leggere e scrivere da o verso una porta di I/O, individuata attraverso l'indirizzo di accesso (l'indirizzo, o meglio lo scostamento, viene trattato come la porta a cui si vuole accedere). Tuttavia, la dimensione dell'informazione da trasferire è valida solo se si tratta di uno o di due byte: per la dimensione di un byte si usano le funzioni ***in_8()*** e ***out_8()***; per due byte si usano le funzioni ***in_16()*** e ***out_16()***. Per dimensioni differenti la lettura o la scrittura non ha effetto.
DEV_ZERO	r	sequenziale	Consente solo la lettura di valori a zero (zero inteso in senso binario).
DEV_TTY	r/w	sequenziale	Rappresenta il terminale virtuale del processo attivo.
DEV_DM***mn***	r/w	diretto	Rappresenta la partizione ***n*** dell'unità di memorizzazione ***m***. La prima unità PATA disponibile ottiene il dispositivo ***DEV_DM00***, la seconda il numero ***DEV_DM10***, ecc.
DEV_KMEM_PS	r	diretto	Rappresenta la tabella contenente le informazioni sui processi. L'indirizzo di accesso indica il numero del processo di partenza; la dimensione da leggere dovrebbe essere abbastanza grande da contenere un processo, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_MMP	r	sequenziale	Rappresenta la mappa della memoria, alla quale si può accedere solo dal suo principio. In pratica, l'indirizzo di accesso viene ignorato, mentre conta solo la quantità di byte richiesta.
DEV_KMEM_SB	r	diretto	Rappresenta la tabella dei super blocchi (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il super blocco; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un super blocco, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_INODE	r	diretto	Rappresenta la tabella degli inode (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare l'inode; la dimensione richiesta dovrebbe essere abbastanza grande da contenere un inode, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.

Dispositivo	Lettura e scrittura r/w	Accesso diretto o sequenziale	Annotazioni
DEV_KMEM_FILE	r	diretto	Rappresenta la tabella dei file (per la gestione delle unità di memorizzazione). L'indirizzo di accesso serve a individuare il file; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di un file, ma anche richiedendo una dimensione maggiore, se ne legge uno solo.
DEV_KMEM_ARP	r	diretto	Rappresenta la tabella ARP (per la trasformazione degli indirizzi IPv4 in indirizzi Ethernet). L'indirizzo di accesso serve a individuare la voce; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
DEV_KMEM_NET	r	diretto	Rappresenta la tabella delle interfacce di rete. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
DEV_KMEM_ROUTE	r	diretto	Rappresenta la tabella degli instradamenti IPv4. L'indirizzo di accesso serve a individuare la voce della tabella; la dimensione richiesta dovrebbe essere abbastanza grande da contenere le informazioni di una voce, ma anche richiedendo una dimensione maggiore, se ne legge una sola.
DEV_CONSOLE	r/w	sequenziale	Legge o scrive relativamente alla console attiva la quantità di byte richiesta, ignorando l'indirizzo di accesso.
DEV_CONSOLE***n***	r/w	sequenziale	Legge o scrive relativamente alla console ***n*** la quantità di byte richiesta, ignorando l'indirizzo di accesso.

84.7.5 Gestione del terminale

Listato 94.4.42 e successivi.

Il terminale offre solo la funzionalità elementare della modalità canonica, dove è possibile scrivere o leggere sequenzialmente. Ci sono al massimo quattro terminali virtuali, selezionabili attraverso le combinazioni di tasti [*Ctrl q*], [*Ctrl r*], [*Ctrl s*] e [*Ctrl t*] e non è possibile controllare i colori o la posizione del testo che si va a esporre; in pratica si opera come su una telescrivente. Le funzioni di livello più basso, relative al terminale hanno nomi che iniziano per '**`tty_…()`**'.

Per la gestione dei quattro terminali virtuali, si utilizza una tabella, in cui ogni voce rappresenta lo stato del terminale virtuale che rappresenta. La tabella è costituita dall'array ***tty_table[]*** che contiene ***TTY_TOTALS*** elementi. L'array è dichiarato nel file '`kernel/driver/tty_public.c`', mentre la macro-variabile

TTY_TOTALS appare nel file ‘`kernel/driver/tty.h`’. Gli elementi di ***tty_table[]*** sono di tipo ‘**`tty_t`**’:

```
typedef struct {
    dev_t          device;
    pid_t          pgrp;            // Process group.
    struct termios attr;            // termios attributes.
    unsigned char  status;          // 0 = edit,
                                    // 1 = end edit.
    char           line[MAX_CANON]; // Canonical input line.
    int            lpr;             // Input line position
                                    // read.
    int            lpw;             // Input line position
                                    // write.
} tty_t;
```

Il membro ***attr*** della voce di un terminale è una variabile strutturata di tipo ‘**`struct termios`**’, come previsto nel file ‘`termios.h`’ della libreria standard.

L’input del terminale, proveniente dalla tastiera, viene depositato dalla funzione ***proc_sch_terminals()*** all’interno del membro ***line[]***, annotando in ***lpw*** l’indice di scrittura. Quando si legge dal terminale, si ottiene un carattere alla volta da ***line[]***, con l’ausilio dell’indice ***lpr***. Quando il terminale virtuale riceve input dalla tastiera, è nello stato definito dalla macro-variabile ***TTY_INPUT_LINE_EDITING***, mentre quando l’inserimento risulta concluso, per esempio perché è stato premuto il tasto [*Invio*], lo stato è quello di ***TTY_INPUT_LINE_CLOSED*** ed è possibile procedere con la lettura del contenuto di ***line[]***: quando la lettura termina perché l’indice ***lpr*** ha raggiunto ***lpw***, gli indici vengono azzerati e lo stato ritorna quello di inserimento. Quando un processo tenta di leggere dal terminale, mentre questo è in fase di inserimento, non ancora concluso, viene sospeso e rimane così fino alla conclusione dell’inserimento stesso.

Figura 84.84. A sinistra le fasi dell’inserimento di una riga da tastiera; a destra le fasi della lettura attraverso la funzione ***tty_read()***.

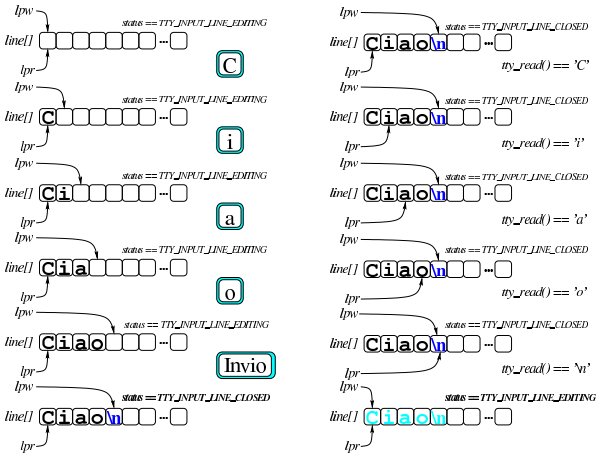


Tabella 84.85. Funzioni per l’accesso al terminale, dichiarate nel file di intestazione ‘`kernel/driver/tty.h`’ e descritte nei file contenuti nella directory ‘`kernel/driver/tty/`’.

Funzione	Descrizione
`dev_t tty_console (dev_t` ***device*** `);`	Seleziona un terminale virtuale, rendendolo attivo, specificandone il numero del dispositivo. La funzione restituisce il dispositivo attivo in precedenza e se le viene fornito solo il valore zero, il terminale virtuale non cambia, ma si ottiene comunque di conoscere qual è quello attuale.
`void tty_init (void);`	Inizializza la gestione dei terminali virtuali, popolando anche la tabella ***tty_table[]*** con i valori predefiniti. Questa funzione viene usata una volta sola all’interno di ***kmain()***.
`int tty_read (dev_t` ***device*** `);`	Legge un carattere dal terminale virtuale specificato attraverso il numero di dispositivo. La lettura avviene solo se l’input da tastiera risulta concluso, altrimenti la funzione restituisce il valore `-1`.
`tty_t *tty_reference (dev_t` ***device*** `);`	Restituisce il puntatore alla voce della tabella ***tty_table[]*** contenente le informazioni sul terminale virtuale indicato attraverso il numero di dispositivo. Se il dispositivo indicato non è valido, si ottiene il puntatore nullo; se viene richiesto il dispositivo indefinito, si ottiene il puntatore all’inizio della tabella.
`void tty_write (dev_t` ***device***`, int` ***c***`);`	Scrive un carattere sullo schermo del terminale specificato.

84.7.5.1 Gestione della tastiera

Listato 94.4.15 e successivi.

Per la gestione della tastiera, nel file ‘`kernel/driver/kbd_public.c`’ viene dichiarata una variabile strutturata, di tipo ‘**`kbd_t`**’, contenente le informazioni sullo stato della stessa e sulla mappa di trasformazione da applicare. A differenza della gestione complessiva dei terminali, in cui ogni terminale virtuale ha un proprio insieme di dati, per la tastiera questo è unico.

Listato 84.86. Definizione del tipo ‘**`kbd_t`**’, contenuto nel file ‘`kernel/driver/kbd.h`’.

```
typedef struct {
    bool          shift;
    bool          shift_lock;
    bool          ctrl;
    bool          alt;
    bool          echo;
    unsigned char key;
    unsigned char map1[128];
    unsigned char map2[128];
} kbd_t;
```

Nella variabile ***kbd***, come si intuisce dai nomi dei suoi membri, viene annotato lo stato di pressione dei tasti delle maiuscole, dei tasti [*Ctrl*] e [*Alt*], per poter recepire eventuali combinazioni di tasti; inoltre, il membro ***echo***, se attivo, indica la richiesta di vedere sullo schermo ciò che si digita.

Dalla tastiera viene recepito un solo tasto alla volta: se questo si traduce in un carattere, stampabile o meno che sia, questo viene depositato nel membro ***key***, da dove la funzione ***proc_sch_terminals()*** deve provvedere a prelevarlo (per trasferirlo nel membro ***line[]*** della voce che descrive il terminale virtuale attivo), azzerando nuovamente ***key***. Fino a quando il membro ***key*** ha un valore diverso da zero, non è possibile recepire altro dalla tastiera.

Dalla tastiera è possibile ottenere solo i caratteri ASCII; in particolare, quelli non stampabili si ottengono per combinazione con il tasto [*Ctrl*], secondo la convenzione tradizionale. Non sono previste altre funzionalità.

Tabella 84.87. Funzioni per la gestione della tastiera, dichiarate nel file di intestazione '`kernel/driver/kbd.h`' e descritte nei file contenuti nella directory '`kernel/driver/kbd/`'.

Funzione	Descrizione
`void kbd_isr (void);`	Questa funzione è chiamata dalla routine di gestione delle interruzioni da tastiera, contenuta nel file '`kernel/ibm_i386/isr.s`'. La funzione legge un carattere dalla porta di I/O 60_{16}, quindi lo interpreta e aggiorna il contenuto della variabile strutturata ***kbd*** di conseguenza.
`void kbd_load (void);`	Questa funzione è chiamata una sola volta da ***kmain()***, per associare la mappa della tastiera ai codici prodotti dalla stessa. Attualmente questa funzione produce esclusivamente l'associazione necessaria per una tastiera italiana.

La funzione ***proc_scheduler()***, il cui scopo principale è quello di alternare i processi in esecuzione, tra le altre cose, avvia ogni volta la funzione ***proc_scheduler_terminals()***. La funzione ***proc_scheduler_terminals()*** verifica se nella variabile ***kbd.key*** è disponibile un valore diverso da zero e, se c'è, lo acquisisce per conto del terminale attivo. Prima di tutto verifica se si tratta di una combinazione di tasti che richiede lo scambio a un altro terminale virtuale; poi controlla se si tratta di un codice di interruzione (come quello provocato da [*Ctrl c*]) e, se la configurazione del terminale attivo lo permette, conclude il processo più interno appartenente al gruppo che risulta connesso al terminale stesso; alla fine, dopo altre ipotesi particolari, se si tratta di un carattere «normale» e il terminale si trova in fase di inserimento (***TTY_INPUT_LINE_EDITING***), questo viene depositato nell'array ***line[]***, con il conseguente aggiornamento dell'indice di scrittura al suo interno; ricevendo invece un codice che rappresenta la conclusione dell'inserimento, si rimette il terminale nello stato di conclusione dell'inserimento (***TTY_INPUT_LINE_CLOSED***).

84.7.5.2 Gestione dello schermo

«

Listato 94.4.30 e successivi.

