Tricky C

...a Davide Bertuzzi

Copyright © 1994-1998 Barninga Z!.

La riproduzione, con qualsiasi mezzo, di questo libro o di sue parti è liberamente consentita alle persone fisiche, purché non effettuata a fini di lucro. Le persone giuridiche, nonché le persone fisiche qualora agiscano con finalità di lucro, devono preventivamente ottenere autorizzazione scritta da parte dell'autore. Alle medesime condizioni sono ammessi utilizzo e riproduzione dei files sorgenti e compilati memorizzati sul floppy disk allegato al libro.

Tutti i nomi di marchi e nomi di prodotti citati nel testo e nei sorgenti su floppy disk sono marchi depositati o registrati dai rispettivi proprietari.

Pensierino di Barninga Z!

Imperdonabile il titolo in lingua inglese, vero? In effetti, avrei potuto scegliere *Stratagemmi di programmazione in linguaggio C* o qualcosa di simile, ma sarebbe stato forse pretenzioso e, tutto sommato, meno efficace. D'altra parte, chiunque si sia scontrato con l'informatica, in particolare con la programmazione, è certamente consapevole della presenza costante e quasi ingombrante dell'inglese: si potrebbe azzardare che esso rappresenta, in questo campo, la lingua... di *default*.

Tricky C nasce come raccolta di appunti: un modo per non perdere traccia delle scoperte fatte in tante notti più o meno insonni; solamente in un secondo tempo si arricchisce della parte che descrive (sommariamente) le principali regole sintattiche del linguaggio e le sue caratteristiche fondamentali. La parte più interessante (o meno noiosa!) rimane comunque, a mio parere, quella dedicata all'esplorazione degli stratagemmi: interrupt, TSR, Device Driver e altro ancora. Molto spesso, infatti, per realizzare programmi apparentemente complessi, è sufficiente conoscere alcuni trucchi del mestiere, tipico frutto di scoperte quasi casuali o celati tra le righe della manualistica che accompagna i compilatori.

Il contenuto di queste pagine vuole dunque costituire, senza pretesa alcuna di completezza, un insieme di suggerimenti: è buona norma ricordare sempre che essi sono in parte tratti dalle più svariate fonti, ufficiali e non, ed in parte basati sulla mia esperienza personale, in campo professionale ed amatoriale. Mi vedo pertanto costretto a declinare ogni responsabilità per qualsiasi conseguenza derivante dall'utilizzo delle tecniche descritte e dei listati riprodotti (e mi scuso in anticipo per gli errori, temo non solo ortografici, sicuramente sfuggiti alle pur numerose riletture).

E' doveroso, a questo punto, ringraziare coloro che hanno collaborato alla realizzazione di questo lavoro con suggerimenti, contributi tecnici, critiche ed incoraggiamenti. Sarebbe impossibile menzionare tutti singolarmente, quindi dovranno accontentarsi di un *Grazie!!* collettivo. Tuttavia, qualcuno merita davvero un riconoscimento particolare: Flavio Cometto (abile risolutore di decine di *miei* dubbi e problemi, nonché autore di geniali intuizioni, tra le quali il titolo), Angelo Secco (molto di quanto si trova in *Tricky C* è stato scoperto nello sforzo di risolvere decine di *suoi* dubbi e problemi) e gli amici di *Zero! BBS Torino* e *Radio Black Out Torino* (soprattutto - in ordine alfabetico - Roberto Dequal, Luciano Paccagnella e Luisa Tatoni, senza il cui tenace supporto tecnico e organizzativo *Tricky C* sarebbe forse rimasto intrappolato per sempre nei meandri del mio hard disk).

Grazie!! Anche a tutti coloro dai quali ho ricevuto il prezioso supporto necessario a creare e pubblicare in Internet la versione HTML di *Tricky C* (dedicata a mia figlia Ilaria Rossana Ginevra) e, in particolare, a tutto lo staff di *Aspide*. Gli interessati possono dare un'occhiata all'indirizzo `http://www.aspide.it/trickyc`.

Ma il ringraziamento più vivo è per mia moglie, che ha sopportato con infinita pazienza lunghe serate trascorse alle prese con esperimenti, bozze, listati e scartoffie di vario genere. A lei sono grato per avermi dato la forza di arrivare sino in fondo.

Ultima revisione: luglio 1998

INDICE

INDICE..1

IL LINGUAGGIO C: CENNI GENERALI..1

LA PRODUZIONE DEI PROGRAMMI C...3
- Linguaggi interpretati e compilati 3
  - L'Interprete...3
  - Il Compilatore..3
  - Quale dei due?..4
- Dall'idea all'applicazione...4

I PROGRAMMI C: UN PRIMO APPROCCIO...7

LA GESTIONE DEI DATI IN C 11
- I tipi di dato..11
- Le variabili...13
- I puntatori 16
  - Gli indirizzi di memoria 16
  - Gli operatori * e & 17
  - Complicazioni 21
  - Puntatori far e huge..21
  - Puntatori static...25
  - Le stringhe 25
  - Gli array...29
  - L'aritmetica dei puntatori..32
  - Puntatori a puntatori...33
  - Puntatori void...34
- L'accessibilità e la durata delle variabili 34
  - Le variabili automatic...34
  - Le variabili register...36
  - Le variabili static 37
  - Le variabili external 39
- Le costanti 42
  - Le costanti manifeste...44
  - Le costanti simboliche..46
- Entità complesse..47
  - Le strutture..48
  - Le unioni..55
  - Gli enumeratori..57
  - I campi di bit 58

GLI OPERATORI 61
- Not logico...63
- Complemento a uno...64
- Negazione algebrica...64
- Autoincremento e autodecremento...64
- Cast e conversioni di tipo 65
- Operatore sizeof()...68
- Operatori aritmetici...68
- Resto di divisione intera...68

Shift su bit 69
Operatori logici di test 70
Operatori logici su bit 71
Operatore condizionale 73
Assegnamento 73
Separatore di espressioni 74

IL FLUSSO ELABORATIVO 75
Le istruzioni di controllo condizionale 75
if...else 75
switch 77
goto 79
I cicli 79
while 79
do...while 81
for 81

LE FUNZIONI 85
Definizione, parametri e valori restituiti 87
Puntatori a funzioni 93
La ricorsione 100
main(): parametri e valore restituito 105

ALLOCAZIONE DINAMICA DELLA MEMORIA 109

L'I/O E LA GESTIONE DEI FILE 115
Gli stream 115
Stream standard 115
Gli stream in C 116
Il caching 124
Altri strumenti di gestione dei file 126

LANCIARE PROGRAMMI 129
La libreria C 129
system() 129
spawn...() 131
Funzioni del gruppo "l" 131
Funzioni del gruppo "v" 132
exec...() 133
Tabella sinottica 133
Condivisione dei file 135
Portabilità 136

GLI INTERRUPT: UTILIZZO 139
ROM-BIOS e DOS, Hardware e Software 139
La libreria C 140

I MODELLI DI MEMORIA 143
Tiny model 144
Small model 144
Medium model 144
Compact model 145
Large model 145
Huge model 147

SCRIVERE FUNZIONI DI LIBRERIA 149
Accorgimenti generali 149

Esigenze tecniche 150
La realizzazione pratica....152

INTERAZIONE TRA C E ASSEMBLER 155
Assembler....155
Inline assembly....157
Lo stack 158
Utilizzo dei registri 161
Variabili e indirizzi C....164
Altri strumenti di programmazione mista....167
Pseudoregistri....167
`geninterrupt()` 169
`__emit__()` 169
Uno stratagemma: dati nel code segment....170

C E CLIPPER....177
Passaggio di parametri e restituzione di valori 177
Reference e puntatori....181
Allocazione della memoria....182
Alcuni esempi 183

GESTIONE A BASSO LIVELLO DELLA MEMORIA....189
Il compilatore C....189
Memoria convenzionale....190
Upper memory 198
Memoria espansa....202
Memoria estesa, High Memory Area e UMB....226
I servizi XMS per la memoria estesa 227
I servizi XMS per la HMA....239
I servizi XMS per gli UMB....246

GLI INTERRUPT: GESTIONE....251
La tavola dei vettori....251
Le funzioni interrupt....253
Le funzioni far....259
Utilizzo dei gestori originali 261
Due o tre esempi....268
Inibire CTRL-C e CTRL-BREAK....268
Inibire CTRL-ALT-DEL 271
Redirigere a video l'output della stampante 273

I PROGRAMMI TSR....275
Tipi di TSR 275
La struttura del TSR....275
Installazione del TSR....276
Dati, stack e librerie....277
Ottimizzazione dell'impiego della RAM 280
Allocazione dinamica della RAM....282
I TSR e la memoria EMS....283
Rilasciare l'environment del TSR 285
Due parole sullo stack 286
Utilizzo delle funzioni di libreria....289
Gestione degli interrupt....294
Hardware, ROM-BIOS e DOS 294
I flag del DOS....295
Int 05h (BIOS): Print Screen....299

Int 08h (Hardware): Timer 300
Int 09h (Hardware): Tastiera........ 300
Int 10h (BIOS): Servizi video 303
Int 13h (BIOS): I/O dischi........ 304
Int 16h (BIOS): Tastiera........ 304
Int 1Bh (BIOS): CTRL-BEAK 307
Int 1Ch (BIOS): Timer........ 308
Int 21h (DOS): servizi DOS........ 308
Int 23h (DOS): CTRL-C........ 309
Int 24h (DOS): Errore critico........ 310
Int 28h (DOS): DOS libero........ 311
Int 2Fh (DOS): Multiplex........ 311
Gestione dello I/O 313
Tastiera........ 313
Video........ 314
File........ 318
DTA 322
Gestione del PSP........ 324
Ricapitolando...... 325
Disattivazione e disinstallazione........ 327
keep() ed exit() 327
Suggerimenti operativi........ 328
Controllo di avvenuta installazione........ 329
Richiesta dell'indirizzo dei dati........ 329
Rimozione della porzione residente del TSR 330
Precauzioni........ 330
Alcuni esempi pratici........ 334

I DEVICE DRIVER........ 353
Aspetti tecnici........ 353
Il bootstrap........ 353
Tipi di device driver........ 354
Struttura e comportamento dei device driver........ 355
Il Device Driver Header: in profondità 357
Il Request Header e i servizi: tutti i particolari 360
Servizio 00: Init........ 363
Servizio 01: Media Check........ 365
Servizio 02: Build BPB........ 366
Servizio 03: IOCTL Read 367
Servizio 04: Read (Input)........ 368
Servizio 05: Nondestructive Read........ 369
Servizio 06: Input Status........ 370
Servizio 07: Flush Input Buffers........ 371
Servizio 08: Write (Output)........ 371
Servizio 09: Write With Verify........ 371
Servizio 10: Output Status........ 371
Servizio 11: Flush Output Buffers 371
Servizio 12: IOCTL Write........ 372
Servizio 13: Device Open 372
Servizio 14: Device Close........ 373
Servizio 15: Removable Media........ 373
Servizio 16: Output Until Busy 373
Servizio 19: Generic IOCTL Request........ 374
Servizio 23: Get Logical Device........ 375

Servizio 24: Set Logical Device....376
I Device Driver e il C....377
Un timido tentativo....378
Progetto di un toolkit....385
Il nuovo startup module....386
La libreria di funzioni....397
La utility per modificare gli header....431
Il toolkit al lavoro....439
La funzione init()....441
Altre funzioni e macro....443
L'accesso al device driver request header....443
Le variabili globali dello startup module....445
Esempio: alcune cosette che il toolkit rende possibili....446
Esempio: esperimenti di output e IOCTL....450

LINGUAGGIO C E PORTABILITÀ....461
Dipendenze dallo hardware....461
Dipendenze dai compilatori....463
Dipendenze dal sistema operativo....465

DI TUTTO... DI PIÙ....467
Due file sono il medesimo file?....467
Dove mi trovo?....469
La command line....475
_setargv__() e _setenvp__()....476
WILDARGS.OBJ....477
PSP e command line....478
Una gestione Unix-like....479
La sintassi....479
Le funzioni per la libreria....481
La gestione degli errori....499
Un esempio di... disinfestante....501
Un esempio di... pirateria....505
Individuare la strategia di protezione....505
Superare la barriera....507
Sulla retta via....509
Insoddisfatti della vostra tastiera?....509
Ridefinire la tastiera....509
Una utility....524
Vuotare il buffer della tastiera....526
Catturare il contenuto del video....527
Disinstallare i TSR....542
Vettori di interrupt o puntatori?....550
Il CMOS....554
C... come Cesare....560
Lavorare con i file batch....564
L'idea più semplice: EMPTYLVL....565
Data e ora nei comandi: DATECMD....567
File piccoli a piacere: FCREATE....577
Attendere il momento buono: TIMEGONE....579
Estrarre una sezione da un file: SELSTR....587
Estrarre colonne di testo da un file: CUT....593
Il comando FOR rivisitato: DOLIST....603
I comandi nidificati: CMDSUBST....606

Numeri a caso 612
CONTENUTO DEL FLOPPY DISK 619
INDICE DELLE FIGURE.. 627
INDICE DELLE TABELLE.. 629
INDICE ANALITICO 631

IL LINGUAGGIO C: CENNI GENERALI

Come qualunque linguaggio, anche il linguaggio C si compone di parole e regole; queste ultime costituiscono la portante grammaticale e sintattica che consente di aggregare le prime per formare frasi di senso compiuto.

Come qualsiasi linguaggio di programmazione, inoltre, il linguaggio C rappresenta in qualche modo un compromesso tra l'intelligenza umana e l'intrinseca stupidità della macchina. Esso costituisce il mezzo tramite il quale il programmatore "spiega" alla macchina come effettuare determinate operazioni.

Naturalmente (si perdoni l'ovvietà) la macchina non è in grado di capire direttamente il linguaggio C: essa è troppo stupida per poterlo fare, e d'altra parte il C è direttamente leggibile e comprensibile dagli esseri umani; troppo distante, quindi, dalla logica binaria, l'unica che abbia senso per un calcolatore.

Perché la macchina possa eseguire un programma scritto in C, anche il più banale, occorre rielaborarlo fino a ridurlo ad un insieme di valori esprimibili in codice binario: diciamo, per capirci, che questi rappresentano la traduzione in linguaggio macchina di quanto il programmatore ha espresso in linguaggio C, il quale non è altro che un "sottolinguaggio" della lingua parlata dal programmatore stesso, o, in altre parole, un sottoinsieme di quella.

In effetti, l'utilizzo di sottoinsiemi della lingua parlata per scrivere programmi deriva dall'esigenza di semplificare la traduzione del programma nell'unica forma comprensibile alla macchina (il binario, appunto): lo scopo è eliminare a priori le possibili ambiguità, le scorrettezze grammaticali e, in generale, tutti quei "punti oscuri" dei discorsi in lingua naturale che sono solitamente risolti dall'essere umano mediante un processo di interpretazione o di deduzione dei significati da conoscenze che non sono direttamente reperibili nel discorso stesso ma derivano dall'esperienza e dalla capacità di inventare. La macchina non impara e non inventa.

Sebbene quanto detto valga, evidentemente, per tutti i linguaggi di programmazione, va sottolineato che il C ha caratteristiche proprie, che ne fanno, in qualche modo, un linguaggio particolare.

Innanzitutto esso dispone di un insieme limitatissimo di istruzioni. E' dunque un linguaggio intrinsecamente povero, ma può essere facilmente ed efficacemente arricchito: chiunque può aggiungere nuove istruzioni (o meglio, funzioni) alla strumentazione che accompagna il linguaggio. In pratica si possono coniare neologismi e creare dei vocabolari aggiuntivi a quello proprio del linguaggio, che potranno essere utilizzati all'occorrenza. Di fatto, in C, anche la gestione dello I/O (Input/Output) è implementata così. Un gran numero di funzioni esterne al linguaggio è ormai considerato universalmente parte del linguaggio stesso ed accompagna sempre il compilatore[1], nonostante, dal punto di vista strettamente tecnico, questo sia comunque in grado di riconoscere solo un piccolo numero di istruzioni intrinseche.