Lo schermo di os32 viene gestito secondo quanto prescrive l'hardware VGA (come descritto nella sezione 83.3), per cui ciò che si vuole fare apparire deve essere scritto in memoria a partire dall'indirizzo $B8000_{16}$, usando per ogni carattere 16 bit (8 bit di questo gruppo servono per gli attributi).

Dal momento che si gestiscono dei terminali virtuali, per ognuno di questi occorre tenere una copia dell'immagine dello schermo, così, quando si seleziona un terminale differente, la copia di quel terminale viene usata per sovrascrivere l'area di memoria che rappresenta lo schermo. Per la gestione degli schermi virtuali si usa una tabella, denominata ***screen_table[]***, composta da voci di tipo '**`screen_t`**'.

Listato 84.88. Definizione del tipo '**`screen_t`**', contenuto nel file '`kernel/driver/screen.h`'.

```
typedef struct {
    uint16_t  cell[SCREEN_CELLS];
    int       position;
} screen_t;
```

All'interno della struttura rappresentata dal tipo '**`screen_t`**', si vede un array che riproduce la rappresentazione in memoria dello stesso, da copiare a partire dall'indirizzo $B8000_{16}$, quando lo schermo virtuale diventa quello attivo; inoltre si vede il membro ***position***, usato per ricordare la posizione in cui si trova il cursore.

Tabella 84.89. Funzioni per la gestione dello schermo, dichiarate nel file di intestazione '`kernel/driver/screen.h`' e descritte nei file contenuti nella directory '`kernel/driver/screen/`'.

Funzione	Descrizione
`int screen_clear (screen_t *screen);`	Ripulisce il contenuto dello schermo selezionato, riposizionando il cursore all'inizio.
`screen_t *screen_current (void);`	Restituisce il puntatore alla voce della tabella ***screen_table[]*** che descrive lo schermo virtuale attivo.
`void screen_init (void);`	Inizializza la gestione degli schermi virtuali, ripulendoli e collocando il cursore all'inizio. Questa funzione viene usata da ***tty_init()***.
`int screen_newline (screen_t *screen);`	Produce sullo schermo virtuale selezionato un avanzamento alla riga successiva. Ciò può comportare semplicemente il riposizionamento del cursore, oppure lo scorrimento in avanti del contenuto, quando il cursore si trova già sull'ultima riga visualizzabile.
`int screen_number (screen_t *screen);`	Restituisce il numero dello schermo corrispondente al puntatore fornito, purché questo sia valido. È in pratica l'opposto della funzione ***screen_pointer()***.
`screen_t *screen_pointer (int scrn);`	Restituisce il puntatore alla voce della tabella ***screen_table[]*** che descrive lo schermo virtuale indicato per numero. È in pratica l'opposto della funzione ***screen_number()***.
`int screen_putc (screen_t *screen, int c);`	Colloca sullo schermo virtuale individuato dal puntatore che costituisce il primo parametro, il carattere richiesto come secondo. Se il carattere in questione è *<CR>* o *<LF>*, si produce un avanzamento alla riga successiva, mentre con un carattere *<BS>* si produce un arretramento del cursore.
`uint16_t screen_cell (c, attributo);`	Si tratta di una macroistruzione che produce il valore corretto per una cella dello schermo VGA, contenente sia l'informazione sul carattere, sia quella dell'attributo associato.
`int screen_scroll (screen_t *screen);`	Fa scorrere in avanti lo schermo, di una riga, ricollocando di conseguenza il cursore.
`int screen_select (screen_t *screen);`	Seleziona lo schermo indicato come schermo attivo, facendone apparire il contenuto sullo schermo VGA reale.

Funzione	Descrizione
`void screen_update (screen_t *screen);`	Aggiorna la memoria VGA sulla base della copia che rappresenta lo schermo virtuale attivo. L'aggiornamento implica anche la collocazione del cursore visibile in corrispondenza della posizione attuale.

84.7.5.3 Configurazione del terminale

Lo standard dei sistemi Unix prescrive che per ogni terminale gestito sia prevista una variabile strutturata, di tipo ***struct termios***, allo scopo di contenere la configurazione dello stesso. os32 gestisce i terminali virtuali soltanto in modalità «canonica», ovvero come se si trattasse di telescriventi, anche se munite di video invece che di carta, pertanto utilizza solo un sottoinsieme delle opzioni previste.

```
typedef uint16_t      tcflag_t;
typedef unsigned char cc_t;
…
struct termios {
    tcflag_t c_iflag;
    tcflag_t c_oflag;
    tcflag_t c_cflag;
    tcflag_t c_lflag;
    cc_t     c_cc[NCCS];
};
```

Il membro ***c_cc[]*** è un array di caratteri di controllo, a cui viene attribuita una definizione.

Tabella 84.91. Caratteri di controllo riconosciuti da os32, secondo le definizioni del file '`termios.h`'.

Definizione	Corrispondenza	Descrizione
VEOF	04_{16} <*EOT*>	Carattere di fine file.
VERASE	08_{16} <*BS*>	Carattere di cancellazione.
VINTR	03_{16} <*ETX*>	Carattere di interruzione.
VQUIT	$1C_{16}$ <*FS*>	Carattere di abbandono.

Il membro ***c_iflag*** serve a contenere opzioni sull'inserimento, ovvero sul controllo della digitazione.

Tabella 84.92. Opzioni del membro ***c_iflag*** riconosciute da os32.

Opzione	Descrizione
BRKINT	Se questa opzione è attiva e, nel contempo, non è attiva ***IGNBRK***, si intendono recepire i codici di interruzione ***VINTR***. Se l'opzione ***ISIG*** del membro ***c_lflag*** è attiva, il processo più interno del gruppo a cui appartiene il terminale viene concluso; in ogni caso, viene annullato il contenuto della riga di inserimento in corso.
ICRNL	Se si riceve il carattere <*CR*>, questo viene convertito in <*NL*>.
IGNBRK	Se questa opzione è attiva, fa sì che il carattere definito come ***VINTR*** sia ignorato.
IGNCR	Se si riceve il carattere <*CR*>, questo viene ignorato semplicemente.
INLCR	Se si riceve il carattere <*NL*>, questo viene convertito in <*CR*>.

Il membro ***c_lflag*** serve a contenere delle opzioni definite come «locali», le quali si occupano in pratica di controllare la visualizzazione della digitazione introdotta e di decidere se l'interruzione ricevuta da tastiera debba produrre l'invio di un segnale di interruzione al processo con cui si sta interagendo. Gli altri due membri della struttura non vengono utilizzati da os32.

Tabella 84.93. Opzioni del membro ***c_lflag*** riconosciute da os32.

Opzione	Descrizione
ECHO	Abilita la visualizzazione sullo schermo del testo inserito da tastiera.
ECHOE	Ammesso che sia attiva l'opzione ***ECHO***, questa abilita il recepimento del carattere definito come ***VERASE*** per cancellare l'ultimo carattere inserito, indietreggiando di una posizione.
ECHONL	Indipendentemente dall'opzione ***ECHO***, questa abilita il recepimento del carattere <*NL*> per fare avanzare il cursore alla riga successiva, con l'eventuale scorrimento in avanti se si trova già sull'ultima.
ISIG	Ammesso che sia recepito e accettato un codice di interruzione, definito come ***VINTR***, con questa opzione si ottiene l'invio di un segnale di interruzione al processo più interno del gruppo collegato al terminale (il processo più interno dovrebbe corrispondere a quello in primo piano al momento della digitazione).

84.7.6 Gestione delle unità di memorizzazione in generale

Listato 94.4 e successivi.

Le unità di memorizzazione vengono viste da os32 attraverso un gruppo di dispositivi astratti, definiti come ***DEV_DM∗***, dove la prima unità PATA disponibile ottiene il numero ***DEV_DM00***, la seconda ***DEV_DM10***,...

Il numero del dispositivo che rappresenta queste unità è composto in modo solito per ciò che riguarda la distinzione tra numero primario e numero secondario, ma il numero secondario si scompone ulteriormente in due parti: l'unità intera e la partizione. Per esempio, il numero 0810_{16} individua la seconda unità di memorizzazione per intero, mentre 0811_{16} rappresenta la prima partizione della seconda unità.

Le unità di memorizzazione riconosciute dal sistema sono raccolte in una tabella, denominata ***dm_table[]***. Ogni elemento di questa tabella contiene l'informazione sul tipo di unità, un puntatore per raggiungere un'altra tabella con le informazioni specifiche sull'unità, in base al tipo di questa (attualmente l'unica tabella in questione può essere quella delle unità PATA), le informazioni sulle partizioni esistenti (ma solo quelle primarie).

Inizialmente, all'interno di ***proc_init()***, viene avviata la funzione ***dm_init()***, la quale a sua volta scandisce le unità PATA attraverso ***ata_init()*** e ne raccoglie le informazioni nella propria tabella ***dm_table()***.

84.7.7 Gestione delle unità PATA

Listato 94.4.3 e successivi.

La gestione delle unità PATA di os32 si limita alla modalità PIO (*programmed input-output*), con accesso LBA28, senza nemmeno considerare le partizioni. La spiegazione sul come avvenga la gestione di un'unità PATA, secondo le stesse modalità usate da os32 è disponibile nella sezione 83.9.

Per la gestione delle unità PATA, os32 utilizza una tabella, denominata ***ata_table[]***, composta da voci di tipo ***ata_t***, ognuna delle quali contiene lo stato di un'unità. L'indice della tabella corrisponde al numero dell'unità, ovvero al parametro ***drive*** di varie funzioni. La tabella è dichiarata formalmente nel file '`kernel/driver/ata/ata_public.c`', mentre il tipo derivato '**`ata_t`**' è descritto nel file '`kernel/driver/ata.h`'. Per comodità, si trova anche il tipo '**`ata_sector_t`**', usato per descrivere lo spazio di memoria usato per collocare la copia di un settore di dati di un'unità PATA.

Tabella 84.94. Funzioni per la gestione delle unità PATA, dichiarate nel file di intestazione '`kernel/driver/ata.h`' e descritte nei file contenuti nella directory '`kernel/driver/ata/`'. Le funzioni sono raggruppate in insiemi logici.

Funzione	Descrizione
`void ata_init (void);`	Inizializza la gestione delle unità PATA, predisponendo i contenuti della tabella ***ata_table[]***, verificando la presenza delle unità. Questa funzione viene usata una volta sola, nella funzione ***proc_init()***.
`void ata_reset (int drive);`	Azzera lo stato di funzionamento dell'unità PATA specificata.
`int ata_valid (int drive);`	Verifica se l'unità richiesta è presente effettivamente. In caso di successo restituisce il valore zero, altrimenti si ottiene -1.

Funzione	Descrizione
`int ata_cmd_identify_device (int drive, void *buffer);`	Richiede all'unità specificata le informazioni sulla sua identificazione. Se l'unità è presente, in corrispondenza del puntatore fornito si ottengono le informazioni nello spazio di un settore (***ATA_SECTOR_SIZE***); l'analisi successiva di questi dati può dare maggiori informazioni sull'unità.
`int ata_cmd_read_sectors (int drive, unsigned int sector, unsigned char count, void *buffer);`	Legge dall'unità ***drive***, a partire dal settore ***sector***, una quantità pari a ***count*** settori, mettendo il risultato a partire dall'indirizzo di memoria ***buffer***. Se ***count*** fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.
`int ata_cmd_write_sectors (int drive, unsigned int sector, unsigned char count, void *buffer);`	Scrive nell'unità ***drive***, a partire dal settore ***sector***, una quantità pari a ***count*** settori, leggendoli a partire dall'indirizzo di memoria ***buffer***. Se ***count*** fosse pari a zero, si intenderebbero 256 settori. Se l'operazione fallisce, restituisce un valore negativo.
`int ata_device (int drive, unsigned int sector);`	Imposta il registro ***device*** dell'unità PATA specificata, con l'indicazione di un numero di settore.

Funzione	Descrizione
`int ata_rdy (int drive, clock_t timeout);`	Attende che l'unità ***drive*** sia pronta, purché ciò avvenga entro il tempo ***timeout***. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.
`int ata_drq (int drive, clock_t timeout);`	Attende che l'unità ***drive*** sia pronta a ricevere dati, purché ciò avvenga entro il tempo ***timeout***. Se l'operazione ha successo, la funzione restituisce zero, altrimenti dà un valore negativo.

Funzione	Descrizione
`int ata_lba28 (int drive, unsigned int sector, unsigned char count);`	Invia all'unità ***drive*** la prima parte di un comando, in cui sono contenute le coordinate LBA28.

Funzione	Descrizione
`int ata_read_sector (int drive, unsigned int sector, void *buffer);`	È una macroistruzione che legge dall'unità ***drive***, il settore ***sector***, mettendo il risultato a partire dall'indirizzo di memoria ***buffer***. La macroistruzione si avvale praticamente della funzione ***ata_cmd_read_sectors()***, per leggere un solo settore.
`int ata_write_sector (int drive, unsigned int sector, void *buffer);`	È una macroistruzione che scrive nell'unità ***drive***, il settore ***sector***, traendo i dati dall'indirizzo di memoria ***buffer***. La macroistruzione si avvale praticamente della funzione ***ata_cmd_write_sectors()***, per scrivere un solo settore.

84.8 Gestione del file system

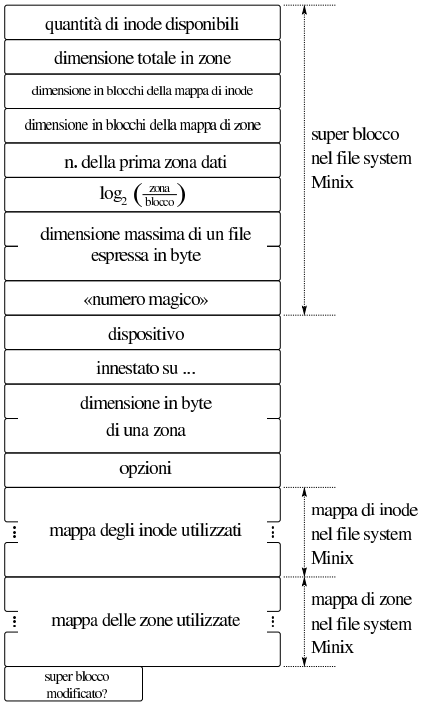
La gestione del file system è suddivisa in diversi file contenuti nella directory '`kernel/fs/`', facenti capo al file di intestazione '`kernel/fs.h`'.

Listato 94.5 e successivi.

84.8.1 File «kernel/fs/sb_...»

I file '`kernel/fs/sb_...`' descrivono le funzioni per la gestione dei super blocchi, distinguibili perché iniziano tutte con il prefisso '**`sb_`**'. Tra questi file si dichiara l'array ***sb_table[]***, il quale rappresenta una tabella le cui righe sono rappresentate da elementi di tipo '**`sb_t`**' (il tipo '**`sb_t`**' è definito nel file '`kernel/fs.h`'). Per uniformare l'accesso alla tabella, la funzione ***sb_reference()*** permette di ottenere il puntatore a un elemento dell'array ***sb_table[]***, specificando il numero del dispositivo cercato.

Figura 84.95. Struttura del tipo '**sb_t**', corrispondente agli elementi dell'array ***sb_table[]***.



Listato 84.96. Struttura del tipo '**sb_t**', corrispondente agli elementi dell'array ***sb_table[]***.

```
typedef struct sb        sb_t;

struct sb {
  uint16_t  inodes;
  uint16_t  zones;
  uint16_t  map_inode_blocks;
  uint16_t  map_zone_blocks;
  uint16_t  first_data_zone;
  uint16_t  log2_size_zone;
  uint32_t  max_file_size;
  uint16_t  magic_number;
  //-------------------------
  dev_t     device;
  inode_t  *inode_mounted_on;
  blksize_t blksize;
  int       options;
  uint16_t  map_inode[SB_MAP_INODE_SIZE];
  uint16_t  map_zone[SB_MAP_ZONE_SIZE];
  char      changed;
};
```

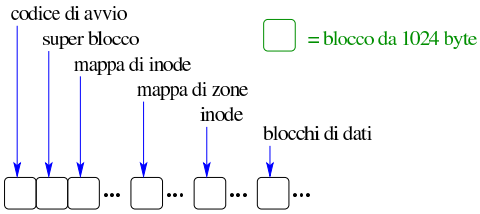
Il super blocco rappresentato dal tipo '**sb_t**' include anche le mappe delle zone e degli inode impegnati. Queste mappe hanno una dimensione fissa in memoria, mentre nel file system reale possono essere di dimensione minore. La tabella di super blocchi, contiene le informazioni dei dispositivi di memorizzazione innestati nel sistema. L'innesto si concretizza nel riferimento a un inode, contenuto nella tabella degli inode (descritta in un altro capitolo), il quale rappresenta la directory di un'altra unità, su cui tale innesto è avvenuto. Naturalmente, l'innesto del file system principale rappresenta un caso particolare.

Tabella 84.97. Funzioni per la gestione dei dispositivi di memorizzazione di massa, a livello di super blocco, definite nei file '`kernel/fs/sb_...`'.

Funzione	Descrizione
`sb_t *sb_reference (dev_t device);`	Restituisce il riferimento a un elemento della tabella dei super blocchi, in base al numero del dispositivo di memorizzazione. Se il dispositivo cercato non risulta già innestato, si ottiene il puntatore nullo; se si chiede il dispositivo zero, si ottiene il puntatore al primo elemento della tabella.
`sb_t *sb_mount (dev_t device, inode_t **inode_mnt, int options);`	Innesta il dispositivo rappresentato numericamente dal primo parametro, sulla directory corrispondente all'inode a cui punta il secondo parametro, con le opzioni del terzo parametro. Quando si tratta del primo innesto del file system principale, la directory è quella dello stesso file system, pertanto, in tal caso, ****inode_mnt*** è inizialmente un puntatore nullo e deve essere modificato dalla funzione stessa.
`int sb_save (sb_t *sb);`	Salva il super blocco nella sua unità di memorizzazione, se questo risulta modificato. In questo caso, il super blocco include anche le mappe degli inode e delle zone.
`int sb_zone_status (sb_t *sb, zno_t zone);`	Restituisce uno se la zona rappresentata dal secondo parametro è impegnata nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.
`int sb_inode_status (sb_t *sb, ino_t ino);`	Restituisce uno se l'inode rappresentato dal secondo parametro è impegnato nel super blocco a cui si riferisce il primo parametro; diversamente restituisce zero.
`void sb_print (void);`	Funzione diagnostica per la visualizzazione sullo schermo dello stato della tabella dei super blocchi.

84.8.2 File «kernel/fs/zone_...»

Nel file system Minix 1, si distinguono i concetti di blocco e zona di dati, con il vincolo che la zona ha una dimensione multipla del blocco. Il contenuto del file system, dopo tutte le informazioni amministrative, è organizzato in zone; in altri termini, i blocchi di dati si raggiungono in qualità di zone.



La zona rimane comunque un tipo di blocco, potenzialmente più grande (ma sempre multiplo) del blocco vero e proprio, che si nu-

mera a partire dall'inizio dello spazio disponibile, con la differenza che è utile solo per raggiungere i blocchi di dati. Nel super blocco del file system si trova l'informazione del numero della prima zona che contiene dati, in modo da non dover ricalcolare questa informazione ogni volta.

I file '`kernel/fs/zone_...`' descrivono le funzioni per la gestione del file system a zone.

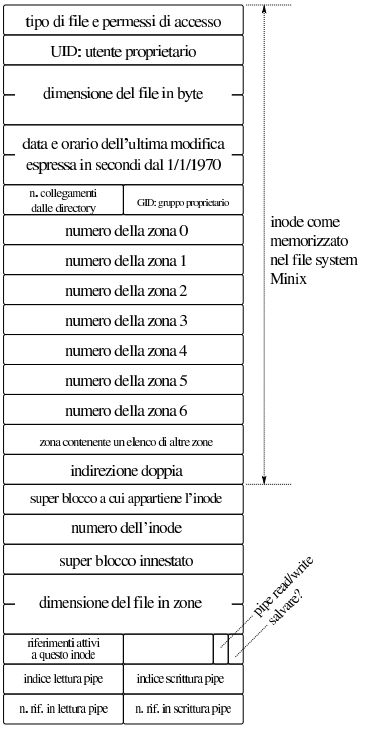
Tabella 84.99. Funzioni per la gestione delle zone, definite nei file '`kernel/fs/zone_...`'.

Funzione	Descrizione
`zno_t zone_alloc (sb_t *sb);`	Alloca una zona, restituendo il numero della stessa. In pratica, cerca la prima zona libera nel file system a cui si riferisce il super blocco ***sb** e la segna come impegnata, restituendone il numero.
`int zone_free (sb_t *sb, zno_t zone);`	Libera una zona, impegnata precedentemente.
`int zone_read (sb_t *sb, zno_t zone, void *buffer);`	Legge il contenuto di una zona, memorizzandolo a partire dalla posizione di memoria rappresentato da ***buffer***.
`int zone_write (sb_t *sb, zno_t zone, void *buffer);`	Sovrascrive una zona, utilizzando il contenuto della memoria a partire dalla posizione rappresentata da ***buffer***.
`void zone_print (sb_t *sb, zno_t zone);`	Funzione diagnostica per la visualizzazione dello stato di una zona.

84.8.3 File «kernel/fs/inode_...»

I file '`kernel/fs/inode_...`' descrivono le funzioni per la gestione dei file, in forma di inode. In uno di questi file viene dichiarata la tabella degli inode in uso nel sistema, rappresentata dall'array ***inode_table[]*** e per individuare un certo elemento dell'array si usa preferibilmente la funzione ***inode_reference()***. Gli elementi della tabella degli inode sono di tipo '**`inode_t`**' (definito nel file '`kernel/fs.h`'); una voce della tabella rappresenta un inode utilizzato se il campo dei riferimenti (***references***) ha un valore maggiore di zero.

Figura 84.100. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.



Listato 84.101. Struttura del tipo '**`inode_t`**', corrispondente agli elementi dell'array ***inode_table[]***.

```
typedef struct inode     inode_t;

struct inode {
    mode_t        mode;
    uid_t         uid;
    ssize_t       size;
    time_t        time;
    uint8_t       gid;
    uint8_t       links;
    zno_t         direct[7];
    zno_t         indirect1;
    zno_t         indirect2;
    //------------------------
    sb_t         *sb;
    ino_t         ino;
    sb_t         *sb_attached;
    blkcnt_t      blkcnt;
    unsigned char references;
    char          changed  : 1,
                  pipe_dir : 1;
    unsigned char pipe_off_read;
    unsigned char pipe_off_write;
    unsigned char pipe_ref_read;
    unsigned char pipe_ref_write;
};
```

Figura 84.102. Collegamento tra la tabella degli inode e quella dei super blocchi.

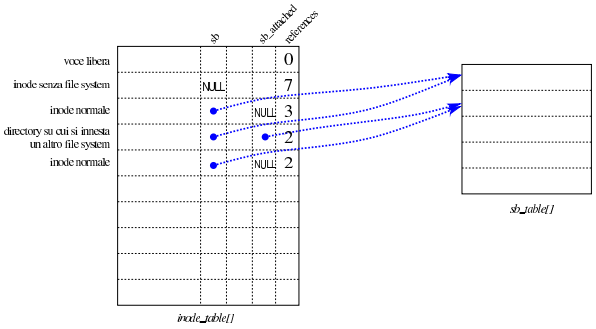


Tabella 84.103. Funzioni per la gestione dei file in forma di inode, definite nei file '`kernel/fs/inode_...`'.

Funzione	Descrizione
`inode_t *inode_reference (dev_t device, ino_t ino);`	Restituisce il puntatore a un inode, rappresentato in pratica da un elemento dell'array ***inode_table[]***, corrispondente a quello con il numero di dispositivo e di inode indicati come argomenti. Se entrambi gli argomenti sono a zero, si ottiene il puntatore al primo elemento; se entrambi i valori sono pari a `-1`, si ottiene il puntatore al primo elemento libero; se viene indicato il dispositivo zero e l'inode numero uno, si ottiene il puntatore all'elemento corrispondente alla directory radice del file system principale.
`inode_t *inode_alloc (dev_t device, mode_t mode, uid_t uid gid_t gid);`	La funzione ***inode_alloc()*** cerca un inode libero nel file system del dispositivo indicato, quindi lo alloca (lo segna come utilizzato) e lo modifica aggiornando il tipo e la modalità dei permessi, oltre al proprietario del file e al gruppo. Se la funzione riesce nel suo intento, restituisce il puntatore all'inode in memoria, il quale rimane così aperto e disponibile per ulteriori elaborazioni.
`int inode_free (inode_t *inode);`	Segna l'inode indicato come libero.
`inode_t *inode_get (dev_t device, ino_t ino);`	Restituisce il puntatore all'inode rappresentato dal numero di dispositivo e di inode, indicati come argomenti. Se l'inode è già presente nella tabella degli inode, la cosa si risolve nell'incremento di una unità del numero dei riferimenti di tale inode; se invece l'inode non è ancora presente, questo viene caricato dal suo file system nella tabella e gli viene attribuito inizialmente un riferimento attivo.