Tutto ciò si traduce anche in una relativa semplicità funzionale del compilatore: sono quindi molti gli ambienti per i quali è stato sviluppato un compilatore dedicato. Essendo la maggior parte del linguaggio esterna al compilatore, è possibile riutilizzarla semplicemente facendo "rimasticare" i file sorgenti delle funzioni (e sono sorgenti C) al compilatore con il quale dovranno essere associati. In questo sta la cosiddetta portabilità del C, cioè la possibilità di utilizzare gli stessi sorgenti in ambienti diversi, semplicemente ricompilandoli.

Un'altra caratteristica del C è la scarsità di regole sintattiche. Il programmatore ha molta libertà di espressione, e può scrivere programmi che riflettono ampiamente il suo personalissimo stile, il suo modo di risolvere i problemi, il suo tipo di approccio all'algoritmo. Ciò ha senz'altro riflessi positivi sull'efficienza del programma, ma può essere causa di difficoltà talora insuperabili, ad esempio nei successivi interventi sul sorgente a scopo di manutenzione o di studio, quando lo stile del programmatore

[1] Pericoloso strumento di cui diremo tra breve.

sia eccessivamente criptico. La regola forse più importante che il programmatore C deve seguire, nonostante non faccia parte del linguaggio, è la cosiddetta regola "KISS" (Keep It Simple, Stupid).

Infine, il C mette a disposizione concetti e strumenti che consentono un'interazione "di basso livello" con la macchina e con le sue risorse. Per "basso livello" non si intende qualcosa di poco valore, ma una piccola distanza dalla logica binaria della macchina stessa. Ne deriva la possibilità di sfruttare a fondo, e con notevole efficienza, tutta la potenza dell'elaboratore. Non a caso il C è nato come linguaggio orientato alla scrittura di sistemi operativi[2]; tuttavia la possibilità di estenderlo mediante librerie di funzioni ha fatto sì che, nel tempo, esso divenisse linguaggio di punta anche nella realizzazione di programmi applicativi, per i quali ne erano tradizionalmente usati altri[3].

L'evoluzione più recente del C è rappresentata dal C++, il quale è, in pratica, un "superset" del C stesso, del quale riconosce ed ammette tutti i costrutti sintattici, affiancandovi le proprie estensioni: queste consentono al programmatore di definire delle entità composte di dati e del codice eseguibile atto alla loro manipolazione. Dette entità vengono così trattate, a livello logico, come se fossero tipi di dato intrinseci al linguaggio e consentono pertanto di descrivere la realtà su cui il programma opera in termini di linguaggio fondamentale: è la cosiddetta programmazione ad oggetti. Che il C++ sia un'estensione del C (e non un nuovo linguaggio) è dimostrato dal fatto che gli arricchimenti sintattici sono tutti implementati a livello di preprocessore[4]; in altre parole, ogni sorgente C++ viene ridotto a sorgente C dal preprocessore e poi masticato da un normale compilatore C.

I programmi eseguibili risultanti sono efficienti e compatti in quanto il compilatore traduce ogni istruzione C in un numero limitato di istruzioni macchina (al limite una sola): solo l'Assembler è più efficiente (ma rende la vita del programmatore assai più dura).

---

[2] Il sistema Unix è l'esempio più famoso.

[3] Ad esempio il Cobol per gli applicativi gestionali e il Fortran per quelli matematici.

[4] Il preprocessore è un programma che opera semplicemente la sostituzione di certe sequenze di caratteri con altre. Esso fa parte da sempre della dotazione di strumenti standard dei compilatori C; il C++ si fonda su versioni particolarmente sofisticate di proprocessore affiancate a compilatori C tradizionali.

LA PRODUZIONE DEI PROGRAMMI C

LINGUAGGI INTERPRETATI E COMPILATI

Si è detto che il linguaggio di programmazione consente di esprimere gli algoritmi in modo "umano", incomprensibile alla macchina, la quale è in grado di eseguire esclusivamente istruzioni codificate in modo binario, cioè con una sequenza di 1 e 0 (che rappresentano, a loro volta, la presenza o l'assenza di una tensione elettrica).

E' perciò indispensabile che il sorgente del programma (cioè il file contenente il testo scritto dal programmatore in un dato linguaggio di programmazione) venga elaborato e trasformato in una sequenza di codici binari significativi per l'elaboratore.

Gli strumenti generalmente utilizzati allo scopo rientrano in due categorie: interpreti e compilatori.

L'Interprete

L'interprete è un programma in grado di leggere un sorgente in un certo linguaggio e, istruzione per istruzione, verificarne la sintassi, effettuarne la traduzione in linguaggio macchina e far eseguire al microprocessore della macchina il codice binario generato. La logica con cui l'interprete lavora è proprio quella di un... interprete: se la medesima istruzione viene eseguita più volte (ad esempio perché si trova all'interno di un ciclo), ad ogni iterazione ne viene verificata la correttezza sintattica, ne è effettuata la traduzione, e così via. L'esecuzione del programma può essere interrotta in qualunque momento ed è possibile modificarne una parte, per poi riprendere l'esecuzione dal punto di interruzione. L'interprete è inoltre in grado di interrompere spontaneamente l'esecuzione quando rilevi un errore di sintassi, consentire al programmatore la correzione dell'errore e riprendere l'esecuzione dall'istruzione appena modificata.

E' facile intuire che la programmazione interpretata facilita enormemente le varie fasi di sviluppo e correzione del programma; tuttavia essa presenta alcuni pesanti svantaggi: il programma "gira" lentamente (perché ogni istruzione deve essere sempre verificata e tradotta, anche più volte nella medesima sessione di lavoro, prima di essere eseguita) ed inoltre può essere eseguito solo ed esclusivamente attraverso l'interprete.

Un esempio classico di linguaggio interpretato (nonostante ve ne siano in commercio versioni compilate o miste) è il Basic.

Il Compilatore

Anche in questo caso l'obiettivo di fondo è tradurre in linguaggio macchina un sorgente scritto in un linguaggio di programmazione perché l'elaboratore sia in grado di eseguirlo; tuttavia l'approccio al problema è sostanzialmente diverso.

Il sorgente viene letto dal compilatore, che effettua il controllo sintattico sulle istruzioni e le traduce in linguaggio macchina. Il risultato della traduzione è scritto in un secondo file, detto *object file*. Questo non è ancora eseguibile dal microprocessore, in quanto non incorpora il codice binario delle funzioni esterne al linguaggio: è dunque necessaria una fase ulteriore di elaborazione, alla quale provvede il *linker*, che incorpora nell'object file gli object file contenenti le funzioni esterne, già compilate in precedenza, solitamente raccolti in "contenitori" detti librerie. Il linker produce in output un terzo file, il

programma vero e proprio, direttamente eseguibile dal microprocessore con la sola intermediazione del sistema operativo.

Per eseguire il programma, dunque, non servono né compilatore o linker, né, tantomeno, il file sorgente.

I vantaggi rispetto all'interprete, in termini di velocità e semplicità di esecuzione, sono evidenti, a fronte di una maggiore complessità del ciclo di sviluppo. Infatti il compilatore, nel caso in cui rilevi errori nel sorgente, li segnala e non produce alcun object file. Il programmatore deve analizzare il sorgente, correggere gli errori e ritentare la compilazione: detta sequenza va ripetuta sino a quando, in assenza di segnalazioni d'errore da parte del compilatore, viene prodotto un object file pronto per l'operazione di *linking*. Anche in questa fase potranno verificarsi errori: il caso classico è quello della funzione esterna non trovata nella libreria. Anche questa volta occorre analizzare il sorgente, correggere l'errore (il nome della funzione potrebbe essere stato digitato in modo errato[5]) e ripetere non solo il linking, ma anche la compilazione.

Solo in assenza di errori tanto nella fase di compilazione quanto in quella di linking si può ottenere un file eseguibile; in altre parole: il programma funzionante[6].

Il C rientra a pieno titolo nella folta schiera dei linguaggi compilati (insieme a Cobol e Fortran, per fare un paio di esempi).

Quale dei due?

Come si vede, sia il compilatore che l'interprete portano con sé vantaggi e svantaggi peculiari. Quale delle due tecniche utilizzare, allora?

Al riguardo si può osservare che la finalità di un programma non è "essere sviluppato", ma servire "bene" allo scopo per il quale viene creato; in altre parole esso deve essere semplice da utilizzare e, soprattutto, efficiente. La scelta del compilatore è quindi d'obbligo per chi intenda realizzare applicazioni commerciali o, comunque, di un certo pregio.

L'interprete si pone quale utile strumento didattico per i principianti: l'interattività nello sviluppo dei programmi facilita enormemente l'apprendimento del linguaggio.

In molti casi, comunque, la scelta è obbligata: per quel che riguarda il C, non esistono in commercio interpreti in grado di offrire un valido supporto al programmatore, al di là dell'apprendimento dei primi rudimenti del linguaggio. L'utilizzo del compilatore è imprescindibile anche per la realizzazione di programmi semplici e "senza troppe pretese"; va osservato, in ogni caso, che compilatore e linker sono strumenti con i quali è possibile raggiungere elevati livelli di efficienza e produttività anche in fase di sviluppo, dopo un breve periodo di familiarizzazione.

DALL'IDEA ALL'APPLICAZIONE

Vale la pena, a questo punto, di descrivere brevemente le varie fasi attraverso le quali l'idea diventa programma eseguibile, attraverso un sorgente C.

In primo luogo occorre analizzare il problema e giungere alla definizione dell'algoritmo, scindendo il problema originale in sottoproblemi di minore complessità. Banale, si tratta dell'unico approccio valido indipendentemente dal linguaggio utilizzato...

---

[5] Un errore di tale genere non può essere individuato dal compilatore, proprio perché si tratta di una funzione esterna al linguaggio, e come tale sconosciuta al compilatore, il quale non può fare altro che segnalarne all'interno dell'object file il nome e il punto di chiamata e scaricare il barile al linker.

[6] Il fatto che il programma funzioni non significa che svolga bene il proprio compito: compilatore e linker non possono individuare errori nella logica del programma. Qui il termine "funzionante" va inteso in senso puramente tecnico.

A questo punto ci si procura un editor, cioè un programma di videoscrittura, piu' o meno sofisticato, in grado di salvare il testo prodotto in formato ASCII puro[7] e si inizia a digitare il programma. "Si inizia" perché può essere utile procedere per piccoli passi, scrivendone alcune parti, compilandole ed eseguendole per controllo... insomma, meglio non mettere troppa carne al fuoco.

Dopo avere scritto una parte di programma di "senso compiuto", tale, cioè, da poter essere compilata e consolidata[8] onde controllarne il funzionamento, si mette da parte l'editor e si dà il file sorgente in pasto (che di solito ha estensione `.C`) al compilatore.

In genere il compilatore provvede a tutte le operazioni necessarie: lancia il preprocessore per effettuare le macrosostituzioni necessarie, compila il sorgente così modificato[9] e, se non vi sono errori, lancia il linker, producendo in output direttamente il file eseguibile.

Nel caso in cui il compilatore segnali errori[10], il linker non viene lanciato e non è prodotto l'object file, che in questo caso sarebbe inutilizzabile. Occorre ricaricare il sorgente nell'editor ed effettuare le correzioni necessarie, tenendo presente che a volte i compilatori vengono fuorviati da errori particolari, che danno origine a molte altre segnalazioni in cascata. E' dunque meglio cominciare a coreggerli a partire dal primo segnalato; è possibile che molti altri scompaiano "da sé". A questo punto può essere nuovamente lanciato il compilatore.

Attenzione, però: il compilatore può segnalare due tipi di errori: gli *error* ed i *warning*. La presenza anche di un solo error in compilazione impedisce sempre l'invocazione del linker: si tratta per lo più di problemi nella sintassi o nella gestione dei tipi di dato per i quali è necessario l'intervento del programmatore. I warning, al contrario, non arrestano il processo e viene pertanto prodotto comunque un file eseguibile. Essi riguardano situazioni di ambiguità che il compilatore può risolvere basandosi sui propri standard, ma che è opportuno segnalare al programmatore, in quanto essi potrebbero essere la manifestazione di situazioni non desiderate, quali, ad esempio, errori di logica. E' raro che l'eseguibile generato in presenza di warning funzioni correttamente, ma non impossibile: alcuni messaggi di warning possono essere tranquillamente ignorati a ragion veduta.

Se gli errori sono segnalati dal linker, è ancora probabile che si debba intervenire sul sorgente, come accennato poco sopra (pag. 4), e quindi lanciare nuovamente il compilatore; in altri casi può trattarsi di problemi di configurazione del linker stesso o di una compilazione effettuata senza indicare le librerie necessarie: è sufficiente lanciare ancora il linker dopo aver eliminato la causa dell'errore.

Il file eseguibile prodotto dal linker ha, in ambiente DOS, estensione `.EXE` o `.COM`. La scelta tra i due tipi di eseguibile dipende, oltre che dalle caratteristiche intrinseche del programma, anche dalle preferenze del programmatore. Avremo occasione di tornare su tali argomenti, esaminando i *modelli di memoria* (pag. 143) e della struttura degli eseguibili (pag. 281 e dintorni).

Come si vede, il tutto non è poi così complicato...

---

[7] Cioè senza caratteri di controllo, formati, e via dicendo.

[8] Tentiamo di sopprimere la tentazione rappresentata dal verbo *linkare*!

[9] Il compilatore riceve in input il testo del sorgente già modificato dal preprocessore, tuttavia il file sorgente non viene alterato.

[10] La segnalazione di errore comprende: il numero di riga (del sorgente) alla quale l'errore è stato rilevato, una sua breve descrizione e l'indicazione della funzione interessata.

I PROGRAMMI C: UN PRIMO APPROCCIO

E' giunto il momento di cominciare ad addentrarsi nei segreti del C. All'approccio tecnico seguito dalla maggior parte dei manuali sull'argomento, è sembrata preferibile un'esposizione la più discorsiva possibile (ma non per questo, almeno nelle intenzioni, approssimativa). Le regole sintattiche saranno presentate sulla base di esempi, semplificati ma comunque realistici. Dedichiamoci dunque al nostro primo programma in C, un ipotetico `CIAO.C`:

```
#include <stdio.h>

void main(void);

void main(void)
{
    printf("Ciao Ciao!\n");
}
```

Il programma non è un gran che, ma dalle sue poche righe possiamo già ricavare un certo numero di informazioni utili.

In C, ogni riga contenente istruzioni o chiamate a funzioni o definizioni di dati si chiude con un punto e virgola, e costituisce una riga logica. Il punto e virgola segnala al compilatore il termine della riga logica, in quanto essa può essere suddivisa in più righe fisiche semplicemente andando a capo.

Nello scrivere un programma C si ha molta libertà nel gestire l'estetica del sorgente: il programma appena visto avrebbe potuto essere scritto così:

```
#include <stdio.h>
void main(void); void main(void) {printf("Ciao Ciao!\n");}
```

oppure, ad esempio:

```
#include <stdio.h>
void main
(void); void
main(void) {printf("Ciao Ciao!\n");}
```

Solo la prima riga deve rimanere isolata[11]; per il resto il compilatore non troverebbe nulla da ridire, ma forse i nostri poveri occhi sì...