Funzione	Descrizione
`int inode_put (inode_t *inode);`	Rilascia un inode che non serve più. Ciò comporta la riduzione del contatore dei riferimenti nella tabella degli inode, tenendo conto che se tale valore raggiunge lo zero, si provvede anche al suo salvataggio nel file system (ammesso che l'inode della tabella risulti modificato, rispetto alla versione presente nel file system). La funzione restituisce zero in caso di successo, oppure `-1` in caso contrario.
`int inode_save (inode_t *inode);`	Salva l'inode nel file system, se questo risulta modificato.
`int inode_truncate (inode_t *inode);`	Riduce la dimensione del file a cui si riferisce l'inode a zero. In pratica fa sì che le zone allocate del file siano liberate. La funzione restituisce zero se l'operazione si conclude con successo, oppure `-1` in caso di problemi.
`zno_t inode_zone (inode_t *inode, zno_t fzone, int write);`	Restituisce il numero di zona effettivo, corrispondente a un numero di zona relativo a un certo file di un certo inode. Se il parametro ***write*** è pari a zero, si intende che la zona deve esistere, quindi se questa non c'è, si ottiene semplicemente un valore pari a zero; se invece l'ultimo parametro è pari a uno, nel caso la zona cercata fosse attualmente mancante, verrebbe creata al volo nel file system.
`inode_t * inode_stdio_dev_make (dev_t device, mode_t mode);`	Alloca un inode in memoria, riferito al dispositivo richiesto dal primo parametro, con i permessi del secondo parametro. Tale inode è adatto per essere utilizzato come flusso standard (standard input, standard output o standard error). Questa funzione viene usata solo da ***file_stdio_dev_make()***, la quale, a sua volta, viene usata solo da ***proc_sys_exec()***.
`blkcnt_t inode_fzones_read (inode_t *inode, zno_t zone_start, void *buffer, blkcnt_t blkcnt);`	Legge da un file, identificato attraverso il puntatore all'inode (della tabella di inode), una certa quantità di zone, a partire da una certa zona relativa al file, mettendo il risultato della lettura a partire dalla posizione di memoria rappresentata da un puntatore generico. La funzione restituisce la quantità di zone lette con successo.

Funzione	Descrizione
`blkcnt_t inode_fzones_write (inode_t *inode, zno_t zone_start, void *buffer, blkcnt_t blkcnt);`	Svolge il compito opposto della funzione ***inode_fzones_read()*** e attualmente non viene utilizzata.
`ssize_t inode_file_read (inode_t *inode, off_t offset, void *buffer, size_t count, int *eof );`	Legge il contenuto di un file, individuato da un inode già caricato nella tabella relativa, aggiornando eventualmente una variabile contenente l'indicatore di fine file. La funzione restituisce la quantità di byte letti con successo, oppure il valore -1 in caso di problemi.
`ssize_t inode_file_write (inode_t *inode, off_t offset, void *buffer, size_t count);`	Scrive una certa quantità di byte nel file individuato da un inode già caricato nella tabella relativa. La funzione restituisce la quantità di byte scritti effettivamente, oppure il valore -1 in caso di problemi.
`int inode_check (inode_t *inode, mode_t type, int perm, uid_t uid gid_t gid);`	Verifica che l'inode sia di un certo tipo e abbia i permessi di accesso necessari a un certo utente e gruppo. Nel parametro ***type*** si possono indicare più tipi validi. La funzione restituisce zero in caso di successo, ovvero di compatibilità, mentre restituisce -1 se il tipo o i permessi non sono adatti.
`int inode_dir_empty (inode_t *inode);`	Verifica se la directory a cui si riferisce l'inode è effettivamente una directory ed è vuota, nel qual caso restituisce il valore uno, altrimenti restituisce zero.
`inode_t *inode_pipe_make (void);`	Crea un condotto senza nome (*pipe*), restituendo il puntatore all'inode relativo.
`ssize_t inode_pipe_read (inode_t *inode, void *buffer, size_t count, int *eof );`	Legge il contenuto di un condotto, individuato da un inode già caricato nella tabella relativa, aggiornando eventualmente una variabile contenente l'indicatore di fine file. La funzione restituisce la quantità di byte letti con successo, oppure il valore -1 in caso di problemi.
`ssize_t inode_pipe_write (inode_t *inode, void *buffer, size_t count);`	Scrive una certa quantità di byte nel condotto individuato da un inode già caricato nella tabella relativa. La funzione restituisce la quantità di byte scritti effettivamente, oppure il valore -1 in caso di problemi.
`void inode_print (void);`	Funzione diagnostica per la visualizzazione sintetica del contenuto della tabella degli inode.

84.8.4 Fasi dell'innesto di un file system

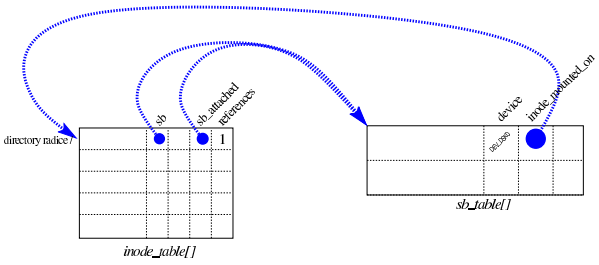
L'innesto e il distacco di un file system, coinvolge simultaneamente la tabella dei super blocchi e quella degli inode. Si distinguono due situazioni fondamentali: l'innesto del file system principale e quello di un file system ulteriore.

Quando si tratta dell'innesto del file system principale, la tabella dei super blocchi è priva di voci e quella degli inode non contiene riferimenti a file system. La funzione ***sb_mount()*** viene chiamata indicando, come riferimento all'inode di innesto, il puntatore a una variabile puntatore contenente il valore nullo:

```
...
    inode_t *inode;
    sb_t    *sb;
    ...
    inode = NULL;
    sb = sb_mount (DEV_DSK0, &inode, MOUNT_DEFAULT);
    ...
```

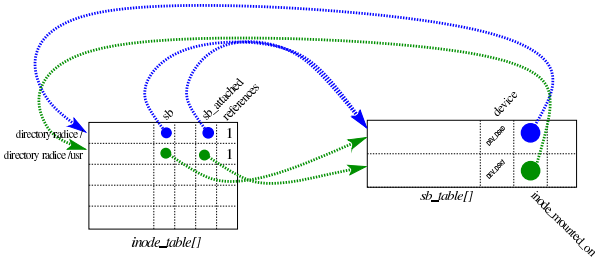
La funzione ***sb_mount()*** carica il super blocco nella tabella relativa, ma trovando il riferimento all'inode di innesto nullo, provvede a caricare l'inode della directory radice dello stesso dispositivo, creando un collegamento incrociato tra le tabelle dei super blocchi e degli inode, come si vede nella figura successiva.

Figura 84.105. Collegamento tra la tabella degli inode e quella dei super blocchi, quando si innesta il file system principale.



Per innestare un altro file system, occorre prima disporre dell'inode di una directory (appropriata) nella tabella degli inode, quindi si può caricare il super blocco del nuovo file system, creando il collegamento tra directory e file system innestato.

Figura 84.106. Innesto di un file system nella directory '`/usr/`'.

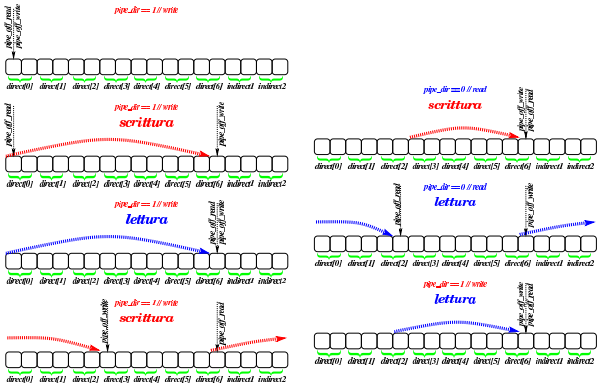


84.8.5 Condotti

I condotti sono gestiti da os32 nel modo tradizionale, sfruttando nell'inode la poca memoria che altrimenti servirebbe per i riferimenti ai blocchi di dati. In pratica, secondo la struttura del tipo '**`inode_t`**', si usa ***direct[]***, ***indirect1*** e ***indirect2***. Ciò comporta complessivamente la disponibilità di soli 18 byte, cosa che comunque sarebbe insufficiente per lo standard attuale dei sistemi Unix.

Lo spazio di questi 18 byte viene trattato come una coda e scandito attraverso due indici: quello di scrittura e quello di lettura. Durante l'accesso all'inode che rappresenta un condotto si distinguono due stati, individuati dal bit ***pipe_dir***.

Figura 84.107. Esempio di utilizzo di un condotto, attraverso varie fasi di scrittura e lettura.



La figura mostra un esempio che dovrebbe chiarire il meccanismo di funzionamento del condotto.

1. Inizialmente gli indici di scrittura e lettura si trovano ad avere lo stesso valore (nella figura si trovano nella posizione zero, ma qualunque altra posizione sarebbe equivalente), mentre il bit ***pipe_dir*** indica «scrittura». In questa situazione, si deve procedere con la scrittura, durante la quale l'indice di scrittura non può superare nuovamente quello di lettura.
2. Nel secondo disegno della figura si vede che è avvenuta una scrittura che ha occupato 13 byte, mentre l'indice di scrittura si trova sul quattordicesimo (quello successivo all'ultimo byte scritto).
3. A questo punto, si suppone che inizi la lettura del condotto: in tal caso, dato che il bit ***pipe_dir*** indica ancora «scrittura», la lettura non può superare la posizione che ha raggiunto l'indice di scrittura. Pertanto si suppone che la lettura raccolga la stessa quantità di byte occupati precedentemente dalla scrittura. Quando l'indice di lettura incontra quello di scrittura, il bit ***pipe_dir*** deve essere impostato a «scrittura», perché non c'è altro che si possa leggere. Tale bit era già impostato nel modo corretto e quindi non si notano variazioni.
4. Nel quarto disegno si vede l'inizio di una nuova fase di scrittura che raggiunge la fine dello spazio dei 18 byte previsti per riprendere dall'inizio. In questa fase di scrittura gli indici non si incontrano e nulla cambia nello stato di ***pipe_dir***.
5. Nel quinto disegno (all'inizio del lato destro), la scrittura riprende e raggiunge l'indice di lettura (che non può essere superato). Qui lo stato rappresentato da ***pipe_dir*** cambia, dal momento che non si può più procedere con la scrittura, adesso indica «lettura».
6. Nel sesto disegno si vede l'inizio di una lettura. Dopo di questa lettura, potrebbe esserci una fase di scrittura, ma senza poter superare l'indice di lettura. Comunque, questa scrittura non viene eseguita.
7. Nell'ultimo disegno si vede che la lettura continua, fino a raggiungere l'indice di scrittura, quando così il valore di ***pipe_dir*** viene invertito nuovamente.

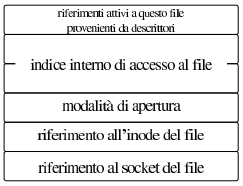
In pratica, quando gli indici di lettura e scrittura coincidono, per sapere se si può procedere con una scrittura o una lettura, occorre chiederlo a ***pipe_dir***; diversamente, con indici diversi, la scrittura o la lettura può procedere indifferentemente, ma solo fino al raggiungimento dell'altro indice. Poi, se è l'indice di lettura che ha appena raggiunto quello di scrittura, ***pipe_dir*** deve essere impostato per richiedere la scrittura; al contrario, quando è l'indice di scrittura che raggiunge quello di lettura, ***pipe_dir*** deve richiedere la lettura successiva.

84.8.6 File «kernel/fs/file_...»

I file '`kernel/fs/file_...`' descrivono le funzioni per la gestione della tabella dei file, la quale si collega a sua volta a quella degli inode. In realtà, le funzioni di questo gruppo sono in numero molto limitato, perché l'intervento nella tabella dei file avviene prevalentemente per opera di funzioni che gestiscono i descrittori.

La tabella dei file è rappresentata dall'array ***file_table[]*** e per individuare un certo elemento dell'array si usa preferibilmente la funzione ***file_reference()***. Gli elementi della tabella dei file sono di tipo '`file_t`' (definito nel file '`kernel/fs.h`'); una voce della tabella rappresenta un file aperto se il campo dei riferimenti (***references***) ha un valore maggiore di zero.

Figura 84.108. Struttura del tipo '`file_t`', corrispondente agli elementi dell'array ***file_table[]***.




```
typedef struct file file_t;

struct file {
    int       references;
    off_t     offset;
    int       oflags;
    inode_t  *inode;
    sock_t   *sock;
};
```

Nel membro ***oflags*** si annotano esclusivamente opzioni relative alla modalità di apertura del file: lettura, scrittura o entrambe; pertanto si possono usare le macro-variabili ***O_RDONLY***, ***O_WRONLY*** e ***O_RDWR***, come dichiarato nel file di intestazione '`lib/fcntl.h`'. Il membro ***offset*** rappresenta l'indice interno di accesso al file, per l'operazione successiva di lettura o scrittura al suo interno. Il membro ***references*** è un contatore dei riferimenti a questa tabella, da parte di descrittori di file.

La tabella dei file si collega a quella degli inode, attraverso il membro ***inode***, oppure a quella dei socket, attraverso il membro ***sock***. Più voci della tabella dei file possono riferirsi allo stesso inode (o allo stesso socket), perché hanno modalità di accesso differenti, oppure soltanto per poter distinguere l'indice interno di lettura e scrittura. Va osservato che le voci della tabella di inode potrebbero essere usate direttamente e non avere elementi corrispondenti nella tabella dei file.

Figura 84.109. Collegamento tra la tabella dei file e quella degli inode.

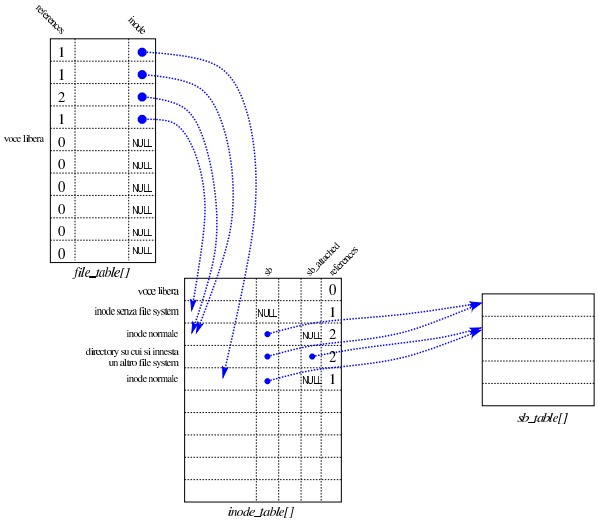


Tabella 84.110. Funzioni fatte esclusivamente per la gestione della tabella dei file ***file_table[]***.

Funzione	Descrizione
`file_t *file_reference (int fno);`	Restituisce il puntatore all'elemento ***fno***-esimo della tabella dei file. Se ***fno*** è un valore negativo, viene restituito il puntatore a una voce libera della tabella.
`file_t *file_stdio_dev_make (dev_t device, mode_t mode, int oflags);`	Crea una voce per l'accesso a un file di dispositivo standard di input-output, restituendo il puntatore alla voce stessa.

84.8.7 Descrittori di file

Le tabelle di super blocchi, inode e file, riguardano il sistema nel complesso. Tuttavia, l'accesso normale ai file avviene attraverso il concetto di «descrittore», il quale è un file aperto da un certo processo elaborativo. Nel file '`kernel/fs.h`' si trova la dichiarazione e descrizione del tipo derivato '**`fd_t`**', usato per costruire una tabella di descrittori, ma tale tabella non fa parte della gestione del file system, bensì è incorporata nella tabella dei processi elaborativi. Pertanto, ogni processo ha una propria tabella di descrittori di file.

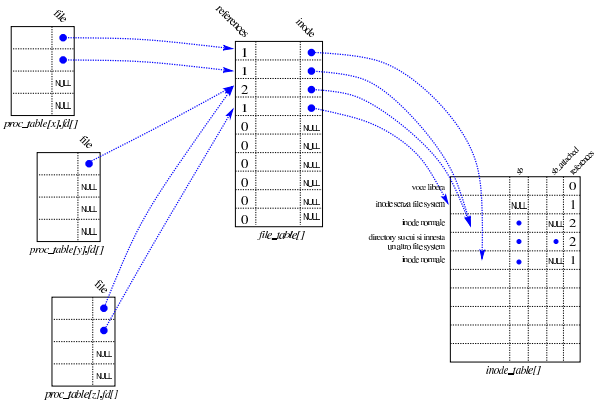
Figura 84.111. Struttura del tipo '**`fd_t`**', con cui si costituiscono gli elementi delle tabelle dei descrittori di file, una per ogni processo.


```
typedef struct fd fd_t;
struct fd {
    int     fl_flags;
    int     fd_flags;
    file_t *file;
};
```

Il membro ***fl_flags*** consente di annotare indicatori del tipo '**`O_RDONLY`**', '**`O_WRONLY`**', '**`O_RDWR`**', '**`O_CREAT`**', '**`O_EXCL`**', '**`O_NOCTTY`**', '**`O_TRUNC`**' e '**`O_APPEND`**', come dichiarato nella libreria standard, nel file di intestazione '`lib/fcntl.h`'. Tali indicatori si combinano assieme con l'operatore binario OR. Altri tipi di opzione che sarebbero previsti nel file '`lib/fcntl.h`', sono privi di effetto nella gestione del file system di os16.

Il membro ***fd_flags*** serve a contenere, eventualmente, l'opzione '**`FD_CLOEXEC`**', definita nel file '`lib/fcntl.h`'. Non sono previste altre opzioni di questo tipo.

Figura 84.112. Collegamento tra le tabelle dei descrittori e la tabella complessiva dei file. La tabella ***proc_table[x].fd[]*** rappresenta i descrittori di file del processo elaborativo ***x***.



84.8.8 File «kernel/fs/path_...»

I file '`kernel/fs/path_...`' descrivono le funzioni che fanno riferimento a file o directory attraverso una stringa che ne descrive il percorso.

Tabella 84.113. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, senza indicazioni sul processo elaborativo.

Funzione	Descrizione
`int path_fix (char *path);`	Verifica il percorso indicato semplificandolo, quindi sovrascrive il percorso originario con quello riveduto e corretto. Un percorso assoluto rimane assoluto; un percorso relativo rimane relativo, mancando qualunque indicazione sulla directory corrente.
`int path_full (const char *path, const char *path_cwd, char *full_path);`	Ricostruisce un percorso assoluto, usando come riferimento la directory corrente indicata in ***path_cwd***, salvandolo in ***path_full***.

Tabella 84.114. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione. Del processo elaborativo si considera soprattutto l'identità efficace, per conoscerne i privilegi e determinare se è data effettivamente la facoltà di eseguire l'azione richiesta.

Funzione	Descrizione
`inode_t *path_inode (pid_t pid, const char *path);`	Apre l'inode del file indicato tramite il percorso, purché il processo ***pid*** abbia i permessi di accesso («x») alle directory che vi conducono. La funzione restituisce il puntatore all'inode aperto, oppure il puntatore nullo se non può eseguire l'operazione.
`dev_t path_device (pid_t pid, const char *path);`	Restituisce il numero del dispositivo di un file di dispositivo; pertanto, il percorso deve fare riferimento a un file di dispositivo, per poter ottenere un risultato valido.
`inode_t *path_inode_link (pid_t pid, const char *path, inode_t *inode, mode_t mode);`	Crea un collegamento fisico con il nome fornito in ***path***, riferito all'inode a cui punta ***inode***, ma se ***inode*** fosse un puntatore nullo, verrebbe semplicemente creato un file vuoto con un nuovo inode. Si richiede inoltre che il processo ***pid*** abbia i permessi di accesso per tutte le directory che portano al file da collegare e che nell'ultima ci sia anche il permesso di scrittura, dovendo intervenire su tale directory in questo modo. Se la funzione riesce nel proprio intento, restituisce il puntatore a ciò che descrive l'inode collegato o creato.

Delle funzioni che, per affinità, farebbero parte di questo gruppo, si trovano nella directory '`kernel/lib_s/`', in quanto servono per attuare delle chiamate di sistema.

Tabella 84.115. Funzioni per la gestione dei file, a cui si fa riferimento attraverso un percorso, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione, contenute nella directory '`kernel/lib_s/`'.

Funzione	Descrizione
`int s_chdir (pid_t pid, const char *path);`	Cambia la directory corrente, utilizzando il nuovo percorso indicato. È l'equivalente della funzione standard ***chdir()*** (sezione 87.6).
`int s_chmod (pid_t pid, const char *path, mode_t mode);`	Cambia la modalità di accesso al file indicato. È l'equivalente della funzione standard ***chmod()*** (sezione 87.7).
`int s_chown (pid_t pid, const char *path, uid_t uid, gid_t gid);`	Cambia l'utente e il gruppo proprietari del file. È l'equivalente della funzione standard ***chown()*** (sezione 87.8).
`int s_link (pid_t pid, const char *path_old, const char *path_new);`	Crea un collegamento fisico. È l'equivalente della funzione standard ***link()*** (sezione 87.30).
`int s_mkdir (pid_t pid, const char *path, mode_t mode);`	Crea una directory, con la modalità dei permessi indicata. È l'equivalente della funzione standard ***mkdir()*** (sezione 87.34).
`int s_mknod (pid_t pid, const char *path, mode_t mode, dev_t device);`	Crea un file vuoto, con il tipo e i permessi specificati da ***mode***; se si tratta di un file di dispositivo, viene preso in considerazione anche il parametro ***device***, per specificare il numero primario e secondario dello stesso. Va osservato che con questa funzione è possibile creare una directory priva delle voci '`.`' e '`..`'. È l'equivalente della funzione standard ***mknod()*** (sezione 87.35).
`int s_mount (pid_t pid, const char *path_dev, const char *path_mnt, int options);`	Innesta il dispositivo corrispondente a ***path_dev***, nella directory ***path_mnt*** (tenendo conto della directory corrente del processo ***pid***), con le opzioni specificate. Le opzioni disponibili sono solo '**`MOUNT_DEFAULT`**' e '**`MOUNT_RO`**', come dichiarato nel file di intestazione '`lib/sys/os32.h`'.
`int s_open (pid_t pid, const char *path, int oflags, mode_t mode);`	Apre un descrittore, fornendo però il percorso del file. È l'equivalente della funzione standard ***open()*** (sezione 87.37).
`int s_stat (pid_t pid, const char *path, struct stat *buffer);`	Aggiorna la variabile strutturata a cui punta ***buffer***, con le informazioni sul file specificato. È l'equivalente della funzione standard ***stat()*** (sezione 87.55).
`int s_umount (pid_t pid, const char *path_mnt);`	Stacca l'unità innestata nella directory indicata, purché nulla al suo interno sia attualmente in uso.

Funzione	Descrizione
`int s_unlink (pid_t pid, const char *path);`	Cancella un file o una directory, purché questa sia vuota. È l'equivalente della funzione standard ***unlink()*** (sezione 87.62).

84.8.9 File «kernel/fs/fd_...»

I file '`kernel/fs/fd_...`' descrivono le funzioni che fanno riferimento a file o directory attraverso il numero di descrittore, riferito a sua volta a un certo processo elaborativo. Pertanto, il numero del processo e il numero del descrittore sono i primi due parametri obbligatori di tutte queste funzioni.

Tabella 84.116. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, relativamente a un certo processo elaborativo, le quali non rappresentano direttamente la realizzazione di una chiamata di sistema.

Funzione	Descrizione
`fd_t *fd_reference (pid_t pid, int *fdn);`	Produce il puntatore ai dati del descrittore ****fdn***. Se ****fdn*** è minore di zero, si ottiene il riferimento al primo descrittore libero, aggiornando anche ****fdn*** stesso.
`int fd_dup (pid_t pid, int fdn_old, int fdn_min);`	Duplica il descrittore ***fdn_old***, creandone un altro con numero maggiore o uguale a ***fdn_min*** (viene scelto il primo libero a partire da ***fdn_num***).

Delle funzioni che, per affinità, farebbero parte di questo gruppo, si trovano nella directory '`kernel/lib_s/`', in quanto servono per attuare delle chiamate di sistema.

Tabella 84.117. Funzioni per la gestione dei file, a cui si fa riferimento attraverso il descrittore, tenendo conto del processo elaborativo per conto del quale si svolge l'operazione, contenute nella directory '`kernel/lib_s/`'.

Funzione	Descrizione
`int s_dup (pid_t pid, int fdn_old);`	Duplica il descrittore ***fdn_old***, creandone un altro (utilizzando il primo numero di descrittore libero) il cui numero viene restituito dalla funzione. È l'equivalente della funzione standard ***dup()*** (sezione 87.12).
`int s_dup2 (pid_t pid, int fdn_old, int fdn_new);`	Duplica il descrittore ***s_old***, creandone un altro con numero ***fdn_new***. Se però ***fdn_new*** è già aperto, prima della duplicazione questo viene chiuso. È l'equivalente della funzione standard ***dup2()*** (sezione 87.12).
`int s_fchmod (pid_t pid, int fdn, mode_t mode);`	Cambia la modalità dei permessi (solo gli ultimi 12 bit del parametro ***mode*** vengono considerati). È l'equivalente della funzione standard ***fchmod()*** (sezione 87.7).
`int s_fchown (pid_t pid, int fdn, uid_t uid, gid_t gid);`	Cambia la proprietà (utente e gruppo). È l'equivalente della funzione standard ***fchown()*** (sezione 87.8).