Gli "a capo", gli *indent* (rientri a sinistra), le righe vuote, sono semplicemente stratagemmi tipografici ideati per rendere più leggibile il sorgente, e per dare qualche indicazione visiva sulla struttura logica del programma. Un po' di chiarezza è indispensabile; d'altra parte si tratta ormai di vere e proprie convenzioni, seguite dalla grande maggioranza dei programmatori C, addirittura codificate in testi ad esse dedicati. Tuttavia, lo ripetiamo, esse non fanno parte dei vincoli sintattici del linguaggio.

Attenzione: il C è un linguaggio *case-sensitive*, il cui compilatore, cioè, distingue le maiuscole dalle minuscole (a differenza, ad esempio, del Basic). E' vero, dunque, che

```
    printf("Ciao Ciao!\n");
```

potrebbe essere scritta

---

[11] Come da standard ANSI in materia di direttive al preprocessore; ma, per ora, non preoccupiamocene più di tanto (per qualche esempio di direttiva si può vedere pag. 44 e seguenti).

```
printf(
"Ciao Ciao!\n"
);
```

ma non si potrebbe scrivere `PRINTF` o `Printf`: non sarebbe la stessa cosa... Il risultato sarebbe una segnalazione di errore da parte del linker, che non riuscirebbe a trovare la funzione nella libreria.

La prima riga del programma è una direttiva al preprocessore (`#include`): questo inserisce tutto il testo del file `STDIO.H` nel nostro sorgente, a partire dalla riga in cui si trova la direttiva stessa. A cosa serve? Nel file `STDIO.H` ci sono altre direttive al preprocessore e definizioni che servono al compilatore per tradurre correttamente il programma. In particolare, in `STDIO.H` è descritto (vedremo come a pag. 87) il modo in cui la funzione `printf()` si interfaccia al programma che la utilizza. Ogni compilatore C è accompagnato da un certo numero di file `.H`, detti *include file* o *header file*, il cui contenuto è necessario per un corretto utilizzo delle funzioni di libreria (anche le librerie sono fornite col compilatore).

Il nome dell'include file è, in questo caso, racchiuso tra parentesi angolari ("<" e ">"): ciò significa che il preprocessore deve ricercarlo solo nelle directory specificate nella configurazione del compilatore. Se il nome fosse racchiuso tra virgolette (ad esempio: `"MYSTDIO.H"`), il preprocessore lo cercherebbe prima nella directory corrente, e poi in quelle indicate nella configurazione.

Da non dimenticare: le direttive al preprocessore non sono mai chiuse dal punto e virgola.

La riga

```
void main(void);
```

descrive le regole di interfaccia della funzione `main()`. Si noti che al termine della riga c'è il punto e virgola. La riga che segue è apparentemente identica:

```
void main(void)
```

ma in essa il punto e virgola non compare. La differenza è notevole, infatti l'assenza del "`;`" ci segnala che questa riga è l'inizio della *definizione* della funzione `main()`, cioè della parte di programma che costituisce, a tutti gli effetti, la funzione `main()` stessa.

Che cosa sia una funzione, e come lavori, sarà oggetto di accaniti sforzi mentali a pag. 85. Per adesso ci limitiamo ad osservare che dopo la riga che ne descrive l'interfaccia c'è una parentesi graffa aperta: questa segna l'inizio del codice eseguibile della funzione, che si chiude (vedere l'ultima riga del listato) con una graffa chiusa, *non* seguita da alcun punto e virgola.

Tutto quello che sta tra le due graffe è il corpo della funzione (*function body*) e definisce le azioni svolte dalla funzione stessa: può comporsi di istruzioni, di chiamate a funzione, di definizioni di variabili... In pratica ogni funzione può essere in qualche modo paragonata ad un programma a se stante.

La `main()` è una funzione molto particolare: tutti i programmi C devono contenere una ed una sola funzione `main()` e l'esecuzione del programma inizia dalla prima riga di questa; di `main()` si discute con maggiore dettaglio a pag. 105.

Quante istruzioni C sono utilizzate da `CIAO.C`? La risposta è... nemmeno una!

La `#include`, abbiamo detto, è una direttiva al preprocessore, e come tale viene da questo elaborata ed eliminata dal testo sorgente (infatti viene sostituita con il contenuto del file `.H`) passato in input al compilatore.

La descrizione dell'interfaccia, detta anche *prototipo*, di `main()`, informa il compilatore che da qualche parte, nel programma, c'è una funzione `main()` che si comporta in un certo modo: dunque non è un'istruzione.

La definizione di `main()`, a sua volta, in quanto tale non è un'istruzione; semmai ne può contenere. Ma l'unica riga contenuta nella definizione di `main()` è la chiamata alla funzione `printf()`, la quale, essendo una funzione, non è un'istruzione (ovvio, no?). In C, un nome seguito da parentesi tonde aperta e chiusa (eventualmente racchiudenti qualcosa) è una chiamata a funzione.

In particolare, `printf()` è esterna al compilatore, ma fa parte di un gruppo di funzioni inserite nella libreria che accompagnano praticamente tutte le implementazioni esistenti di compilatori C: per questo essa è considerata una funzione standard del C. La funzione `printf()` scrive a video[12] la sequenza di caratteri, racchiusa tra virgolette, specificata tra le parentesi tonde. Una sequenza di caratteri tra virgolette è una *stringa*. Quella dell'esempio si chiude con i caratteri '`\`' e '`n`', che in coppia hanno, nel linguaggio C, un significato particolare: "vai a capo".

In pratica tutte le operazioni di interazione tra i programmi e lo hardware, il *firmware*[13] ed il sistema operativo sono delegate a funzioni (aventi interfaccia più o meno standardizzata) esterne al compilatore, il quale non deve dunque implementare particolari capacità di generazione di codice di I/O, peculiari per il sistema al quale è destinato.

Abbiamo scritto un programma C senza utilizzare quasi nulla del C. Stiamo lavorando in ambiente DOS? Bene, è sufficiente compilarlo per ottenere l'eseguibile. Vogliamo utilizzarlo su una macchina Unix? Non dobbiamo fare altro che trasportare il sorgente su quella macchina e compilarlo nuovamente su di essa...

---

[12] In realtà... non proprio a video, ma siccome l'effetto, salvo particolari condizioni, è quello di veder comparire a video i caratteri, per il momento possiamo accettare questa semplificazione. A pag. 116 tutti i particolari.

[13] Software dedicato alla diagnostica e al pilotaggio dei dispositivi hardware, "scolpito" permanentemente nei microchip dell'elaboratore. Nel caso dei personal computer si parla comunemente di BIOS (Basic Input Output System)

LA GESTIONE DEI DATI IN C

Per poter parlare di come si gestiscono i dati, occorre prima precisare che cosa essi siano, o meglio che cosa si intenda con il termine "dati", non tanto dal punto di vista della logica informatica, quanto piuttosto da quello strettamente tecnico ed operativo.

In tal senso, va innanzitutto osservato che tutto quanto viene elaborato dal microprocessore di un computer deve risiedere nella memoria di questo, la cosiddetta RAM[14], che, al di là della sua implementazione hardware, è una sequenza di bit, ciascuno dei quali, ovviamente, può assumere valore 1 oppure 0. Nella RAM si trova anche il codice macchina eseguibile del programma: semplificando un poco, possiamo dire che tutta la parte di RAM non occupata da quello può rappresentare "dati".

E' evidente che nella maggior parte dei casi un programma non controlla tutta la memoria, ma solo una porzione più o meno ampia di essa; inoltre le regole in base alle quali esso ne effettua la gestione sono codificate all'interno del programma stesso e dipendono, almeno in parte, dal linguaggio utilizzato per scriverlo.

Sintetizzando quanto affermato sin qui, i dati gestiti da un programma sono sequenze di bit situate nella parte di RAM che esso controlla: se il programma vi può accedere in lettura e scrittura, dette sequenze rappresentano le cosiddette "variabili"; se l'accesso può avvenire in sola lettura si parla, invece, di "costanti".

Dal punto di vista del loro significato si apre invece il discorso dei tipi di dato.

I TIPI DI DATO

Al fine di attribuire significato ad una sequenza di bit occorre sapere quanti bit la compongono, e, come vedremo, qual è la loro organizzazione al suo interno. La più ristretta sequenza di bit significativa per le macchine è il *byte*, che si compone di 8 bit[15].

In C, al byte corrisponde il tipo di dato *character*, cioè carattere. Esso può assumere 256 valori diversi ($2^8 = 256$). Si distinguono due tipi di character: il *signed character*, in cui l'ottavo bit funge da indicatore di segno (se è 1 il valore è negativo), e l'*unsigned character*, che utilizza invece tutti gli 8 bit per esprimere il valore, e può dunque esclusivamente assumere valori positivi. Un signed char può variare tra `-128` e `127`, mentre un unsigned char può esprimere valori tra `0` e `255`.

La sequenza di bit di ampiezza immediatamente superiore al byte è detta *word*. Qui il discorso si complica leggermente, perché mentre il byte si compone di 8 bit su quasi tutte le macchine, la dimensione della word dipende dal microprocessore che questa utilizza, e può essere, generalmente, di 16 o 32 bit (l'argomento è ripreso a pagina 461). Nelle pagine che seguono faremo riferimento alla word come ad una sequenza di 16 bit, in quanto è tale la sua dimensione su tutte le macchine che utilizzano i processori Intel 8086 o 8088, o i chips 80286, 80386 e 80486 in modalità reale (cioè compatibile con l'Intel 8086).

Il tipo di dato C corrispondente alla word è l'*integer*, cioè intero. Anche l'integer può essere `signed` o `unsigned`. Dando per scontato, come appena detto, che un integer (cioè una word) occupi 16 bit, i valori estremi del signed integer sono `-32768` e `32767`, mentre quelli dell'unsigned integer sono `0` e `65535`.

---

[14] Random Access Memory, cioè memoria ad accesso casuale, perché il contenuto di ogni sua parte può essere letto o modificato, anche più volte, in qualunque momento.

[15] Non è una verità universale: alcuni processori implementano il byte con 7 bit. Vedere anche pag. 461.

Tra il character e l'integer si colloca lo *short integer*, che può essere, manco a dirlo, `signed` o `unsigned`. Lo short integer occupa 16 bit, perciò stanti le assunzioni sulla dimensione della word, ai nostri fini short integer e integer sono equivalenti.

Per esprimere valori interi di notevole entità il C definisce il *long integer*, che occupa 32 bit. Anche il long integer può essere `signed` o `unsigned`. Nelle macchine in cui la word è di 32 bit, integer e long integer coincidono.

Tutti i tipi sin qui descritti possono rappresentare solo valori interi, e sono perciò detti *integral types*.

In C è naturalmente possibile gestire anche numeri in virgola mobile, mediante appositi tipi di dato[16]: il *floating point*, il *double precision* e il *long double precision*. Il floating point occupa 32 bit ed offre 7 cifre significative di precisione, il double precision occupa 64 bit con 15 cifre di precisione e il long double precision 80 bit[17] con 19 cifre di precisione. Tutti i tipi in virgola mobile sono dotati di segno.

La tabella che segue riassume le caratteristiche dei tipi di dato sin qui descritti.

TIPI DI DATO IN C

TIPO	BIT	VALORI AMMESSI	PRECISIONE
character	8	da `-128` a `127`	-
unsigned character	8	da `0` a `255`	-
short integer	16	da `-32768` a `32767`	-
unsigned short integer	16	da `0` a `65535`	-
integer	16	da `-32768` a `32767`	-
unsigned integer	16	da `0` a `65535`	-
long integer	32	da `-2147483648` a `2147483647`	-
unsigned long integer	32	da `0` a `4294967295`	-
floating point	32	da $3.4*10^{-38}$ a $3.4*10^{38}$	7 cifre
double precision	64	da $1.7*10^{-308}$ a $1.7*10^{308}$	15 cifre
long double precision	80	da $3.4*10^{-4932}$ a $1.1*10^{4932}$	19 cifre

Il C non contempla un tipo di dato "stringa". Le stringhe di caratteri (come `"Ciao Ciao!\n"`) sono gestite come *array* (pag. 29) di character, cioè come sequenze di caratteri che occupano posizioni contigue in memoria ed ai quali è possibile accedere mediante l'indice della loro

[16] I numeri in virgola mobile sono gestiti in formato esponenziale: una parte dei bit sono dedicati alla mantissa, una parte all'esponente ed uno al segno.

[17] Corrisponde alla dimensione dei registri del coprocessore matematico.

posizione. Le stringhe possono anche essere gestite mediante i puntatori (pag. 16 e seguenti); sulle stringhe in particolare si veda pag. 25.

Vi è, infine, un tipo di dato particolare, utilizzabile per esprimere l'assenza di dati o per evitare di specificare a quale tipo, tra quelli appena descritti, appartenga il dato: si tratta del *void type*. Esso può essere utilizzato esclusivamente per dichiarare puntatori void (pag. 34) e funzioni (pag. 87).

LE VARIABILI

E' il momento di ripescare `CIAO.C` e complicarlo un poco.

```
#include <stdio.h>

void main(void);

void main(void)
{
    unsigned int anni;
    float numero;

    anni = 31;
    numero = 15.66;
    printf("Ciao Ciao! Io ho %u anni\n",anni);
    printf("e questo è un float: %f\n",numero);
}
```

Nella nuova versione, `CIAO2.C` abbiamo introdotto qualcosa di molto importante: l'uso delle variabili. Il C consente di individuare una certa area di memoria mediante un nome arbitrario che le viene attribuito con un'operazione detta definizione della variabile; la variabile è ovviamente l'area di RAM così identificata. Ogni riferimento al nome della variabile è in realtà un riferimento al valore in essa contenuto; si noti, inoltre, che nella definizione della variabile viene specificato il tipo di dato associato a quel nome (e dunque contenuto nella variabile). In tal modo il programmatore può gestire i dati in RAM senza conoscerne la posizione e senza preoccuparsi (entro certi limiti) della loro dimensione in bit e dell'organizzazione interna dei bit, cioè del significato che ciascuno di essi assume nell'area di memoria assegnata alla variabile.

Con la riga

```
    unsigned int anni;
```

viene definita una variabile di nome anni e di tipo unsigned integer (intero senza segno): essa occupa perciò una word nella memoria dell'elaboratore e può assumere valori da `0` a `65535`. Va osservato che alla variabile non è associato, per il momento, alcun valore: essa viene inizializzata con la riga

```
    anni = 31;
```

che costituisce un'operazione di assegnazione: il valore `31` è assegnato alla variabile anni; l'operatore "`=`", in C, è utilizzato solo per le assegnazioni (che sono sempre effettuate da destra a sinistra), in quanto per il controllo di una condizione di uguaglianza si utilizza un operatore apposito ("`==`"): per saperne di più, vedere pag. 70.

Tuttavia è possibile inizializzare una variabile contestualmente alla sua dichiarazione:

```
    unsigned int anni = 31;
```

è, in C, un costrutto valido.

Nella definizione di variabili di tipo integral, la parola `int` può essere sempre omessa, eccetto il caso in cui sia "sola":

```
unsigned anni = 31;      // OK! sinonimo di unsigned int
long abitanti;           // OK! sinonimo di long int
valore;                  // ERRORE! il solo nome della variabile NON basta!
```

Dal momento che ci siamo, anche se non c'entra nulla con le variabili, tanto vale chiarire che le due barre "//" introducono un commento, come si deduce dalle dichiarazioni appena viste. Viene considerato commento tutto ciò che segue le due barre, fino al termine della riga. Si possono avere anche commenti multiriga, aperti da "/*" e chiusi da "*/". Ad esempio:

```
/* abbiamo esaminato alcuni esempi
   di dichiarazioni di variabili */
```

Tutto il testo che fa parte di un commento viene ignorato dal compilatore e non influisce sulle dimensioni del programma eseguibile; perciò è bene inserire con una certa generosità commenti chiarificatori nei propri sorgenti. Non è infrequente che un listato, "dimenticato" per qualche tempo, risulti di difficile lettura anche all'autore, soprattutto se questi non ha seguito la regola... KISS, già menzionata a pag. 2. I commenti tra "/*" e "*/" non possono essere nidificati, cioè non si può fare qualcosa come:

```
/* abbiamo esaminato alcuni esempi
   /* alcuni validi e altri no */
   di dichiarazioni di variabili */
```

Il compilatore segnalerebbe strani errori, in quanto il commento è chiuso dal primo "*/" incontrato.

Tornando all'argomento del paragrafo, va ancora precisato che in una riga logica possono essere definite (e, volendo, inizializzate) più variabili, purché tutte dello stesso tipo, separandone i nomi con una virgola:

```
int var1, var2;      // due variabili int, nessuna delle quali inizializzata
char ch1 = 'A', ch2; // due variabili char, di cui solo la prima inizializ.
float num,                     // dichiarazione di 3 float ripartita su 3
      v1,                      // righe fisiche; solo l'ultima variabile
      terzaVar = 12.4;         // e' inizializzata
```

Soffermiamoci un istante sulla dichiarazione dei 3 `float`: la suddivisione in più righe non è obbligatoria, ed ha esclusivamente finalità di chiarezza (dove avremmo sistemato i commenti?). Inoltre, e questo è utile sottolinearlo, l'inizializzazione ha effetto esclusivamente sull'ultima variabile dichiarata, `terzaVar`. Un errore commesso frequentemente dai principianti (e dai distratti) è assegnare un valore ad una sola delle variabili dichiarate, nella convinzione che esso venga assegnato anche a tutte quelle dichiarate "prima". Ebbene, non è così. Ogni variabile deve essere inizializzata esplicitamente, altrimenti essa contiene... già... che cosa? Cosa contiene una variabile non inizializzata? Ai capitoli successivi, ed in particolare a pag. 34 l'ardua sentenza... per il momento, provate a pensarci su.

Inoltre, attenzione alle maiuscole. La variabile `terzaVar` deve essere sempre indicata con la "`V`" maiuscola:

```
int terzavar;        //OK!
char TerzaVar;       //OK!
double terzaVar;     //ERRORE! terzaVar esiste gia'!
```

Non è possibile dichiarare più variabili con lo stesso nome in una medesima funzione (ma in funzioni diverse sì). A rendere differente il nome è sufficiente una diversa disposizione di maiuscole e minuscole.

I nomi delle variabili devono cominciare con una lettera dell'alfabeto o con l'*underscore* ("_") e possono contenere numeri, lettere e underscore. La loro lunghezza massima varia a seconda del

compilatore; le implementazioni commerciali più diffuse ammettono nomi composti di oltre 32 caratteri (vedere pag. 87).

```
double _numVar;
int Variabile_Intera_1;
char 1_carattere;            //ERRORE! il nome inizia con un numero
```

Anche il void type può essere incontrato nelle dichiarazioni: esso può però essere utilizzato solo per dichiarare funzioni o puntatori, ma non comuni variabili; la parola chiave da utilizzare nelle dichiarazioni è `void`.

Per riassumere, ecco l'elenco dei tipi di dato e delle parole chiave da utilizzare per dichiarare le variabili.

TIPI DI DATO E DICHIARATORI

TIPO	**DICHIARATORI VALIDI**
character	`char`
unsigned character	`unsigned char`
short integer	`short int, short`
unsigned short integer	`unsigned short int, unsigned short`
integer	`int`
unsigned integer	`unsigned int, unsigned`
long integer	`long int, long`
unsigned long integer	`unsigned long int, unsigned long`
floating point	`float`
double precision floating point	`double`
long double precision floating point	`long double`
void type	`void`

Un'ultima osservazione: avete notato che nelle stringhe passate a `printf()` sono comparsi strani simboli ("`%u`", "`%f`")? Si tratta di formattatori di campo e indicano a `printf()` come interpretare (e quindi visualizzare) le variabili elencate dopo la stringa stessa. La sequenza "`%u`" indica un intero senza segno, mentre "`%f`" indica un dato di tipo `float`. Un intero con segno si indica con "`%d`", una stringa con "`%s`", un carattere con "`%c`".

Dalle pagine che precedono appare chiaro che la dimensione dell'area di memoria assegnata dal compilatore ad una variabile dipende dall'ingombro in byte del tipo di dato dichiarato. In molti casi può tornare utile sapere quanti byte sono allocati (cioè assegnati) ad una variabile, o a un tipo di dato. Allo scopo è possibile servirsi dell'operatore `sizeof()`, che restituisce come `int` il numero di byte occupato dal tipo di dato o dalla variabile indicati tra le parentesi. Vedere, per un esempio, pag. 68.

I PUNTATORI

Una variabile è un'area di memoria alla quale è associato un nome simbolico, scelto dal programmatore (vedere pag. 13). Detta area di memoria è grande quanto basta per contenere il tipo di dato indicato nella dichiarazione della variabile stessa, ed è collocata dal compilatore, automaticamente, in una parte di RAM non ancora occupata da altri dati. La posizione di una variabile in RAM è detta indirizzo, o *address*. Possiamo allora dire che, in pratica, ad ogni variabile il compilatore associa sempre due valori: il dato in essa contenuto e il suo indirizzo, cioè la sua posizione in memoria.

Gli indirizzi di memoria

Proviamo ad immaginare la memoria come una sequenza di piccoli contenitori, ciascuno dei quali rappresenta un byte: ad ogni "contenitore", talvolta detto "locazione", potremo attribuire un numero d'ordine, che lo identifica univocamente. Se il primo byte ha numero d'ordine 0, allora il numero assegnato ad un generico byte ne individua la posizione in termini di spostamento (*offset*) rispetto al primo byte, cioè rispetto all'inizio della memoria. Così, il byte numero 12445 dista proprio 12445 byte dal primo, il quale, potremmo dire, dista 0 byte da se stesso. L'indirizzamento (cioè l'accesso alla memoria mediante indirizzi) avviene proprio come appena descritto: ogni byte è accessibile attraverso il suo offset rispetto ad un certo punto di partenza, il quale, però, non necessariamente è costituito dal primo byte di memoria in assoluto. Vediamo perché.

Nella CPU del PC sono disponibili alcuni byte, organizzati come vere e proprie variabili, dette registri (*register*). La CPU è in grado di effettuare elaborazioni unicamente sui valori contenuti nei propri registri (che si trovano fisicamente al suo interno e non nella RAM); pertanto qualunque valore oggetto di elaborazione deve essere "caricato", cioè scritto, negli opportuni registri. Il risultato delle operazioni compiute dalla CPU deve essere conservato, se necessario, altrove (tipicamente nella RAM), al fine di lasciare i registri disponibili per altre elaborazioni.

Anche gli indirizzi di memoria sono soggetti a questa regola.

I registri del processore Intel 8086 si compongono di 16 bit ciascuno, pertanto il valore massimo che essi possono esprimere è quello dell'integer, cioè `65535` (esadecimale `FFFF`): il massimo offset gestibile dalla CPU permette dunque di indirizzare una sequenza di 65536 byte (compreso il primo, che ha offset pari a 0), corrispondenti a 64Kb.

Configurazioni di RAM superiori (praticamente tutte) devono perciò essere indirizzate con uno stratagemma: in pratica si utilizzano due registri, rispettivamente detti registro di segmento (*segment register*) e registro di offset (*offset register*). Segmento e offset vengono solitamente indicati in notazione esadecimale, utilizzando i due punti ("`:`") come separatore, ad esempio `045A:10BF`. Ma non è tutto.

Se segmento e offset venissero semplicemente affiancati, si potrebbero indirizzare al massimo 128Kb di RAM: infatti si potrebbe avere un offset massimo di 65535 byte a partire dal byte numero 65535. Quello che occorre è invece un valore in grado di numerare, o meglio di indirizzare, almeno 1Mb: i fatidici 640Kb, ormai presenti su tutte le macchine in circolazione, più gli indirizzi riservati al BIOS e alle schede adattatrici[18]. Occorre, in altre parole, un indirizzamento a 20 bit[19].

Questo si ottiene sommando al segmento i 12 bit più significativi dell'offset, ed accodando i 4 bit rimanenti dell'offset stesso: tale tecnica consente di trasformare un indirizzo segmento:offset in un

[18] Ma perché proprio tale suddivisione? Perché gli indirizzi superiori al limite dei 640Kb sono stati riservati, proprio in sede di progettazione del PC, al firmware BIOS e al BIOS delle schede adattatrici (video, rete, etc.), sino al limite di 1 Mb.

[19] In effetti, 2^{20} = 1.048.576: provare per credere. Già che ci siamo, vale la pena di dire che le macchine basate su chip 80286 o superiore possono effettuare indirizzamenti a 21 bit: i curiosi possono leggere i particolari a pagina 226.

indirizzo lineare[20]. L'indirizzo *seg:off* di poco fa (`045A:10BF`) corrisponde all'indirizzo lineare `0565F`, infatti `045A+10B = 565` (le prime 3 cifre di un valore esadecimale di 4 cifre, cioè a 16 bit, corrispondono ai 12 bit più significativi).

Complicato? Effettivamente... Ma dal momento che le cose stanno proprio così, tanto vale adeguarsi e cercare di padroneggiare al meglio la situazione. In fondo è anche questione di abitudine.

*Gli operatori * e &*

Il C consente di pasticciare a volontà, ed anche... troppo, con gli indirizzi di memoria mediante particolari strumenti, detti puntatori, o *pointers*.

Un puntatore non è altro che una normalissima variabile contenente un indirizzo di memoria. I puntatori non rappresentano un tipo di dato in sé, ma piuttosto sono tipizzati in base al tipo di dato a cui... puntano, cioè di cui esprimono l'indirizzo. Perciò essi sono dichiarati in modo del tutto analogo ad una variabile di quel tipo, anteponendo però al nome del puntatore stesso l'operatore "`*`", detto operatore di indirezione (*dereference operator*).

Così, la riga

```
int unIntero;
```

dichiara una variabile di tipo `int` avente nome `unIntero`, mentre la riga

```
int *puntaIntero;
```

dichiara un puntatore a `int` avente nome `puntaIntero` (il puntatore ha nome `puntaIntero`, non l'`int`... ovvio!). E' importante sottolineare che si tratta di un puntatore a integer: il compilatore C effettua alcune operazioni sui puntatori in modo automaticamente differenziato a seconda del tipo che il puntatore indirizza[21], ma è altrettanto importante non dimenticare mai che un puntatore contiene semplicemente un indirizzo (o meglio un valore che viene gestito dal compilatore come un indirizzo). Esso indirizza, in altre parole, un certo byte nella RAM; la dichiarazione del tipo "puntato" permette al compilatore di "capire" di quanti byte si compone l'area che inizia a quell'indirizzo e come è organizzata al proprio interno, cioè quale significato attribuire ai singoli bit (vedere pag. 11).

Si possono dichiarare più puntatori in un'unica riga logica, come del resto avviene per le variabili: la riga seguente dichiare tre puntatori ad intero.

```
int *ptrA, *ptrB, *ptrC;
```

Si noti che l'asterisco, o meglio, l'operatore di indirezione, è ripetuto davanti al nome di ogni puntatore. Se non lo fosse, tutti i puntatori dichiarati senza di esso sarebbero in realtà... normalissime variabili di tipo `int`. Ad esempio, la riga che segue dichiara due puntatori ad intero, una variabile intera, e poi ancora un puntatore ad intero.

```
int *ptrA, *ptrB, unIntero, *intPtr;
```

Come si vede, la dichiarazione mista di puntatori e variabili è un costrutto sintatticamente valido; occorre, come al solito, prestare attenzione a ciò che si scrive se si vogliono evitare errori logici piuttosto insidiosi. Detto tra noi, principianti e distratti sono i più propensi a dichiarare correttamente il

---

[20] Per indirizzo lineare si intende un offset relativo all'inizio della memoria, cioè relativo al primo byte della RAM. Al riguardo si veda anche pag. 23.

[21] Di *aritmetica dei puntatori* si parla a pag. 33.

primo puntatore e privare tutti gli altri dell'asterisco nella convinzione che il tipo dichiarato sia `int*`. In realtà, una riga di codice come quella appena riportata dichiara una serie di oggetti di tipo `int`; è la presenza o l'assenza dell'operatore di indirezione a stabilire, singolarmente per ciascuno di essi, se si tratti di una variabile o di un puntatore.

Mediante l'operatore `&` (detto "indirizzo di", o *address of*) è possibile, inoltre, conoscere l'indirizzo di una variabile:

```
float numero;      // dichiara una variabile float
float *numPtr;     // dichiara un puntatore ad una variabile float

numero = 12.5;     // assegna un valore alla variabile
numPtr = &numero;  // assegna al puntatore l'indirizzo della variabile
```

E' chiaro il rapporto tra puntatori e variabili? Una variabile contiene un valore del tipo della dichiarazione, mentre un puntatore contiene l'indirizzo, cioè la posizione in memoria, di una variabile che a sua volta contiene un dato del tipo della dichiarazione. Dopo le operazioni dell'esempio appena visto, `numPtr` non contiene `12.5`, ma l'indirizzo di memoria al quale `12.5` si trova.

Anche un puntatore è una variabile, ma contiene un valore che non rappresenta un dato di un particolare tipo, bensì un indirizzo. Anche un puntatore ha il suo bravo indirizzo, ovviamente. Riferendosi ancora all'esempio precedente, l'indirizzo di `numPtr` può essere conosciuto con l'espressione `&numPtr` e risulta sicuramente diverso da quello di `numero`, cioè dal valore contenuto in `numPtr`. Sembra di giocare a rimpiattino...

Proviamo a confrontare le due dichiarazioni dell'esempio:

```
float numero;
float *numPtr;
```

Esse sono fortemente analoghe; del resto abbiamo appena detto che la dichiarazione di un puntatore è identica a quella di una comune variabile, ad eccezione dell'asterisco che precede il nome del puntatore stesso. Sappiamo inoltre che il nome attribuito alla variabile identifica un'area di memoria che contiene un valore del tipo dichiarato: ad esso si accede mediante il nome stesso della variabile, cioè il simbolo che, nella dichiarazione, si trova a destra della parola chiave che indica il tipo, come si vede chiaramente nell'esempio che segue.

```
printf("%f\n",numero);
```

L'accesso al valore della variabile avviene nella modalità appena descritta non solo in lettura, ma anche in scrittura:

```
numero = 12.5;
```

Cosa troviamo a destra dell'identificativo di tipo in una dichiarazione di puntatore? Il nome preceduto dall'asterisco. Ma allora anche il nome del puntatore con l'asterisco rappresenta un valore del tipo dichiarato... Provate ad immaginare cosa avviene se scriviamo:

```
printf("%f\n",*numPtr);
```

La risposta è: `printf()` stampa il valore di `numero`[22]. In altre parole, l'operatore di indirezione non solo differenzia la dichiarazione di un puntatore da quella di una variabile, ma consente anche di accedere al contenuto della variabile (o, più in generale, della locazione di memoria) indirizzata dal puntatore. Forse è opportuno, a questo punto, riassumere il tutto con qualche altro esempio.