Funzione	Descrizione
`int s_fcntl (pid_t pid, int fdn, int cmd, int arg);`	Svolge il compito della funzione standard ***fcntl()*** (sezione 87.18).
`int s_fstat (pid_t pid, int fdn, struct stat *buffer);`	Svolge il compito della funzione standard ***fstat()*** (sezione 87.55).
`off_t s_lseek (pid_t pid, int fdn, off_t offset, int whence);`	Riposiziona l'indice interno di accesso del descrittore di file. È l'equivalente della funzione standard ***lseek()*** (sezione 87.33).
`int s_pipe (pid_t pid, int pipefd[2]);`	Crea un condotto senza nome (*pipe*), per il quale, i due descrittori necessari vengono salvati in ***pipefd[]***. Per la precisione, ***pipefd[0]*** individua il descrittore del lato di lettura, mentre ***pipefd[1]*** individua quello del lato di scrittura. È l'equivalente della funzione standard ***pipe()*** (sezione 87.38).
`ssize_t s_read (pid_t pid, int fdn, void *buffer, size_t count, int *eof);`	Legge da un descrittore, aggiornando eventualmente la variabile ****eof*** in caso di fine del file. È l'equivalente della funzione standard ***read()*** (sezione 87.39).
`ssize_t s_write (pid_t pid, int fdn, const void *buffer, size_t count);`	Scrive nel descrittore. È l'equivalente della funzione standard ***write()*** (sezione 87.64).

84.9 Gestione delle interfacce di rete

Il sistema os32 può gestire soltanto interfacce di rete Ethernet e l'interfaccia virtuale locale, nota con il nome *loopback*. Le interfacce di rete hanno tutte nomi del tipo '**`net`*n***', dove ***n*** è un numero intero, a partire da zero, e di norma l'interfaccia virtuale locale coincide con il nome '**`net0`**'.

84.9.1 Gestione dei dispositivi NE2K

Il kernel di os32 è in grado di gestire soltanto le interfacce di rete Ethernet NE2000, collocate nel bus PCI: NE2K. Ciò consente di conoscere la porta di I/O necessaria per accedervi, in modo automatico. Le funzioni per la gestione di queste interfacce sono contenute nei file della directory '`kernel/driver/nic/ne2k/`' e fanno capo al file di intestazione '`kernel/driver/nic/ne2k.h`' (listato 94.4.19 e successivi).

Le interfacce di rete NE2000 dispongono di una piccola memoria tampone interna per la ricezione; tuttavia, appena viene individuato un pacchetto ricevuto, os32 lo trasferisce immediatamente in una propria memoria tampone, contenuta nella tabella delle interfacce, descritta nella sezione successiva.

Per una maggiore semplicità progettuale, la trasmissione di un pacchetto avviene mettendo tutto il sistema in attesa, fino a che l'interfaccia dà un responso, positivo o negativo che sia. Tuttavia, ciò comporta anche il rischio di bloccare definitivamente il sistema, nel caso si dovessero manifestare dei problemi all'interfaccia.

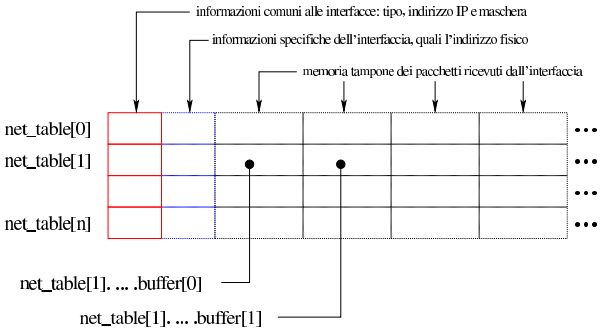
Tabella 84.118. Funzioni per la gestione dell'interfaccia di rete NE2000.

Funzione	Descrizione
`int ne2k_check (uintptr_t io);`	Verifica se l'interfaccia corrispondente alla porta di I/O specificata è veramente di tipo NE2000 [94.4.20].
`int ne2k_isr (uintptr_t io);`	Verifica lo stato dell'interfaccia e ne acquisisce i dati, se disponibili. In caso di ricezione di una trama, viene chiamata la funzione ***ne2k_rx()*** per trasferirla nella memoria tampone della tabella delle interfacce [94.4.21].
`int ne2k_isr_expect (uintptr_t io, unsigned int isr_expect);`	Rimane in attesa fino a che il registro ***ISR*** dell'interfaccia si attiva almeno un indicatore corrispondente a quanto richiesto con il parametro ***isr_expect***. Questa funzione viene usata, in particolare, nella trasmissione dei pacchetti, per i quali occorre verificare quando l'interfaccia ha completato il procedimento [94.4.22].
`int ne2k_reset (uintptr_t io, void *address);`	Azzera l'interfaccia e ne estrae l'indirizzo fisico, collocandolo in corrispondenza di ****address*** [94.4.23].
`int ne2k_rx (uintptr_t io);`	Copia tutte le trame accumulate nella memoria tampone interna dell'interfaccia, in quella della tabella delle interfacce [94.4.24].
`int ne2k_rx_reset (uintptr_t io);`	Reinizializza il processo di ricezione [94.4.25].
`int ne2k_tx (uintptr_t io, void *buffer, size_t size);`	Trasmette una trama Ethernet, contenuta all'interno di ****buffer***, della lunghezza specificata da ***size***. La funzione attende il completamento dell'operazione, prima di concludere il proprio funzionamento [94.4.26].

84.9.2 Tabella delle interfacce e funzioni accessorie

Nei file '`kernel/net/net_public.c`' [94.12.31] e '`kernel/net.h`' [94.12] viene dichiarata la tabella delle interfacce, corrispondente all'array ***net_table[]***, con lo scopo di contenere la memoria tampone delle trame ricevute da ogni interfaccia. La struttura della tabella è definita dal tipo '**`net_t`**' e appare semplificata nella figura successiva.

Figura 84.119. Struttura semplificata della tabella delle interfacce.



Listato 84.120. Struttura di ogni elemento della tabella delle interfacce; i dettagli dei membri ***buffer*** non sono evidenziati, ma contengono sempre, a loro volta, i membri ***clock*** e ***size***

```
typedef struct {
  clock_t              clock;
  size_t               size;
  …
} net_buffer_eth_t net_buffer_lo_t;

typedef struct {
  unsigned int type;
  h_addr_t     ip;      // IPv4 address in host byte order.
  uint8_t      m;       // Short netmask.
  union {
    //
    // Ethernet type data:
    //
    struct {
      uint8_t              mac[6];
      uintptr_t            base_io;
      unsigned char        irq;
      net_buffer_eth_t buffer[NET_MAX_BUFFERS];
    } ethernet;
    //
    // Loopback type data:
    //
    struct {
      net_buffer_lo_t buffer[NET_MAX_BUFFERS];
    } loopback;
  };
} net_t;
```

Ogni pacchetto accumulato nella memoria tampone della tabella delle interfacce, oltre al contenuto del pacchetto, include l'orario in cui questo è stato ricevuto (in unità '**`clock_t`**') e la sua dimensione effettiva.

La scansione della tabella richiede generalmente due indici: il numero che individua l'interfaccia e il numero che rappresenta la trama memorizzata (PDU di livello 2 nel caso di interfaccia Ethernet, oppure di livello 3 nel caso di interfaccia virtuale locale), assieme a delle informazioni accessorie. Per esempio, '**`net_table[0].loopback.buffer[`*`f`*`].clock`**' individua l'orario di ricevimento di un pacchetto con indice ***f*** dell'interfaccia locale '**`net0`**' (*loopback*), mentre '**`net_table[1].ethernet.buffer[`*`f`*`].size`**' individua la dimensione del pacchetto ***f*** dell'interfaccia Ethernet '**`net1`**'.

I pacchetti, a livello della rete fisica, vengono depositati nella memoria tampone della tabella, in corrispondenza dell'interfaccia da cui provengono; da qui, poi, attraverso la funzione ***net_rx()***, i pacchetti vengono passati ai gestori appropriati, cancellandoli dalla tabella originaria.

Tabella 84.121. Funzioni per la gestione della tabella delle interfacce, contenute nella directory '`kernel/net/`', e altre accessorie relative alla gestione Ethernet.

Funzione	Descrizione
`net_buffer_eth_t *` `net_buffer_eth (int n);`	Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia '**`net`*`n`*', purché questa sia di tipo Ethernet [94.12.23].
`net_buffer_lo_t *` `net_buffer_lo (int n);`	Restituisce il puntatore a un elemento libero, o utilizzabile, della memoria tampone dell'interfaccia '**`net`*`n`*', purché questa sia di tipo *loopback*, ossia l'interfaccia locale virtuale [94.12.24].
`int net_index (h_addr_t ip);`	Restituisce l'indice della tabella delle interfacce, corrispondente all'indirizzo IPv4 fornito come argomento [94.12.27].

`int net_index_eth (h_addr_t ip,`
`uint8_t mac[6],`
`uintptr_t io);`
	Restituisce l'indice della tabella delle interfacce, corrispondente a uno dei dati forniti come argomento (i valori nulli vengono ignorati), purché si tratti di un'interfaccia Ethernet [94.12.28].

`void net_init (void);`	Inizializza la gestione della rete, utilizzando le informazioni attraverso le opzioni di avvio per configurare anche le interfacce Ethernet [94.12.29].
`void net_rx (void);`	Scandisce i pacchetti memorizzati nella tabella delle interfacce, passandoli al gestore appropriato e rimuovendoli poi dalla tabella [94.12.32].

`int net_eth_ip_tx (h_addr_t src,`
`h_addr_t dst,`
`const void *packet,`
`size_t size);`
	A partire da un pacchetto IPv4 completo e dagli indirizzi IPv4 di origine e di destinazione, viene assemblata e spedita una trama Ethernet. La funzione richiede separatamente l'indicazione degli indirizzi IPv4 di origine e destinazione, per semplificare il codice, evitando di estrapolarli dal pacchetto IPv4 stesso [94.12.25].

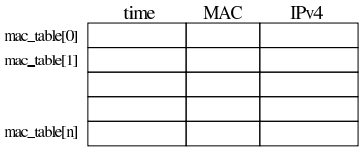
`int net_eth_tx (int n,`
`void *buffer,`
`size_t size);`
	Provvede a trasmettere una trama Ethernet attraverso l'interfaccia ***n*** (ovvero ***net_table[n]***), la quale deve essere di tipo Ethernet [94.12.26].

84.9.3 Tabella ARP

Per mantenere memoria delle corrispondenze tra indirizzi IPv4 e indirizzi Ethernet, si utilizza la tabella ARP, descritta nel file '`kernel/net/arp.h`' [94.12.1] e dichiarata nel file '`kernel/net/arp/arp_public.c`' [94.12.6].

Le voci della tabella sono valide per un tempo limitato, definito dalla macro-variabile ***ARP_MAX_TIME*** e periodicamente vengono scandite e cancellate le voci troppo vecchie.

Figura 84.122. Struttura della tabella ARP, costituita da elementi di tipo '**arp_t**'.




```
typedef struct {
  time_t    time;
  uint8_t   mac[6];
  h_addr_t  ip;
} arp_t;
```

Tabella 84.123. Funzioni per la gestione della tabella ARP, contenute nella directory '`kernel/net/arp/`'.

Funzione	Descrizione
`void arp_init (void);`	Azzera completamente la tabella ARP: si usa una volta sola all'avvio della gestione della rete [94.12.4].
`void arp_clean (void);`	Azzera le voci della tabella ARP che risultano troppo vecchie e che devono essere rinnovate [94.12.2].
`int arp_index (unsigned char mac[6], h_addr_t ip);`	Restituisce l'indice della tabella ARP, corrispondente all'indirizzo Ethernet o all'indirizzo IPv4 fornito [94.12.3].
`arp_t *arp_reference (void);`	Restituisce il puntatore a un elemento della tabella ARP contenente la voce più vecchia, allo scopo presumibile di riutilizzarla per un indirizzo nuovo [94.12.7].
`void arp_request (h_addr_t ip);`	Invia una richiesta ARP, preparando il pacchetto relativo e inviandolo attraverso la funzione ***ethernet_tx()*** [94.12.8].
`int arp_rx (int n, int f);`	Legge dalla tabella delle interfacce il pacchetto individuato dall'indice ***n*** per l'interfaccia e dall'indice ***f*** per la trama relativa. Il pacchetto in questione deve essere relativo al protocollo ARP: se si tratta di una richiesta, provvede a inviare una risposta, se invece si tratta di una risposta, allora aggiorna la tabella ARP [94.12.9].