---

[22] Presupposto fondamentale è l'assegnazione a `numPtr` dell'indirizzo di numero, come da esempio.

```
float numero = 12.5;
float *numPtr = &numero;
```

Sin qui nulla di nuovo[23]. Supponiamo ora che l'indirizzo di `numero` sia, in esadecimale, `FFE6` e che quello di `numPtr` sia `FFE4`: non ci resta che giocherellare un po' con gli operatori address of ("`&`") e dereference ("`*`")...

```
printf("numero = %f\n",numero);
printf("numero = %f\n",*numPtr);
printf("l'indirizzo di numero e' %X\n",&numero);
printf("l'indirizzo di numero e' %X\n",numPtr);
printf("l'indirizzo di numPtr e' %X\n",&numPtr);
```

L'output prodotto è il seguente:

```
numero = 12.5
numero = 12.5
l'indirizzo di numero è FFE6
l'indirizzo di numero è FFE6
l'indirizzo di numPtr è FFE4
```

Le differenza tra le varie modalità di accesso al contenuto e all'indirizzo delle veriabili dovrebbe ora essere chiarita. Almeno, questa è la speranza. Tra l'altro abbiamo imparato qualcosa di nuovo su `printf()`: per stampare un intero in formato esadecimale si deve inserire nella stringa, invece di `%d`, `%X` se si desidera che le cifre `A-F` siano visualizzate con caratteri maiuscoli, `%x` se si preferiscono i caratteri minuscoli.

Va osservato che è prassi usuale esprimere gli indirizzi in notazione esadecimale. A prima vista può risultare un po' scomodo, ma, operando in tal modo, la logica di alcune operazioni sugli indirizzi stessi (e sui puntatori) risulta sicuramente più chiara. Ad esempio, ogni cifra di un numero esadecimale rappresenta quattro bit in memoria: si è già visto (pag. 17) come ciò permetta di trasformare un indirizzo segmentato nel suo equivalente lineare con grande facilità. Per la cronaca, tale operazione è detta anche "normalizzazione" dell'indirizzo (o del puntatore): avremo occasione di riparlarne (pag. 21).

Vogliamo complicarci un poco la vita? Eccovi alcune interessanti domandine, qualora non ve le foste ancora posti...

a) Quale significato ha l'espressione `*&numPtr`?
b) Quale significato ha l'espressione `**numPtr`?
c) E l'espressione `*numero`?
d) E l'espressione `&*numPtr`?
e) `&*numPtr` e `numPtr` sono la stessa cosa?
f) Cosa restituisce l'espressione `&&numero`?
g) E l'espressione `&&numPtr`?

---

[23] Attenzione, però, all'istruzione

```
float *numPtr = &numero;
```

Può infatti sembrare in contrasto con quanto affermato sin qui l'assegnazione di un un indirizzo (`&numero`) ad una indirezione (`*numPtr`) che non rappresenta un indirizzo, ma un `float`. In realtà va osservato che l'istruzione riportata assegna a `numPtr` un valore contestualmente alla dichiarazione di questo: l'operatore di indirezione, qui, serve unicamente a indicare che `numPtr` è un puntatore; esso deve cioè essere considerato parte della sintassi necessaria alla dichiarazione del puntatore e non come strumento per accedere a ciò che il puntatore stesso indirizza. Alla luce di tale considerazione l'assegnazione appare perfettamente logica.

h) Cosa accade se si esegue `*numPtr = 21.75`?
i) Cosa accade se si esegue `numPtr = 0x24A6`?
j) E se si esegue `&numPtr = 0xAF2B`?

Provate a rispondere prima di leggere le risposte nelle righe che seguono!

a) Una cosa alla volta. `&numPtr` esprime l'indirizzo di `numPtr`. L'indirezione di un indirizzo (cioè l'asterisco davanti a "qualcosa" che esprime un indirizzo) restituisce il valore memorizzato a quell'indirizzo. Pertanto, `*&numPtr` esprime il valore memorizzato all'indirizzo di `numPtr`. Cioè il valore contenuto in `numPtr`. Cioè l'indirizzo di `numero`. Simpatico, vero?

b) Nessuno! Infatti `*numPtr` esprime il valore memorizzato all'indirizzo puntato da `numPtr`, cioè il valore di `numero`. Applicare una indirezione (il primo asterisco) a detto valore non ha alcun senso, perché il contenuto di `numero` non è un indirizzo. Per di più `numero` è un `float`, mentre gli indirizzi sono sempre numeri interi. Il compilatore ignora l'asterisco di troppo.

c) Nessuno! Di fatto, si ricade nel caso precedente, poiché `*numPtr` equivale a `numero` e `**numPtr`, pertanto, equivale a `*numero`.

d) Allora: `numPtr` esprime l'indirizzo di `numero`, quindi la sua indirezione `*numPtr` rappresenta il contenuto di `numero`. L'indirizzo del contenuto di `numero` è... l'indirizzo di `numero`, quindi `&*numPtr` equivale a `numPtr` (e a `*&numPtr`). Buffo...

e) Evidentemente sì, come si vede dalla risposta precedente.

f) `&&numero` restituisce... una segnalazione d'errore del compilatore. Infatti `&numero`, espressione lecita, rappresenta l'indirizzo di `numero`, ma che senso può avere parlare dell'indirizzo dell'indirizzo di `numero`? Attenzione: `numPtr` contiene l'indirizzo di `numero`, ma l'indirizzo dell'indirizzo di `numero` non può essere considerato sinonimo dell'indirizzo di `numPtr`.

g) Anche `&&numPtr` è un'espressione illecita. L'indirizzo dell'indirizzo di una variabile (puntatore o no) non esiste...

h) Viene modificato il contenuto di `numero`. Infatti `numPtr` rappresenta l'indirizzo di `numero`, cioè punta all'area di memoria assegnata a `numero`; `*numPtr` rappresenta `numero` nel senso che restituisce il contenuto dell'area di memoria occupata da `numero`. Un'operazione effettuata sull'indirezione di un puntatore è sempre, a tutti gli effetti, effettuata sulla locazione di memoria a cui esso punta.

i) Si assegna un nuovo valore a `numPtr`, questo è evidente. L'effetto (forse meno evidente a prima vista) è che ora `numPtr` non punta più a numero, ma a ciò che si trova all'indirizzo `0x24A6`. Qualsiasi cosa sia memorizzata a quell'indirizzo, se referenziata mediante `numPtr` (cioè mediante la sua indirezione), viene trattata come se fosse un `float`.

j) Si ottiene, ancora una volta, un errore in compilazione. Infatti `&numPtr` restituisce l'indirizzo di `numPtr`, il quale, ovviamente, non può essere modificato in quanto stabilito dal compilatore.

Complicazioni

I puntatori sono, dunque, strumenti appropriati alla manipolazione ad alto livello degli indirizzi delle variabili. C'è proprio bisogno di preoccuparsi dei registri della CPU e di tutte le contorsioni possibili tra indirizzi seg:off e indirizzi lineari? Eh, sì... un poco è necessario; ora si tratta di capire il perché.

Poco fa abbiamo ipotizzato che l'indirizzo di `numero` e di `numPtr` fossero, rispettivamente, `FFE6` e `FFE4`. A prescindere dai valori, realisitci ma puramente ipotetici, è interessante notare che si tratta di due `unsigned int`. In effetti, per visualizzarli correttamente, abbiamo passato a `printf()` stringhe contenenti `%X`, lo specificatore di formato per gli interi in formato esadecimale. Che significa tutto ciò?

Significa che il valore memorizzato in `numPtr` (e in qualsiasi altro puntatore[24]) è una word, occupa 16 bit e si differenzia da un generico intero senza segno per il solo fatto che esprime un indirizzo di memoria. E' evidente, alla luce di quanto appena affermato, che l'indirizzo memorizzato in `numPtr` è un offset: come tutti i valori a 16 bit esso è gestito dalla CPU in uno dei suoi registri e può variare tra 0 e 65535. Un puntatore come `numPtr` esprime allora, in byte, la distanza di una variabile da... che cosa? Dall'indirizzo contenuto in un altro registro della CPU, gestito automaticamente dal compilatore.

Con qualche semplificazione possiamo dire che il compilatore, durante la traduzione del sorgente in linguaggio macchina, stabilisce quanto spazio il programma ha a disposizione per gestire i propri dati e a quale distanza dall'inizio del codice eseguibile deve avere inizio l'area riservata ai dati. Dette informazioni sono memorizzate in una tabella, collocata in testa al file eseguibile, che il sistema operativo utilizza per caricare l'opportuno valore in un apposito registro della CPU. Questo registro contiene la parte segmento dell'indirizzo espresso da ogni puntatore dichiarato come `numPtr`.

Nella maggior parte dei casi l'esistenza dei registri di segmento è del tutto trasparente al programmatore, il quale non ha alcun bisogno di proccuparsene, in quanto compilatore, linker e sistema operativo svolgono automaticamente tutte le operazioni necessarie alla loro gestione. Nello scrivere un programma è di solito sufficiente lavorare con i puntatori proprio come abbiamo visto negli esempi che coinvolgono `numero` e `numPtr`: gli operatori "`*`" e "`&`" sono caratterizzati da una notevole potenza operativa.

Puntatori far e huge

Le considerazioni sin qui espresse, però, aprono la via ad alcuni approfondimenti. In primo luogo, va sottolineato ancora una volta che `numPtr` occupa 16 bit di memoria, cioè 2 byte, proprio come qualsiasi `unsigned int`. E ciò è valido anche se il tipo di `numero`, la variabile puntata, è il `float`, che ne occupa 4. In altre parole, un puntatore occupa sempre lo spazio necessario a contenere l'indirizzo del dato puntato, e non il tipo di dato; tutti i puntatori come `numPtr`, dunque, occupano 2 byte, indipendentemente che il tipo di dato puntato sia un `int`, piuttosto che un `float`, o un `double`... Una semplice verifica empirica può essere effettuata con l'aiuto dell'operatore `sizeof()` (vedere pag. 68).

```
int unIntero;
long unLongInt;
float unFloating;
double unDoublePrec;

int *intPtr;
long *longPtr;
float *floatPtr;
double *doublePtr;

printf("intPtr:    %d bytes (%d)\n",sizeof(intPtr),sizeof(int *));
printf("longPtr:   %d bytes (%d)\n",sizeof(longPtr),sizeof(long *));
printf("floatPtr:  %d bytes (%d)\n",sizeof(floatPtr),sizeof(float *));
printf("doublePtr: %d bytes (%d)\n",sizeof(doublePtr),sizeof(double *));
```

Tutte le `printf()` visualizzano due volte il valore 2, che è appunto la dimensione in byte di un generico puntatore. L'esempio mostra, tra l'altro, come `sizeof()` possa essere applicato sia al tipo di dato che al nome di una variabile (in questo caso dei puntatori); se ne trae, infine, che il tipo di un puntatore è dato dal tipo di dato puntato, seguito dall'asterisco.

[24] Ciò vale se si lascia che il compilatore lavori basandosi sui propri default. Torneremo sull'argomento in tema di modelli di memoria (pag. 143).

Tutti i puntatori come numPtr, dunque, gestiscono un offset da un punto di partenza automaticamente fissato dal sistema operativo in base alle caratteristiche del file eseguibile. E' possibile in C, allora, gestire indirizzi lineari, o quanto meno comprensivi di segmento ed offset? La risposta è sì. Esistono due parole chiave, dette modificatori di tipo, che consentono di dichiarare puntatori speciali, in grado di gestire sia la parte segmento che la parte offset di un indirizzo di memoria: si tratta di far e huge.

```
double far *numFarPtr;
```

La riga di esempio dichiara un puntatore far a un dato di tipo double. Per effetto del modificatore far, numFarPtr è un puntatore assai differente dal numPtr degli esempi precedenti: esso occupa 32 bit di memoria, cioè 2 word, ed è pertanto equivalente ad un long int. Di conseguenza numFarPtr è in grado di esprimere tanto la parte offset di un indirizzo (nei 2 byte meno significativi), quanto la parte segmento (nei 2 byte più significativi[25]). La parte segmento è utilizzata dalla CPU per caricare l'opportuno registro di segmento, mentre la parte offset è gestita come al solito: in tal modo un puntatore far può esprimere un indirizzo completo del tipo segmento:offset e indirizzare dati che si trovano al di fuori dell'area dati assegnata dal sistema operativo al programma.

Ad esempio, se si desidera che un puntatore referenzi l'indirizzo 596A:074B, lo si può dichiarare ed inizializzare come segue:

```
double far *numFarPtr = 0x596A074B;
```

Per visualizzare il contenuto di un puntatore far con printf() si può utilizzare un formattatore speciale:

```
printf("numFarPtr = %Fp\n",numFarPtr);
```

Il formattatore %Fp forza printf() a visualizzare il contenuto di un puntatore far proprio come segmento ed offset, separati dai due punti:

```
numFarPtr = 596A:074B
```

è l'output prodotto dalla riga di codice appena riportata.

Abbiamo appena detto che un puntatore far rappresenta un indirizzo seg:off. E' bene... ripeterlo qui, sottolineando che quell'indirizzo, in quanto seg:off, non è un indirizzo lineare. Parte segmento e parte offset sono, per così dire, indipendenti, nel senso che la prima è considerata costante, e la seconda variabile. Che significa? la riga

```
char far *vPtr = 0xB8000000;
```

dichiara un puntatore far a carattere e lo inizializza all'indirizzo B800:0000; la parte offset è nulla, perciò il puntatore indirizza il primo byte dell'area che ha inizio all'indirizzo lineare B8000 (a 20 bit). Il secondo byte ha offset pari a 1, perciò può essere indirizzato incrementando di 1 il puntatore, portandolo al valore 0xB8000001. Incrementando ancora il puntatore, esso assume valore 0xB8000002 e punta al terzo byte. Sommando ancora 1 al puntatore, e poi ancora 1, e poi ancora... si giunge ad un valore particolare, 0xB800FFFF, corrispondente all'indirizzo B800:FFFF, che è proprio quello del byte

[25] I processori Intel memorizzano i valori in RAM con la tecnica *backwords*, cioè a "parole rovesciate". Ciò significa che i byte più significativi di ogni valore sono memorizzati ad indirizzi di memoria superiori: ad esempio il primo byte di una word (quello composto dai primi 8 bit) è memorizzato nella locazione di memoria successiva a quella in cui si trova il secondo byte (bit 8-15), che contiene la parte più importante (significativa) del valore.

avente offset 65535 rispetto all'inizio dell'area. Esso è l'ultimo byte indirizzabile mediante un comune puntatore *near*[26]. Che accade se si incrementa ancora vPtr? Contrariamente a quanto ci si potrebbe attendere, la parte offset si riazzera senza che alcun "riporto" venga sommato alla parte segmento. Insomma, il puntatore si "riavvolge" all'inizio dell'area individuata dall'indirizzo lineare rappresentato dalla parte segmento con uno 0 alla propria destra (che serve a costruire l'indirizzo a 20 bit). Ora si comprende meglio (speriamo!) che cosa si intende per parte segmento e parte offset separate: esse sono utilizzate proprio per caricare due distinti registri della CPU e pertanto sono considerate indipendenti l'una dall'altra, così come lo sono tra loro tutti i registri del microprocessore.

Tutto ciò ha un'implicazione estremamente importante: con un puntatore far è possibile indirizzare un dato situato ad un qualunque indirizzo nella memoria disponibile entro il primo Mb, ma non è possibile "scostarsi" dall'indirizzo lineare espresso dalla parte segmento oltre i 64Kb. Per fare un esempio pratico, se si intende utilizzare un puntatore far per gestire una tabella, la dimensione complessiva di questa non deve eccedere i 64Kb.

Tale limitazione è superata tramite il modificatore huge, che consente di avere puntatori in grado di indirizzare linearmente tutta la memoria disponibile (sempre entro il primo Mb). La dichiarazione di un puntatore huge non presenta particolarità:

```
int huge *iHptr;
```

Il segreto dei puntatori huge consiste in alcune istruzioni assembler che il compilatore introduce di soppiatto nei programmi tutte le volte che il valore del puntatore viene modificato o utilizzato, e che ne effettuano la normalizzazione. Con tale termine si indica un semplice calcolo che consente di esprimere l'indirizzo seg:off come rappresentazione di un indirizzo lineare: in modo, cioè, che la parte offset sia variabile unicamente da 0 a 15 (F esadecimale) ed i riporti siano sommati alla parte segmento. In pratica si tratta di sommare alla parte segmento i 12 bit più significativi della parte offset, con una tecnica del tutto analoga a quella utilizzata a pag. 17. Riprendiamo l'esempio precedente, utilizzando questa volta un puntatore huge.

```
char huge *vhugePtr = 0xB8000000;
```

L'inizializzazione del puntatore huge, come si vede, è identica a quella del puntatore far. Incrementando di 1 il puntatore si ottiene il valore 0xB8000001, come nel caso precedente. Sommando ancora 1 si ha 0xB8000002, e poi 0xB8000003, e così via. Sin qui, nulla di nuovo. Al quindicesimo incremento il puntatore vale 0xB800000F, come nel caso del puntatore far.

Ma al sedicesimo incremento si manifesta la differenza: il puntatore far assume valore 0xB8000010, mentre il puntatore huge vale 0xB8010000: la parte segmento si è azzerata ed il 16 sottratto ad essa ha prodotto un riporto[27] che è andato ad incrementare di 1 la parte segmento. Al trentunesimo incremento il puntatore far vale 0xB800001F, mentre quello huge è 0xB801000F. Al trentaduesimo incremento il puntatore far diventa 0xB8000020, mentre quello huge vale 0xB8020000.

Il meccanismo dovrebbe essere ormai chiaro, così come il fatto che le prime 3 cifre della parte offset di un puntatore huge sono sempre 3 zeri. Fingiamo per un attimo di non vederli: la parte segmento e la quarta cifra della parte offset rappresentano proprio un indirizzo lineare a 20 bit.

La normalizzazione effettuata dal compilatore consente di gestire indirizzi lineari pur caricando in modo indipendente parte segmento e parte offset in registri di segmento e, rispettivamente, di offset

[26] Si dicono near i puntatori non far e non huge; quelli, in altre parole, che esprimono semplicemente un offset rispetto ad un registro di segmento della CPU.

[27] Nella numerazione esadecimale, cioè in base 16, si calcola un riporto ogni 16 unità, e non ogni 10 come invece avviene nella numerazione in base decimale.

della CPU; in tal modo, con un puntatore `huge` non vi sono limiti né all'indirizzo di partenza, né alla quantità di memoria indirizzabile a partire da quell'indirizzo. Naturalmente ciò ha un prezzo: una piccola perdita di efficienza del codice eseguibile, introdotta dalla necessità di eseguire la routine di normalizzazione prima di utilizzare il valore del puntatore.

Ancora una precisazione: nelle dichiarazioni multiple di puntatori `far` e `huge`, il modificatore deve essere ripetuto per ogni puntatore dichiarato, analogamente a quanto occorre per l'operatore di indirezione. L'omissione del modificatore determina la dichiarazione di un puntatore "offset" a 16 bit.

```
    long *lptr, far *lFptr, lvar, huge *lHptr;
```

Nell'esempio sono dichiarati, nell'ordine, il puntatore a `long` a 16 bit `lptr`, il puntatore `far` a `long` `lFptr`, la variabile `long` `lvar` e il puntatore `huge` a `long` `lHptr`.

E' forse il caso di sottolineare ancora che la dichiarazione di un puntatore riserva spazio in memoria esclusivamente per il puntatore stesso, e non per una variabile del tipo di dato indirizzato. Ad esempio, la dichiarazione

```
    long double far *dFptr;
```

alloca, cioè riserva, 32 bit di RAM che potranno essere utilizzate per contenere l'indirizzo di un `long double`, i cui 80 bit dovranno essere allocati con un'operazione a parte[28].

Tanto per confondere un poco le idee, occorre precisare un ultimo particolare. I sorgenti C possono essere compilati, tramite particolari opzioni riconosciute dal compilatore, in modo da applicare differenti criteri di default alla gestione dei puntatori. In particolare, vi sono modalità di compilazione che trattano tutti i puntatori come variabili a 32 bit, eccetto quelli esplicitamente dichiarati `near`. Ne riparleremo a pagina 143, descrivendo i modelli di memoria.

Per il momento è il caso di accennare a tre macro, definite in `DOS.H`, che agevolano in molti casi la manipolazione dei puntatori a 32 bit, siano essi `far` o `huge`: si tratta di `MK_FP()`, che "costruisce" un puntatore a 32 bit dati un segmento ed un offset entrambi a 16 bit, di `FP_SEG()`, che estrae da un puntatore a 32 bit i 16 bit esprimenti la parte segmento e di `FP_OFF()`, che estrae i 16 bit esprimenti l'offset. Vediamole al lavoro:

```
#include <dos.h>
    ....
    unsigned farPtrSeg;
    unsigned farPtrOff;
    char far *farPtr;
    ....
    farPtr = (char far *)MK_FP(0xB800,0);              // farPtr punta a B800:0000
    farPtrSeg = FP_SEG(farPtr);                       // farPtrSeg contiene 0xB800
    farPtrOff = FP_OFF(farPtr);                            // farPtrOff contiene 0
```

Le macro testè descritte consentono di effettuare facilmente la normalizzzione di un puntatore, cioè trasformare l'indirizzo in esso contenuto in modo tale che la parte offset non sia superiore a `0Fh`:

```
    char far *cfPtr;
    char huge *chPtr;
    ....
    chPtr = (char huge *)(((long)FP_SEG(cfPtr)) << 16)+
                     (((long)(FP_OFF(cfPtr) >> 4)) << 16)+(FP_OFF(cfPtr) & 0xF);
```

[28] Ad esempio con la dichiarazione di una variabile `long double` o con una chiamata ad una delle funzioni di libreria dedicate all'allocazione dinamica della memoria (pag. 109). Non siate impazienti...

Come si vede, dalla parte offset sono scartati i 4 bit meno significativi: i 12 bit più significativi sono sommati al segmento; dalla parte offset sono poi scartati i 12 bit più significativi e i 4 bit restanti sono sommati al puntatore. Circa il significato degli operatori di *shift* `<<` e `>>` vedere pag. 69; l'operatore `&` (che in questo caso non ha il significato di *address of*, ma di *and su bit*) è descritto a pag. 71.

L'indirizzo lineare corrispondente all'indirizzo segmentato espresso da un puntatore `huge` può essere ricavato come segue:

```
    char huge *chPtr;
    long linAddr;
    ....
    linAddr = ((((((long)FP_SEG(chPtr)) << 16)+(FP_OFF(chPtr) << 12)) >> 12) &
                                                                    0xFFFFFL);
```

Per applicare tale algoritmo ad un puntatore `far` è necessario che questo sia dapprima normalizzato come descritto in precedenza.

E' facile notare che due puntatori `far` possono referenziare il medesimo indirizzo pur contenendo valori a 32 bit differenti, mentre ciò non si verifica con i puntatori normalizzati, nei quali segmento e offset sono sempre gestiti in modo univoco: ne segue che solamente i confronti tra puntatori `huge` (o normalizzati) garantiscono risultati corretti.

P u n t a t o r i   s t a t i c

La dichiarazione

```
    static float *ptr;
```

dichiara un puntatore `static` a un dato di tipo `float`. In realtà non è possibile, nel dichiarare un puntatore, indicare che esso indirizza un dato `static` essendo questo un modificatore della visibilità delle variabili, e non già del loro tipo. Si veda anche quanto detto a pagina 100.

L e   s t r i n g h e

A pagina 13 abbiamo anticipato che non esiste, in C, il tipo di dato "stringa". Queste sono gestite dal compilatore come sequenze di caratteri, cioè di dati di tipo `char`. Un metodo comunemente utilizzato per dichiarare e manipolare stringhe nei programmi è offerto proprio dai puntatori, come si vede nel programma dell'esempio seguente, che visualizza "`Ciao Ciao!`" e porta a capo il cursore.

```
#include <stdio.h>

char *string = "Ciao";

void main(void)
{
    printf(string);
    printf(" %s!\n",string);
}
```

La dichiarazione di `string` può apparire, a prima vista, anomala. Si tratta infatti, a tutti gli effetti, della dichiarazione di un puntatore e la stranezza consiste nel fatto che a questo non è assegnato un indirizzo di memoria, come ci si potrebbe aspettare, bensì una costante stringa. Ma è proprio questo l'artificio che consente di gestire le stringhe con normali puntatori a carattere: il compilatore, in realtà, assegna a `string`, puntatore a 16 bit, l'indirizzo della costante `"Ciao"`. Dunque la word occupata da

`string` non contiene la parola `"Ciao"`, ma i 16 bit che esprimono la parte offset del suo indirizzo. A sua volta, `"Ciao"` occupa 5 byte di memoria. Proprio 5, non si tratta di un errore di stampa: i 4 byte necessari a memorizzare i 4 caratteri che compongono la parola, più un byte, nel quale il compilatore memorizza il valore binario `0`, detto terminatore di stringa o *null terminator*. In C, tutte le stringhe sono chiuse da un null terminator, ed occupano perciò un byte in più del numero di caratteri "stampabili" che le compongono.

La prima chiamata a `printf()` passa quale argomento proprio `string`: dunque la stringa parametro indispensabile di `printf()` non deve essere necessariamente una stringa di formato quando l'unica cosa da visualizzare sia proprio una stringa. Lo è, però, quando devono essere visualizzati caratteri o numeri, o stringhe formattate in un modo particolare, come avviene nella seconda chiamata.

Qui va sottolineato che per visualizzare una stringa con `printf()` occore fornirne l'indirizzo, che nel nostro caso è il contenuto del puntatore `string`. Se `string` punta alla stringa `"Ciao"`, che cosa restituisce l'espressione `*string`? La tentazione di rispondere `"Ciao"` è forte, ma se così fosse perché per visualizzare la parola occorre passare a `printf()` `string` e non `*string`? Il problema non si poneva con gli esempi precedenti, perché tutti i puntatori esaminati indirizzavano un unico dato di un certo tipo. Con le dichiarazioni

```
float numero = 12.5;
float *numPtr = &numero;
```

si definisce il puntatore `numPtr` e lo si inizializza in modo che contenga l'indirizzo della variabile `numero`, la quale, in fondo proprio come `string`, occupa più di un byte. In questo caso, però, i 4 byte di `numero` contengono un dato unitariamente considerato. In altre parole, nessuno dei 4 byte che la compongono ha significato in sé e per sé. Con riferimento a `string`, al contrario, ogni byte è un dato a sé stante, cioè un dato di tipo `char`: bisogna allora precisare che un puntatore indirizza sempre il primo byte di tutti quelli che compongono il tipo di dato considerato, se questi sono più d'uno. Se ne ricava che `string` contiene, in realtà, l'indirizzo del primo carattere di `"Ciao"`, cioè la `'C'`. Allora `*string` non può che restituire proprio quella, come si può facilmente verificare con la seguente chiamata a `printf()`:

```
printf("%c è il primo carattere...\n",*string);
```

Non dimentichiamo che le stringhe sono, per il compilatore C, semplici sequenze di `char`: la stringa del nostro esempio inizia con il `char` che si trova all'indirizzo contenuto in `string` (la `'C'`) e termina con il primo byte nullo incontrato ad un indirizzo uguale o superiore a quello (in questo caso il byte che segue immediatamente la `'o'`).

Per accedere ai caratteri che seguono il primo è sufficiente incrementare il puntatore o, comunque, sommare ad esso una opportuna quantità (che rappresenta l'offset, cioè lo spostamento, dall'inizo della stringa stessa). Vediamo, come al solito, un esempio:

```
int i = 0;

while(*(string+i) != 0) {
    printf("%c\n",*(string+i));
    ++i;
}
```

L'esempio si basa sull'aritmetica dei puntatori (pag. 33), cioè sulla possibilità di accedere ai dati memorizzati ad un certo offset rispetto ad un indirizzo sommandovi algebricamente numeri interi. Il ciclo visualizza la stringa `"Ciao"` in senso verticale. Infatti l'istruzione `while` (finalmente una "vera" istruzione C!) esegue le istruzioni comprese tra le parentesi graffe finché la condizione espressa tra le parentesi tonde è vera (se questa è falsa la prima volta, il ciclo non viene mai eseguito; vedere pag. 79): in questo caso la `printf()` è eseguita finché il byte che si trova all'indirizzo contenuto in `string`

aumentato di i unità è diverso da 0, cioè finché non viene incontrato il null terminator. La printf() visualizza il byte a quello stesso indirizzo e va a capo. Il valore di i è inizialmente 0, pertanto nella prima iterazione l'indirizzo espresso da string non è modificato, ma ad ogni loop i è incrementato di 1 (tale è il significato dell'operatore ++, vedere pag. 64), pertanto ad ogni successiva iterazione l'espressione string+i restituisce l'indirizzo del byte successivo a quello appena visualizzato. Al termine, i contiene il valore 4, che è anche la lunghezza della stringa: questa è infatti convenzionalmente pari al numero dei caratteri stampabili che compongono la stringa stessa; il null terminator non viene considerato. In altre parole la lunghezza di una stringa è inferiore di 1 al numero di byte che essa occupa effettivamente in memoria. La lunghezza di una stringa può quindi essere calcolata così:

```
    unsigned i = 0;

    while(*(string+i))
        ++i;
```

La condizione tra parentesi è implicita: non viene specificato alcun confronto. In casi come questo il compilatore assume che il confronto vada effettuato con il valore 0, che è proprio quel che fa al nostro caso. Inoltre, dato che il ciclo si compone di una sola riga (l'autoincremento di i), le graffe non sono necessarie (ma potrebbero essere utilizzate ugualmente)[29].

Tutta questa chiacchierata dovrebbe avere reso evidente una cosa: quando ad una funzione viene passata una costante stringa, come in

```
    printf("Ciao!\n");
```

il compilatore, astutamente, memorizza la costante da qualche parte (non preoccupiamoci del "dove", per il momento) e ne passa l'indirizzo.

Inoltre, il metodo visto poco fa per "prelevare" uno ad uno i caratteri che compongono una stringa vale anche nel caso li si voglia modificare:

```
char *string = "Rosso\n";

void main(void)
{
    printf(string);
    *(string+3) = 'p';
    printf(string);
}
```

Il programma dell'esempio visualizza dapprima la parola "Rosso" e poi "Rospo". Si noti che il valore di string non è mutato: esso continua a puntare alla medesima locazione di memoria, ma è mutato il contenuto del byte che si trova ad un offset di 3 rispetto a quell'indirizzo. Dal momento che l'indirezione di un puntatore a carattere restituisce un carattere, nell'assegnazione della lettera 'p' è necessario esprimere quest'ultima come un char, e pertanto tra apici (e non tra virgolette). La variabile string è dichiarata all'esterno di main(): a pag. 39 scoprirete perché.

E' possibile troncare una stringa? Sì, basta inserire un NULL dove occorre:

```
    *(string+2) = NULL;
```

---

[29] Detto tra noi, esiste un metodo più comodo per conoscere la lunghezza di una stringa: la funzione di libreria strlen(), che accetta quale parametro l'indirizzo di una stringa e restituisce, come intero, la lunghezza della medesima (escluso, dunque, il null terminator).