84.10 Gestione di IPv4

La gestione di IPv4, da parte di os32, è estremamente limitata, per semplificare il codice e la sua comprensione. In particolare non si considerano le opzioni che potrebbero essere contenute tra l'intestazione minima e il contenuto del pacchetto IPv4.

Tabella 84.124. Funzioni per la gestione dei pacchetti a livello IP.

Funzione	Descrizione
`uint16_t ip_checksum (uint16_t *data1, size_t size1, uint16_t *data2, size_t size2);`	Produce il codice di controllo usato nel protocollo IPv4, partendo da due blocchi di dati [94.12.16].
`int ip_rx (int n, int f);`	Si occupa di acquisire un pacchetto IPv4, dalla tabella ***net_table[]***, per copiarlo nella tabella ***ip_table[]*** e trattarlo per le questioni urgenti [94.12.21].

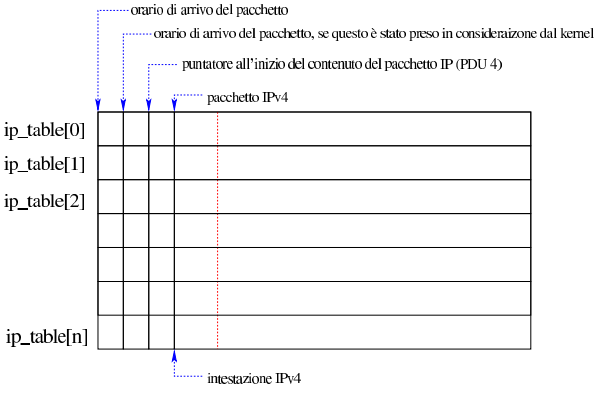
Funzione	Descrizione
`ip_t *ip_reference (void);`	Restituisce il puntatore a un elemento della tabella ***ip_table[]*** che possa essere riutilizzato, perché mai usato prima oppure perché contenente il pacchetto IPv4 ricevuto che risulta essere più vecchio di tutti gli altri [94.12.20].
`ssize_t ip_header (h_addr_t src, h_addr_t dst, uint16_t id, uint8_t ttl, uint8_t protocol, void *buffer, size_t length);`	Scrive, in corrispondenza di ***buffer***, un'intestazione IPv4, sulla base dei dati contenuti negli altri parametri [94.12.17].
`int ip_tx (h_addr_t src, h_addr_t dst, int protocol, const void *buffer, size_t size);`	Produce e trasmette un pacchetto IPv4, partendo dal contenuto che deve avere e dai dati necessari a costruire l'intestazione IPv4 [94.12.22].
`h_addr_t ip_mask (int m);`	A partire da un numero che rappresenta la dimensione di una maschera di rete, si ottiene il valore a 32 bit della maschera stessa. Per esempio, dal valore 16, si ottiene 255.255.0.0 [94.12.18].

In varie situazioni si usa il tipo '**h_addr_t**', il quale rappresenta un indirizzo IPv4, a 32 bit, espresso però secondo l'architettura dell'elaboratore (*host byte order*). Questo tipo derivato si contrappone a quello standard, denominato '**in_addr_t**', il quale rappresenta lo stesso indirizzo, ma secondo l'ordinamento adatto alla trasmissione in rete (*network byte order*).

84.10.1 Tabella IPv4

Quando un pacchetto viene ricevuto ed è riconosciuto dalla funzione ***net_rx()*** come riguardante IPv4, questa chiama la funzione ***ip_rx()*** che lo copia nella tabella ***ip_table[]***, dove rimane fino a quando viene rimpiazzato da un nuovo pacchetto, secondo il criterio per cui i pacchetti più vecchi lasciano il posto a quelli più recenti.

Figura 84.125. Struttura della tabella dei pacchetti IPv4.



Listato 84.126. Struttura di ogni elemento della tabella dei pacchetti IPv4.

```
typedef struct {
  clock_t   clock;
  clock_t   kernel_serviced;
  uint8_t  *pdu4;
  union {
    uint8_t      octet[NET_IP_MAX_PACKET_SIZE];
    struct iphdr header;
  } packet;
} ip_t;
```

Il membro ***kernel_serviced*** contiene inizialmente il valore zero, per poi ottenere una copia dell'orario di arrivo del pacchetto, appena questo risulta essere stato considerato dal kernel, ai fini del protocollo ICMP (in quanto il protocollo ICMP viene gestito internamente). Quando il kernel trova un pacchetto che ha l'orario di arrivo uguale a quello di elaborazione, sa così che l'ha già preso in considerazione nella propria gestione interna e non deve farci altro.

Il membro ***pdu4*** contiene un puntatore all'inizio del contenuto del pacchetto IPv4, ovvero a ciò che c'è nel pacchetto, dopo l'intestazione IPv4 e dopo le opzioni eventuali.

84.10.2 Ricezione di un pacchetto IPv4

La ricezione di un pacchetto IPv4 avviene per opera della funzione ***ip_rx()***, la quale viene avviata da ***net_rx()***, quando si accorge di avere a che fare con un pacchetto di questo tipo.

La funzione ***ip_rx()*** riceve due argomenti, ***n*** e ***f***, i quali rappresentano, rispettivamente, l'indice dell'interfaccia che ha ricevuto la trama e l'indice della trama stessa. Con questi indici, la funzione ***ip_rx()*** è in grado di estrapolare il pacchetto IPv4 dalla tabella ***net_table[]***.

Avendo individuato l'inizio del pacchetto IPv4, verifica l'integrità del contenuto dell'intestazione con il codice di controllo relativo: se la verifica ha successo, e se non si tratta di un frammento (in quanto os32 non gestisce pacchetti frammentati), il pacchetto viene accolto e copiato nella prima posizione disponibile della tabella ***ip_table[]***, annotando l'orario di arrivo.

Il pacchetto ricevuto in questo modo, dovrebbe risultare destinato a un'interfaccia del proprio sistema. Se però l'indirizzo IPv4 di destinazione non è abbinato ad alcuna interfaccia, viene trasmesso un pacchetto ICMP con il messaggio di destinazione irraggiungibile.

Se il pacchetto ricevuto risulta includere informazioni su porte UDP o TCP, viene verificato se nella tabella ***sock_table[]*** è prevista la ricezione nella porta che questo pacchetto dovrebbe raggiungere. Se non è così, viene trasmesso un pacchetto ICMP con il messaggio di porta non raggiungibile.

La tabella ***sock_table[]*** è dichiarata nel file '`kernel/fs.h`' (94.5), perché le connessioni TCP e UDP, a cui si riferisce, hanno un trattamento affine a quello dei file comuni.

Alla fine, se il pacchetto risulta essere di tipo ICMP, viene avviata la funzione ***icmp_rx()*** perché se ne occupi; diversamente viene semplicemente copiato l'orario di ricevimento del pacchetto nel campo che rappresenta l'elaborazione dello stesso a livello IP.

84.10.3 Instradamenti

Nella directory '`kernel/net/route/`' si trovano i file delle funzioni che consentono la gestione degli instradamenti, raccolte nel file di intestazione '`kernel/net/route.h`' (listato 94.12.33 e successivi). Per la limitazione di os32, gli instradamenti servono in pratica solo per la trasmissione, in quanto non è previsto il funzionamento in qualità di router (quindi non si pone il problema di reindirizzare i pacchetti ricevuti).

Figura 84.127. Struttura della tabella ***route_table[]*** per la gestione degli instradamenti.

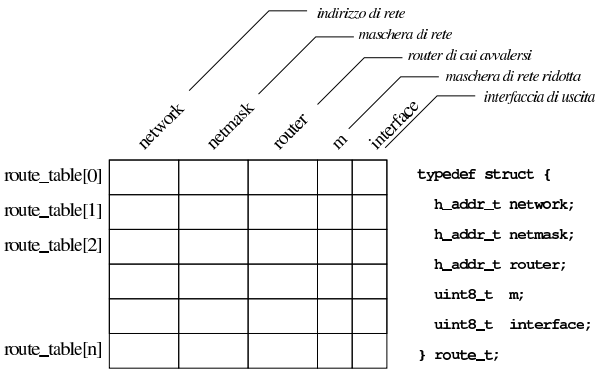


Tabella 84.128. Funzioni per la gestione della tabella degli instradamenti.

Funzione	Descrizione
`void route_init (void);`	Inizializza la tabella ***route_table[]***, predisponendo la voce ***route_table[0]*** per l'interfaccia locale *loopback* [94.12.34].
`void route_sort (void);`	Riordina la tabella degli instradamenti [94.12.39].
`h_addr_t route_remote_to_local (h_addr_t remote);`	Restituisce l'indirizzo IPv4 locale, più adatto per intrattenere una connessione con l'indirizzo remoto fornito come argomento. Questo tipo di analisi viene determinato partendo dalla tabella degli instradamenti, per determinare l'indirizzo IPv4 locale dell'interfaccia interessata dal collegamento [94.12.37].
`h_addr_t route_remote_to_router (h_addr_t remote);`	Restituisce l'indirizzo IPv4 del router da utilizzare per raggiungere l'indirizzo IPv4 remoto specificato. Se l'indirizzo restituito è pari a `-1` significa che non è stata ottenuta alcuna voce corrispondente, mentre se si ottiene zero significa che non c'è bisogno di router per raggiungere la destinazione [94.12.38].

84.11 Gestione del protocollo ICMP

Listato 94.12.10 e successivi.

Quando viene ricevuto un pacchetto IPv4 che contiene un messaggio ICMP, la funzione ***ip_rx()*** chiama la funzione ***icmp_rx()*** per il trattamento di questa informazione. La funzione ***icmp_rx()*** verifica il tipo di messaggio e si comporta di conseguenza: a una richiesta di eco risponde con la trasmissione di un pacchetto appropriato, attraverso la funzione ***icmp_tx_echo()***; a un messaggio di destinazione irraggiungibile, comunica l'informazione nella tabella ***sock_table[]***, dopo aver trovato lì dentro la voce di una connessione con le caratteristiche appropriate.

Tabella 84.129. Funzioni per la gestione del protocollo ICMP.

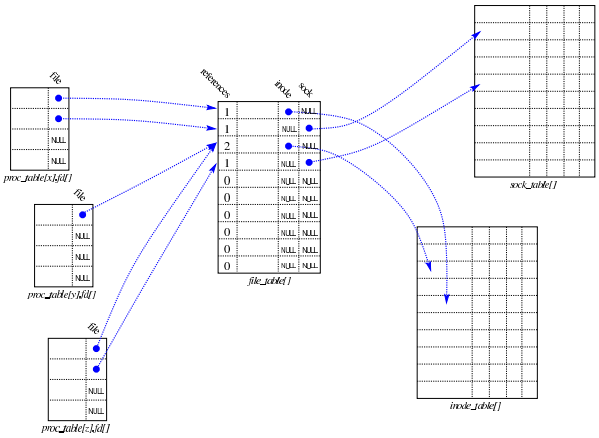
Funzione	Descrizione
`int icmp_rx (int i);`	Recepisce un pacchetto ICMP dalla tabella dei pacchetti IPv4, conoscendo l'indice di questa tabella, trattando le richieste di eco e i messaggi di destinazione irraggiungibile [94.12.11].
`int icmp_tx (h_addr_t src, h_addr_t dst, int type, int code, icmp_packet_t *icmp, size_t size);`	Completa l'intestazione di un pacchetto ICMP, con il tipo e il codice del messaggio, oltre al codice di controllo; questo pacchetto poi viene imbustato in un pacchetto IPv4 per la trasmissione [94.12.12].
`int icmp_tx_echo (h_addr_t src, h_addr_t dst, int type, int code, int identifier, int sequence, uint8_t *data, size_t size);`	Prepara un pacchetto ICMP di eco, partendo dal contenuto di questo pacchetto e dalle informazioni da aggiungere nell'intestazione ICMP; l'intestazione di questo pacchetto viene completata attraverso la funzione ***icmp_tx()***, la quale poi provvede a imbustarlo in un pacchetto IPv4 per la trasmissione [94.12.13].
`int icmp_tx_unreachable (h_addr_t src, h_addr_t dst, int type, int code, icmp_packet_t *icmp, size_t size);`	Prepara un pacchetto ICMP con un messaggio di destinazione irraggiungibile, partendo dal contenuto di questo pacchetto e dalle informazioni da aggiungere nell'intestazione ICMP; l'intestazione di questo pacchetto viene completata attraverso la funzione ***icmp_tx()***, la quale poi provvede a imbustarlo in un pacchetto IPv4 per la trasmissione [94.12.14].