A questo punto una chiamata a `printf()` visualizzerebbe la parola `"Ro"`. `NULL` è una costante manifesta (vedere pag. 44) definita in `STDIO.H`, e rappresenta lo zero binario; infatti la riga di codice precedente potrebbe essere scritta così:

```
*(string+2) = 0;
```

E' possibile allungare una stringa? Sì, basta... essere sicuri di avere spazio a disposizione. Se si sovrascrive il `NULL` con un carattere, la stringa si allunga sino al successivo `NULL`. Occorre fare alcune considerazioni: in primo luogo, tale operazione ha senso, di solito, solo nel caso di concatenamento di stringhe (quando cioè si desidera accodare una stringa ad un'altra per produrne una sola, più lunga). In secondo luogo, se i byte successivi al `NULL` sono occupati da altri dati, questi vengono perduti, sovrascritti dai caratteri concatenati alla stringa: l'effetto può essere disastroso. In effetti esiste una funzione di libreria concepita appositamente per concatenare le stringhe: la `strcat()`, che richiede due stringhe quali parametri. L'azione da essa svolta consiste nel copiare i byte che compongono la seconda stringa, `NULL` terminale compreso, in coda alla prima stringa, sovrascrivendone il `NULL` terminale.

In una dichiarazione come quella di `string`, il compilatore riserva alla stringa lo spazio strettamente necessario a contenere i caratteri che la compongono, più il `NULL`. E' evidente che concatenare a `string` un'altra stringa sarebbe un grave errore (peraltro non segnalato dal compilatore, perché esso lascia il programmatore libero di gestire la memoria come crede: se sbaglia, peggio per lui). Allora, per potere concatenare due stringhe senza pericoli occorre riservare in anticipo lo spazio necessario a contenere la prima stringa e la seconda... una in fila all'altra. Affronteremo il problema parlando di array (pag. 29) e di allocazione dinamica della memoria (pag. 109).

Avvertenza: una dichiarazione del tipo:

```
char *sPtr;
```

riserva in memoria lo spazio sufficiente a memorizzare il puntatore alla stringa, e non una (ipotetica) stringa. I byte allocati sono 2 se il puntatore è, come nell'esempio, `near`; mentre sono 4 se è `far` o `huge`. In ogni caso va ricordato che prima di copiare una stringa a quell'indirizzo bisogna assolutamente allocare lo spazio necessario a contenerla e assegnarne l'indirizzo a `sPtr`. Anche a questo proposito occorre rimandare gli approfondimenti alle pagine in cui esamineremo l'allocazione dinamica della memoria (pag. 109).

E' meglio sottolineare che le librerie standard del C comprendono un gran numero di funzioni (dichiarate in `STRING.H`) per la manipolazione delle stringhe, che effettuano le più svariate operazioni: copiare stringhe o parte di esse (`strcpy()`, `strncpy()`), concatenare stringhe (`strcat()`, `strncat()`), confrontare stringhe (`strcmp()`, `stricmp()`), ricercare sottostringhe o caratteri all'interno di stringhe (`strstr()`, `strchr()`, `strtok()`)... insomma, quando si deve trafficare con le stringhe vale la pena di consultare il manuale delle librerie e cercare tra le funzioni il cui nome inizia con "`str`": forse la soluzione al problema è già pronta.

Gli array

Un *array* (o vettore) è una sequenza di dati dello stesso tipo, sistemati in memoria... in fila indiana. Una stringa è, per definizione, un array di `char`. Si possono dichiarare array di `int`, di `double`, o di qualsiasi altro tipo. Il risultato è, in pratica, riservare in memoria lo spazio necessario a contenere un certo numero di variabili di quel tipo. In effetti, si può pensare ad un array anche come ad un gruppo di variabili, aventi tutte identico nome ed accessibili, quindi, referenziandole attraverso un indice. Il numero di "variabili" componenti l'array è indicato nella dichiarazione:

```
int iArr[15];
```

La dichiarazione di un array è analoga a quella di una variabile, ad eccezione del fatto che il nome dell'array è seguito dal numero di elementi che lo compongono, racchiuso tra parentesi quadre. Quella dell'esempio forza il compilatore a riservare lo spazio necessario a memorizzare 15 interi, dunque 30 byte. Per accedere a ciascuno di essi occorre sempre fare riferimento al nome dell'array, `iArr`: il singolo `int` desiderato è individuato da un indice tra parentesi quadre, che ne indica la posizione.

```
iArr[0] = 12;
iArr[1] = 25;
for(i = 2; i < 15; i++)
    iArr[i] = i;
for(i = 0; i < 15;) {
    printf("iArr[%d] = %d\n",i,iArr[i]);
    i++;
}
```

Nell'esempio i primi due elementi dell'array sono inizializzati a `12` e `25`, rispettivamente. Il primo ciclo `for` inizializza i successivi elementi (dal numero `2` al numero `14`) al valore che `i` assume ad ogni iterazione. Il secondo ciclo `for` visualizza tutti gli elementi dell'array. Di `for` ci occuperemo a pag. 81. Qui preme sottolineare che gli elementi di un array sono numerati a partire da 0 (e non da 1), come ci si potrebbe attendere. Dunque, l'ultimo elemento di un array ha indice inferiore di 1 rispetto al numero di elementi in esso presenti. Si vede chiaramente che gli elementi di `iArr`, dichiarato come array di 15 interi, sono referenziati con indice che va da `0` a `14`.

Che accade se si tenta di referenziare un elemento che non fa parte dell'array, ad esempio `iArr[15]`? Il compilatore non fa una grinza: `iArr[15]` può essere letto e scritto tranquillamente... E' ovvio che nel primo caso (lettura) il valore letto non ha alcun significato logico ai fini del programma, mentre nel secondo caso (scrittura) si rischia di perdere (sovrascrivendolo) qualche altro dato importante. Anche questa volta il compilatore si limita a mettere a disposizione del programmatore gli strumenti per gestire la memoria, senza preoccuparsi di controllarne più di tanto l'operato. Per il compilatore, `iArr[15]` è semplicemente la word che si trova a 30 byte dall'indirizzo al quale l'array è memorizzato. Che farne, è affare del programmatore[30].

Un array, come qualsiasi altro oggetto in memoria, ha un indirizzo. Questo è individuato e scelto dal compilatore. Il programmatore non può modificarlo, ma può conoscerlo attraverso il nome dell'array stesso, usandolo come un puntatore. In C, il nome di un array equivale, a tutti gli effetti, ad un puntatore all'area di memoria assegnata all'array. Pertanto, le righe di codice che seguono sono tutte lecite:

```
int *iPtr;

printf("indirizzo di iArr: %X\n",iArr);
iPtr = iArr;
printf("indirizzo di iArr: %X\n",iPtr);
printf("primo elemento di iArr: %d\n",*iArr);
printf("secondo elemento di iArr: %d\n",*(iArr+1));
++iPtr;
printf("secondo elemento di iArr: %d\n",*iPtr);
```

mentre non sono lecite le seguenti:

```
++iArr;          // l'indirizzo di un array non puo' essere modificato
iArr = iPtr;     // idem
```

ed è lecita, ma inutilmente complessa, la seguente:

[30] Insomma, il programmatore dovrebbe sempre ricordare la regola KISS (pag. 2), il compilatore, da parte sua, applica con tenacia la regola MYOB (*Mind Your Own Business*, fatti gli affari tuoi).

```
    iPtr = &iArr;
```

in quanto il nome dell'array ne restituisce, di per se stesso, l'indirizzo, rendendo inutile l'uso dell'operatore `&` (address of).

Il lettore attento dovrebbe avere notato che l'indice di un elemento di un array ne esprime l'offset, in termini di numero di elementi, dal primo elemento dell'array stesso. In altre parole, il primo elemento di un array ha offset `0` rispetto a se stesso; il secondo ha offset `1` rispetto al primo; il terzo ha offset `2`, cioè dista 2 elementi dal primo...

Banale? Mica tanto. Il compilatore "ragiona" sugli arrays in termini di elementi, e non di byte. Riprenderemo l'argomento tra breve (pag. 33).

Ripensando alle stringhe, appare ora evidente che esse non sono altro che array di `char`. Si differenziano solo per l'uso delle virgolette; allora il problema del concatenamento di stringhe può essere risolto con un array:

```
char string[100];
```

Nell'esempio abbiamo così a disposizione 100 byte in cui copiare e concatenare le nostre stringhe.

Puntatori ed array hanno caratteristiche fortemente simili. Si differenziano perché ad un array non può essere assegnato un valore[31], e perché un array riserva direttamente, come si è visto, lo spazio necessario a contenere i suoi elementi. Il numero di elementi deve essere specificato con una costante. Non è mai possibile utilizzare una variabile. Con una variabile, utilizzata come indice, si può solo accedere agli elementi dell'array dopo che questo è stato dichiarato.

Gli array, se dichiarati al di fuori di qualsiasi funzione[32], possono essere inizializzati:

```
int iArr[] = {12,25,66,0,144,-2,26733};
char string[100] = {'C','i','a','o'};
float fArr[] = {1.44,,0.3};
```

Per inizializzare un array contestualmente alla dichiarazione bisogna specificare i suoi elementi, separati da virgole e compresi tra parentesi graffe aperta e chiusa. Se non si indica tra le parentesi quadre il numero di elementi, il compilatore lo desume dal numero di elementi inizializzati tra le parentesi graffe. Se il numero di elementi è specificato, e ne viene inizializzato un numero inferiore, tutti quelli "mancanti" verranno inizializzati a `0` dal compilatore. Analoga regola vale per gli elementi "saltati" nella lista di inizializzazione: l'array `fArr` contiene 3 elementi, aventi valore `1.44`, `0.0` e `0.3` rispettivamente.

Su `string` si può effettuare una concatenazione come la seguente senza rischi:

```
    strcat(string," Pippo");
```

La stringa risultante, infatti, è `"Ciao Pippo"`, che occupa 11 byte compreso il `NULL` terminale: sappiamo però di averne a disposizione 100.

Sin qui si è parlato di array monodimensionali, cioè di array ogni elemento dei quali è referenziabile mediante un solo indice. In realtà, il C consente di gestire array multidimensionali, nei quali per accedere ad un elemento occorre specificarne più "coordinate". Ad esempio:

```
        int iTab[3][6];
```

dichiara un array a 2 dimensioni, rispettivamente di 3 e 6 elementi. Per accedere ad un singolo elemento bisogna, allo stesso modo, utilizzare due indici:

---

[31] Si possono assegnare valori solo ai suoi elementi.

[32] Il perché sarà chiarito a pag. 34 e seguenti.

```
int i, j, iTab[3][6];

for(i = 0; i < 3; ++i)
    for(j = 0; j < 6; ++j)
        iTab[i][j] = 0;
```

Il frammento di codice riportato dichiara l'array bidimensionale `iTab` e ne inizializza a `0` tutti gli elementi. I due cicli `for` sono nidificati, il che significa che le iterazioni previste dal secondo vengono compiute tutte una volta per ogni iterazione prevista dal primo. In tal modo vengono "percorsi" tutti gli elementi di `iTab`. Infatti il modo in cui il compilatore C alloca lo spazio di memoria per gli array multidimensionali garantisce che per accedere a tutti gli elementi nella stessa sequenza in cui essi si trovano in memoria, è l'indice più a destra quello che deve variare più frequentemente.

E' evidente, d'altra parte, che la memoria è una sequenza di byte: ciò implica che pur essendo `iTab` uno strumento che consente di rappresentare molto bene una tabella di 3 righe e 6 colonne, tutti i suoi elementi stanno comunque "in fila indiana". Pertanto, l'inizializzazione di un array multidimensionale contestuale alla sua dichiarazione può essere effettuata come segue:

```
int *tabella[2][5] = {{3, 2, 0, 2, 1},{3, 0, 0, 1, 0}};
```

Gli elementi sono elencati proprio nell'ordine in cui si trovano in memoria; dal punto di vista logico, però, ogni gruppo di elementi nelle coppie di graffe più interne rappresenta una riga. Dal momento che, come già sappiamo, il C è molto elastico nelle regole che disciplinano la stesura delle righe di codice, la dichiarazione appena vista può essere spezzata su due righe, al fine di rendere ancora più evidente il parallelismo concettuale tra un array bidimensionale ed una tabella a doppia entrata:

```
int *tabella[2][5] = {{3, 2, 0, 2, 1},
                      {3, 0, 0, 1, 0}};
```

Si noti che tra le parentesi quadre, inizializzando l'array contestualmente alla dichiarazione, non è necessario specificare entrambe le dimensioni, perché il compilatore può desumere quella mancante dal computo degli elementi inizializzati: nella dichiarazione dell'esempio sarebbe stato lecito scrivere `tabella[][5]` o `tabella[2][]`.

Dalle affermazioni fatte discende infoltre che gli elementi di un array bidimensionale possono essere referenziati anche facendo uso di un solo indice:

```
int *iPtr;

iPtr = tabella;
for(i = 0; i < 2*5; i++)
    printf("%d\n",iPtr[i];
```

In genere i compilatori C sono in grado di gestire array multidimensionali senza un limite teorico (a parte la disponibilità di memoria) al numero di dimensioni. E' tuttavia infrequente, per gli utilizzi più comuni, andare oltre la terza dimensione.

L'aritmetica dei puntatori

Quanti byte di memoria occupa un array? La risposta dipende, ovviamente, dal numero degli elementi e dal tipo di dato dichiarato. Un array di 20 interi occupa 40 byte, dal momento che ogni `int` ne occupa 2. Un array di 20 `long` ne occupa, dunque, 80. Calcoli analoghi occorrono per accedere ad uno qualsiasi degli elementi di un array: il terzo elemento di un array di `long` ha indice 2 e dista 8 byte (2*4) dall'inizio dell'area di RAM riservata all'array stesso. Il quarto elemento di un array di `int` dista 3*2 = 6 byte dall'inizio dell'array. Generalizzando, possiamo affermare che un generico elemento di un array di un

qualsiasi tipo dista dall'indirizzo base dell'array stesso un numero di byte pari al prodotto tra il proprio indice e la dimensione del tipo di dato.

Fortunatamente il compilatore C consente di accedere agli elementi di un array in funzione di un unico parametro: il loro indice[33]. Per questo sono lecite e significative istruzioni come quelle già viste:

```
    iArr[1] = 12;
    printf("%X\n",iArr[j]);
```

E' il compilatore ad occuparsi di effettuare i calcoli sopra descritti per ricavare il giusto offset in termini di byte di ogni elemento, e lo fa in modo trasparente al programmatore per qualsiasi tipo di dato.

Ciò vale anche per le stringhe (o array di caratteri). Il fatto che ogni `char` occupi un byte semplifica i calcoli ma non modifica i termini del problema[34].

E' importante sottolineare che quanto affermato vale non solo nei confronti degli array, bensì di qualsiasi puntatore, come può chiarire l'esempio che segue.

```
#include <stdio.h>

int iArr[]= {12,99,27,0};

void main(void)
{
    int *iPtr;

    iPtr = iArr;                                // punta al primo elemento di iArr[]
    while(*iPtr) {                          // finche' l'int puntato da iPtr non e' 0
        printf("%X -> %d\n",iPtr,*iPtr);           // stampa iPtr e l'intero puntato
        ++iPtr;                                                   // incrementa iPtr
    }
}
```

Il trucco sta tutto nell'espressione `++iPtr`: l'incremento del puntatore è automaticamente effettuato dal compilatore sommando 2 al valore contenuto in `iPtr`, proprio perché esso è un puntatore ad `int`, e l'`int` occupa 2 byte. In altre parole, `iPtr` è incrementato, ad ogni iterazione, in modo da puntare all'intero successivo.

Si noti che l'aritmetica dei puntatori è applicata dal compilatore ogni volta che una grandezza intera è sommata a (o sottratta da) un puntatore, moltiplicando tale grandezza per il numero di byte occupati dal tipo di dato puntato.

Questo modo di gestire i puntatori ha due pregi: da un lato evita al programmatore lo sforzo di pensare ai dati in memoria in termini di numero di byte; dall'altro consente la portabilità dei programmi che fanno uso di puntatori anche su macchine che codificano gli stessi tipi di dato con un diverso numero di bit.