84.12 Gestione dei protocolli UDP e TCP

Listati 94.5, 94.12.53 e successivi, 94.12.40 e successivi.

Per la gestione dei protocolli UDP e TCP, os32 definisce una tabella in cui ogni voce descrive una connessione. Si tratta della tabella ***sock_table[]*** le cui voci sono collegate dalla tabella ***file_table[]***. In pratica, quando si utilizza la funzione ***socket()*** per creare una connessione UDP o TCP, si ottiene un descrittore di file che fa riferimento a una voce nella tabella ***sock_table[]***, invece che alla tabella degli inode.

Figura 84.130. Collegamento tra le tabelle dei descrittori e la tabella complessiva dei file. La tabella ***proc_table[x].fd[]*** rappresenta i descrittori di file del processo elaborativo ***x***.



La struttura degli elementi della tabella ***sock_table[]*** è molto articolata. In generale si distingue una prima parte, comune ai protocolli UDP e TCP, rispetto a una seconda specifica per il protocollo TCP.

Listato 84.131. Struttura di ogni elemento della tabella dei socket.

```
struct sock
{
  int         family;
  int         type;
  int         protocol;
  h_addr_t    laddr;          // indirizzo locale
  h_port_t    lport;          // porta locale
  h_addr_t    raddr;          // indirizzo remoto
  h_port_t    rport;          // porta remota
  struct {
    clock_t   clock[IP_MAX_PACKETS];
  } read;
  uint8_t     active      : 1, // socket utilizzato?
              unreach_net : 1, // rete irraggiungibile?
              unreach_host: 1, // destinazione
                               //  irraggiungibile?
              unreach_prot: 1, // protocollo
                               //  irraggiungibile?
              unreach_port: 1; // porta irraggiungibile?
  struct
  {
    uint16_t  conn        : 4, // stato della connessione
              can_write   : 1, // si può scrivere in
                               //  'send_data[]'?
              can_read    : 1, // si può leggere a
                               //  partire da
                               //   '*recv_index'?
              can_send    : 1, // si possono inviare
                               //  dati?
              can_recv    : 1, // si possono ricevere
                               //  dati?
              send_closed : 1, // il canale di
                               //  trasmissione
                               //   è chiuso?
              recv_closed : 1; // il canale di ricezione
                               //  è chiuso?
    //
    uint32_t  lsq[16];        // array della sequenza locale
    uint32_t  lsq_ack;        // numero di sequenza in attesa
                              //  di conferma
    uint32_t  rsq[16];        // array della sequenza remota
    uint8_t   lsqi      : 4, // indice dell'array della
                              //  sequenza locale
              rsqi      : 4; // indice dell'array della
                              //  sequenza remota
    //
    clock_t   clock;          // istante dell'ultima
                              //  trasmissione
    //
    uint8_t   send_data[TCP_MSS - sizeof (struct tcphdr)];
                              // dati da trasmettere
```

```
    size_t    send_size;     // dimensione dei dati da
                             //  trasmettere
    int       send_flags;
    uint8_t   recv_data[TCP_MAX_DATA_SIZE];
                             // dati ricevuti
    size_t    recv_size;     // dimensione dei dati
                             //  ricevuti
    uint8_t  *recv_index;    // indice per la lettura dei
                             //  dati ricevuti
    pid_t     listen_pid;    // processo in ascolto della
                             //  porta locale, in attesa di
                             //  connessioni
    int       listen_max;    // numero massimo di richieste
                             //  di connessione accettabili
    int       listen_queue[SOCK_MAX_QUEUE];
                             // descrittori di connessioni
                             //  realizzate
  } tcp;
};
```

Tabella 84.132. Funzioni per la gestione dei protocolli TCP e UDP.

Funzione	Descrizione
`int tcp_tx_raw (h_port_t sport, h_port_t dport, uint32_t seq, uint32_t ack_seq, int flags, h_addr_t saddr, h_addr_t daddr, const void *buffer, size_t size);`	Costruisce un pacchetto TCP, utilizzando i dati forniti come argomenti della chiamata; quindi lo trasmette attraverso la funzione ***ip_tx()*** che a sua volta provvede a imbustarlo in un pacchetto IP prima della trasmissione effettiva. Si tratta comunque di una funzione usata soltanto per fare dei test di funzionamento [94.12.50].
`void tcp_show (h_addr_t src, h_addr_t dst, const struct tcphdr *tcphdr);`	Funzione diagnostica realizzata per visualizzare alcune informazioni su un pacchetto TCP, di cui si conosce il contenuto e gli indirizzi IPv4. Questa funzione viene usata prevalentemente da ***tcp()***, quando si attiva la macro-variabile ***DEBUG*** [94.12.46].
`int tcp_tx_rst (void *ip_packet);`	Sulla base di un pacchetto IP ricevuto con un contenuto TCP, trasmette un pacchetto TCP di azzeramento (RST) [94.12.51].
`int tcp_tx_sock (void *sock_item);`	Trasmette quanto contenuto nella coda del socket indicato come argomento, ammesso che il socket sia nella condizione di poter trasmettere [94.12.52].
`int tcp_tx_ack (void *sock_item);`	Trasmette un pacchetto TCP vuoto contenente la conferma di quanto ricevuto in precedenza, sulla base dello stato attuale del socket indicato come argomento [94.12.49].
`int tcp_rx_ack (void *sock_item, void *packet);`	Verifica che il pacchetto indicato come secondo parametro, contenga una conferma valida per il socket specificato come primo parametro [94.12.44].

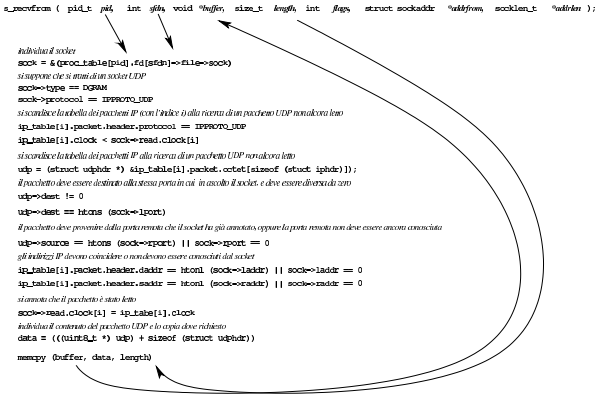
Funzione	Descrizione
`int tcp_rx_data (void *sock_item, void *packet);`	Legge il contenuto di un pacchetto TCP e lo copia all'interno della memoria tampone del socket rappresentata da ***sock_table[s].tcp.recv_data*** (dove ***s*** è l'indice del socket considerato) [94.12.45].
`int tcp_connect (void *sock_item);`	Fa in modo di mettere il socket in connessione, ammesso che ciò sia possibile. Questa funzione serve a ***s_connect()*** [94.12.43].
`int tcp_close (void *sock_item);`	Fa in modo di mettere il socket nello stato di chiusura, ammesso che ciò sia possibile. Questa funzione serve a ***s_close()*** [94.12.42].
`int tcp (void);`	Viene chiamata da ***proc_sch_net()*** e si occupa di gestire lo stato di tutte le connessioni TCP in essere in quel momento [94.12.41].
`int udp_tx (h_port_t sport, h_port_t dport, h_addr_t saddr, h_addr_t daddr, const void *buffer, size_t size);`	Assembla un pacchetto UDP e lo trasmette (dopo aver costruito a sua volta il pacchetto IPv4 complessivo) [94.12.54].

84.12.1 UDP

Quando viene ricevuto un pacchetto IPv4 che contiene dati del protocollo UDP, dopo aver verificato che esiste effettivamente una porta UDP in attesa di ricevere nella tabella ***sock_table[]***, questo viene semplicemente lasciato nella tabella ***ip_table[]***, annotando soltanto che il kernel lo ha già preso in considerazione.

L'acquisizione effettiva del pacchetto UDP avviene attraverso la funzione ***s_recvfrom()***, la quale costituisce la versione interna al kernel di ***recvfrom()***. La funzione ***s_recvfrom()***, chiamata per leggere da un socket UDP, cerca nella tabella ***ip_table[]*** un pacchetto corrispondente alle caratteristiche del socket, che non sia già stato preso in considerazione dal socket stesso (il membro ***read.clock[i]***, dove ***i*** corrisponde all'indice del pacchetto trovato nella tabella ***ip_table[]***, contiene l'orario di un pacchetto già letto in quella posizione: se l'orario del pacchetto contenuto nella tabella ***ip_table[]*** è più recente, allora deve essere letto). Se il pacchetto viene accettato, si aggiorna nel socket il valore del membro ***read.clock[i]*** con l'orario di ricevimento del pacchetto (per evitare di rileggerlo un'altra volta), quindi viene copiato il contenuto del pacchetto nella destinazione specificata dagli argomenti della funzione.

Figura 84.133. Semplificazione dei punti principali del procedimento di lettura di un pacchetto UDP, attraverso la funzione ***s_recvfrom()***.



Va osservato che la lettura del pacchetto UDP, così come viene fatta da os32, si limita alla porzione specificata dalla dimensione massima della memoria tampone; se poi si esegue una nuova lettura, si cerca semplicemente un altro pacchetto, senza terminare eventualmente la lettura del precedente.

La trasmissione avviene attraverso la funzione ***s_send()*** (corrispondente a ***send()*** dal lato utente). Questa funzione, dopo aver determinato che si tratta di un socket UDP, si avvale a sua volta della funzione ***udp_tx()*** per costruire e spedire effettivamente il pacchetto. Come nel caso della ricezione, la trasmissione riguarda un solo pacchetto, e le informazioni eccedenti sono semplicemente eliminate.

84.12.2 TCP

La gestione del TCP è estremamente più complessa rispetto a UDP, in quanto richiede un proprio sistema di gestione delle connessioni. La funzione ***tcp()*** ha il compito di scandire la tabella dei pacchetti ***ip_table[]*** alla ricerca di quelli che riguardano il protocollo TCP e che non sono ancora stati considerati, aggiornando lo stato delle connessioni relative. La funzione ***tcp()*** è chiamata a ogni interruzione da ***proc_sch_net()***.

La funzione ***tcp()***, quando individua nella tabella ***ip_table[]*** un pacchetto TCP ancora da prendere in considerazione, deve valutare le caratteristiche del pacchetto trovato in relazione allo stato della connessione eventualmente già in corso, agendo di conseguenza. Ciò significa che la funzione ***tcp()*** può trovarsi nella necessità di trasmettere a sua volta pacchetti TCP alla controparte, per il fine della gestione della connessione.

È importante osservare che gli indicatori ***sock_table[].tcp.can_read*** e ***sock_table[].tcp.can_write***, necessari a controllare la lettura e la scrittura del socket con le funzioni ***s_recvfrom()*** e ***s_send()***, sono aggiornati dalla funzione ***tcp()***.

Le funzioni ***s_recvfrom()*** e ***s_send()***, se si trovano nell'impossibilità di leggere o scrivere il socket richiesto, in condizioni normali mettono il processo relativo in pausa, in attesa di un cambiamento. Inoltre, la funzione ***s_recvfrom()*** deve aggiornare l'indice di lettura interno al socket, in modo che la lettura successiva riprenda da quella posizione. In pratica, lettura e scrittura avvengono qui in modo analogo a quello di un file, in un flusso continuo di byte.

Il risveglio dei processi in attesa di leggere o scrivere un socket avviene per opera della funzione ***proc_sch_net()*** dopo aver avviato ***tcp()*** per aggiornare lo stato dei socket TCP, in base al fatto che sia stato ricevuto qualcosa o che ci sia motivo di ritenere che sia possibile scrivere attraverso un socket bloccato precedentemente in scrittura.

Esiste un grosso limite di os32, relativo alla gestione del TCP: la chiusura di una connessione elimina le informazioni relative al socket, mentre la controparte potrebbe non essere ancora pronta per recepire tale conclusione. Questa semplificazione serve a far sì che ci sia sempre corrispondenza tra il descrittore di file e il socket, mentre in un sistema reale, il socket deve poter continuare a esistere per un certo tempo, benché chiuso, anche dopo la chiusura del descrittore di file relativo.

Va poi considerato che os32 gestisce finestre TCP pari a un solo pacchetto, per cui si attende la conferma di ogni singola trasmissione dalla controparte.