Un'ultima precisazione: ai putatori possono essere sommate o sottratte solo grandezze intere (`int` o `long`, a seconda che si tratti di puntatori `near` o no).

Puntatori a puntatori

Un puntatore è una variabile che contiene un indirizzo. Perciò è lecito (e, tutto sommato, abbastanza normale) fare uso di puntatori che puntano ad altri puntatori. La dichiarazione di un puntatore a puntatore si effettua così:

---

[33] Che esprime, in definitiva, la loro posizione diminuita di uno.

[34] Inoltre alcuni compilatori consentono di gestire `char` di tipo multibyte, che occupano una word.

```
char **pPtr;
```

In pratica occorre aggiungere un asterisco, in quanto siamo ad un secondo livello di indirezione: `pPtr` non punta direttamente ad un `char`; la sua indirezione `*pPtr` restituisce un altro puntatore a `char`, la cui indirezione, finalmente, restituisce il `char` agognato. Presentando i puntatori è stato analizzato il significato di alcune espressioni (pag. 20); in particolare si è detto che in `**numPtr`, ove `numPtr` è un puntatore a `float`, il primo "`*`" è ignorato: l'affermazione è corretta, perché pur essendo `numPtr` e `pPtr` entrambi puntatori, il secondo punta ad un altro puntatore, al quale può essere validamente applicato il primo dereference operator ("`*`").

L'ambito di utilizzo più frequente dei puntatori a puntatori è forse quello degli array di stringhe: dal momento che in C una stringa è di per sé un array (di `char`), gli array di stringhe sono gestiti come array di puntatori a `char`. A questo punto è chiaro che il nome dell'array (in C il nome di un array è anche puntatore all'array stesso) è un puntatore a puntatori a `char`[35]. Pertanto, ad esempio,

```
    printf(pPtr[2]);
```

visualizza la stringa puntata dal terzo elemento di `pPtr` (con una semplificazione "umana" ma un po' pericolosa potremmo dire che viene visualizzata la terza stringa dell'array).

Puntatori void

Un puntatore può essere dichiarato di tipo `void`. Si tratta di una pratica poco diffusa, avente lo scopo di lasciare indeterminato il tipo di dato che il puntatore indirizza, sino al momento dell'inizializzazione del puntatore stesso. La forma della dichiarazione è intuibile:

```
void *ptr, far *fvptr;
```

Ad un puntatore void può essere assegnato l'indirizzo di qualsiasi tipo di dato.

---

[35] Va osservato che un array di puntatori a carattere potrebbe essere anche dichiarato così:

```
    char *stringhe[10];
```

La differenza consiste principalmente nella necessità di indicare al momento della dichiarazione il numero di puntatori a `char` contenuti nell'array stesso, e nella possibilità di inizializzare l'array:

```
char *stringhe[] = {
    "prima stringa",
    "seconda stringa",
    NULL,
    "quarta ed ultima stringa",
};
```

L'array `stringhe` comprende 4 stringhe di caratteri, o meglio 4 puntatori a `char`: proprio da questo deriva la possibilità di utilizzare `NULL` come elemento dell'array (`NULL`, lo ripetiamo, è una costante manifesta definita in `STDIO.H`, e vale uno zero binario). In pratica, il terzo elemento dell'array è un puntatore che non punta ad aluna stringa.

L'ACCESSIBILITÀ E LA DURATA DELLE VARIABILI

In C le variabili possono essere classificate, oltre che secondo il tipo di dato, in base alla loro accessibilità e alla loro durata. In particolare, a seconda del contesto in cui sono dichiarate, le variabili di un programma C assumono per default determinate caratteristiche di accessibilità e durata; in molti casi, però, queste possono essere modificate mediante l'utilizzo di apposite parole chiave applicabili alla dichiarazione delle variabili stesse.

Per comprendere i concetti di accessibilità (o visibilità) e durata, va ricordato che una variabile altro non è che un'area di memoria, grande quanto basta per contenere un dato del tipo indicato nella dichiarazione, alla quale il compilatore associa, per comodità del programmatore, il nome simbolico da questi scelto.

In termini generali, possiamo dire che la durata di una variabile si estende dal momento in cui le viene effettivamente assegnata un'area di memoria fino a quello in cui quell'area è riutilizzata per altri scopi.

Dal punto di vista dell'accessibilità ha invece rilevanza se sia o no possibile leggere o modificare, da parti del programma diverse da quella in cui la variabile è stata dichiarata, il contenuto dell'area di RAM riservata alla variabile stessa

Cerchiamo di mettere un po' d'ordine...

Le variabili automatic

Qualsiasi variabile dichiarata all'interno di un blocco di codice racchiuso tra parentesi graffe (generalmente all'inizio di una funzione) appartiene per default alla classe *automatic*. Non è dunque necessario, anche se è possibile farlo, utilizzare la parola chiave `auto`. La durata e la visibilità della variabile sono entrambe limitate al blocco di codice in cui essa è dichiarata. Se una variabile è dichiarata in testa ad una funzione, essa esiste (cioè occupa memoria) dal momento in cui la funzione inizia ad essere eseguita, sino al momento in cui la sua esecuzione termina.

Le variabili automatic, dunque, non occupano spazio di memoria se non quando effettivamente servono; inoltre, essendo accessibili esclusivamente dall'interno di quella funzione, non vi è il rischio che possano essere modificate accidentalmente da operazioni svolte in funzioni diverse su variabili aventi medesimo nome: in un programma C, infatti, più variabili automatic possono avere lo stesso nome, purché dichiarate in blocchi di codice diversi. Se i blocchi sono nidificati (cioè uno è interamente all'interno di un altro) ciò è ancora vero, ma la variabile dichiarata nel blocco interno "nasconde" quella dichiarata con identico nome nel blocco esterno (quando, ovviamente, viene eseguito il blocco interno).

Vediamo un esempio:

```
#include <stdio.h>

void main(void)
{
    int x = 1;
    int y = 10;
    {
        int x = 2;

        printf("%d, %d\n",x,y);
    }
    printf("%d, %d\n",x,y);
}
```

La variabile `x` dichiarata in testa alla funzione `main()` è inizializzata a `1`, mentre la `x` dichiarata nel blocco interno è inizializzata a `2`. L'output del programma é:

```
2, 10
1, 10
```

Ciò prova che la "prima" `x` esiste in tutta la funzione `main()`, mentre la "seconda" esiste ed è visibile solo nel blocco più interno; inoltre, dal momento che le due variabili hanno lo stesso nome, nel blocco interno la prima `x` è resa non visibile dalla seconda. La `y`, invece, è visibile anche nel blocco interno.

Se si modifica il programma dell'esempio come segue:

```
#include <stdio.h>

void main(void)
{
    int x = 1;
    int y = 10;
    {
        int x = 2;
        int z = 20;

        printf("%d, %d\n",x,y);
    }
    printf("%d, %d\n",x,y,z);
}
```

il compilatore non porta a termine la compilazione e segnala l'errore con un messaggio analogo a "`undefined symbol z in function main()`" a significare che la seconda `printf()` non può referenziare la variabile `z`, poiche questa cessa di esistere al termine del blocco interno di codice.

La gestione delle variabili automatic è dinamica. La memoria necessaria è allocata alla variabile esclusivamente quando viene eseguito il blocco di codice (tipicamente una funzione) in cui essa è dichiarata, e le viene "sottratta" non appena il blocco termina. Ciò implica che non è possibile conoscere il contenuto di una variabile automatic prima che le venga esplicitamente assegnato un valore da una istruzione facente parte del blocco (vedere pag. 14): a beneficio dei distratti, vale la pena di evidenziare che una variabile automatica può essere utilizzata in lettura prima di essere inizializzata[36], ma il valore in essa contenuto è casuale e, pertanto, inutilizzabile nella quasi totalità dei casi.

E' opportuno sottolineare che mentre le variabili dichiarate nel blocco più esterno di una funzione (cioè in testa alla stessa) esistono e sono visibili (salvo il caso di variabili con lo stesso nome) in tutti i blocchi interni di quella funzione, nel caso di funzioni diverse nessuna di esse può accedere alle variabili automatiche delle altre.

Le variabili register

Dal momento che il compilatore colloca le variabili automatic nella RAM del calcolatore, i valori in esse contenuti devono spesso essere copiati nei registri della CPU per poter essere elaborati e, se modificati dall'elaborazione subita, copiati nuovamente nelle locazioni di memoria di provenienza. Tali operazioni sono svolte in modo trasparente per il programmatore, ma possono deteriorare notevolmente la performance di un programma, soprattutto se ripetute più e più volte (ad esempio all'interno di un ciclo con molte iterazioni).

Dichiarando una variabile automatic con la parola chiave `register` si forza il compilatore ad allocarla direttamente in un registro della CPU, con notevole incremento di efficienza nell'elaborazione del valore in essa contenuto. Ecco un esempio:

[36] Il compilatore, però, si degna di emettere un apposito warning.

```
register int i = 10;

do {
    printf("%2d\n",i);
} while(i--);
```

Il ciclo visualizza, incolonnati[37], i numeri da `10` a `0`; la variabile `i` si comporta come una qualsiasi variabile automatic, ma essendo probabilmente gestita in un registro consente un'elaborazione più veloce. E' d'obbligo scrivere "probabilmente gestita" in quanto non si può essere assolutamente certi che il compilatore collochi una variabile dichiarata con `register` proprio in un registro della CPU: in alcune situazioni potrebbe gestirla come una variabile automatic qualsiasi, allocandola in memoria. I principali motivi sono due: la variabile potrebbe occupare più byte di quanti compongono un registro della CPU[38], o potrebbero non esserci registri disponibili allo scopo[39].

Già che ci siamo, diamo un'occhiata più approfondita all'esempio di poco fa. Innanzitutto va rilevato che nella dichiarazione di `i` potrebbe essere omessa la parola chiave `int` (vedere pag. 14):

```
register i = 10;
```

Abbiamo poi utilizzato un costrutto nuovo: il ciclo `do...while`. Esso consente di identificare un blocco di codice (quello compreso tra le graffe) che viene eseguito finché la condizione specificata tra parentesi dopo la parola chiave `while` continua ad essere vera. Il ciclo viene sempre eseguito almeno una volta, perché il test è effettuato al termine del medesimo (pag. 80). Nel nostro caso, quale test viene effettuato? Dal momento che non è utilizzato alcun operatore di confronto esplicito (pag. 70), viene controllato se il risultato dell'espressione nelle tonde è diverso da `0`. L'operatore `--`, detto di autodecremento (pag. 64), è specificato dopo la variabile a cui è applicato. Ciò assicura che `i` sia decrementata dopo l'effettuazione del test. Perciò il ciclo è eseguito 11 volte, con `i` che varia da `10` a `0` inclusi. Se l'espressione fosse `--i`, il decremento sarebbe eseguito prima del test, con la conseguenza che per `i` pari a `0` il ciclo non verrebbe più eseguito.

Come per le variabili automatic, non è possibile conoscere il contenuto di una variabile `register` prima della sua esplicita inizializzazione mediante un operazione di assegnamento. In questo caso non si tratta di utilizzo e riutilizzo di un'area di memoria, ma di un registro macchina: non possiamo conoscerne a priori il contenuto nel momento in cui esso è destinato alla gestione della variabile (dichiarazione della variabile). Inoltre, analogamente alle variabili automatic, anche quelle `register`

---

[37] Nella stringa di formato passata a `printf()`, il 2 che compare tra l'indicatore di carattere di formato ("`%`") e il carattere di formato stesso ("`d`") serve quale specificatore dell'ampiezza di campo. In altre parole esso indica che il valore contenuto nella variabile `i` deve essere visualizzato in uno spazio ampio 2 caratteri, assicurando così il corretto incolonnamento dei numeri visualizzati.

[38] Le macchine 8086, 8088 e 80286 dispongono esclusivamente di registri a 16 bit. Una dichiarazione come la seguente:

```
register long rLong;
```

non potrebbe che originare, su tali macchine, una normale variabile automatic, perché il tipo long integer occupa 32 bit. Su macchine 80386 (e superiori), invece, il compilatore potrebbe gestire `rLong` in un registro, dal momento che detti elaboratori dispongono di registri a 32 bit (a seconda del compilatore, però, potrebbe essere necessario specificare esplicitamente in fase di compilazione che si desidera generare codice eseguibile specifico per macchine 80386).

[39] In effetti, il numero dei registri macchina è limitato. E' quindi opportuno identificare le variabili più utilizzate e dichiararle per prime come `register`: il compilatore alloca le variabili nell'ordine in cui sono dichiarate.

cessano di esistere all'uscita del blocco di codice (solitamente una funzione) nel quale sono dichiarate e il registro macchina viene utilizzato per altri scopi.

Le variabili `register`, a differenza delle automatic, non hanno indirizzo: ciò appare ovvio se si pensa che i registri macchina si trovano nella CPU e non nella RAM. La conseguenza immediata è che una variabile `register` non può mai essere referenziata tramite un puntatore. Nel nostro esempio, il tentativo di assegnare ad un puntatore l'indirizzo di `i` provocherebbe accorate proteste da parte del compilatore.

```
    register i;
    int *iPtr = &i;                                   // errore! i non ha indirizzo
```

Pur non avendo indirizzo, le variabili `register` possono contenere un indirizzo, cioè un puntatore: la dichiarazione

```
    register char *ptr_1, char *ptr_2;
```

non solo è perfettamente lecita, ma anzi genera, se possibile, due puntatori (a carattere) particolarmente efficienti.

Le variabili static

Una variabile è `static` se dichiarata utilizzando, appunto, la parola chiave `static`:

```
    static float sF, *sFptr;
```

Nell'esempio sono dichiarate due variabili `static`: una di tipo `float` e un puntatore (`static` anch'esso) ad un `float`.

Come nel caso delle variabili automatic, quelle `static` sono locali al blocco di codice in cui sono dichiarate (e dunque sono accessibili solo all'interno di esso). La differenza consiste nel fatto che le variabili `static` hanno durata estesa a tutto il tempo di esecuzione del programma. Esse, pertanto, esistono già prima che il blocco in cui sono dichiarate sia eseguito e continuano ad esistere anche dopo il termine dell'esecuzione del medesimo.

Ne segue che i valori in esse contenuti sono persistenti; quindi se il blocco di codice viene nuovamente eseguito esse si presentano con il valore posseduto al termine dell'esecuzione precedente.

In altre parole, il compilatore alloca in modo permanente alle variabili `static` la memoria loro necessaria.

Il tutto può essere chiarito con un paio di esempi:

```
#include <stdio.h>

void incrementa(void)
{
    int x = 0;

    ++x;
    printf("%d\n",x);
}

void main(void)
{
    incrementa();
    incrementa();
    incrementa();
}
```

Il programma chiama la funzione `incrementa()` 3 volte; ad ogni chiamata la variabile `x`, automatic, è dichiarata ed inizializzata a `0`. Essa è poi incrementata e visualizzata. L'output del programma è il seguente:

```
1
1
1
```

Infatti `x`, essendo una variabile automatic, "sparisce" al termine dell'esecuzione della funzione in cui è dichiarata. Ad ogni chiamata essa è nuovamente allocata, inizializzata a `0`, incrementata, visualizzata e... buttata alle ortiche. Indipendentemente dal numero di chiamate, `incrementa()` visualizza sempre il valore `1`.

Riprendiamo ora la funzione `incrementa()`, modificando però la dichiarazione di `x`:

```
void incrementa(void)
{
    static int x = 0;

    ++x;
    printf("%d\n",x);
}
```

Questa volta `x` è dichiarata `static`. Vediamo l'output del programma:

```
1
2
3
```

La `x` è inizializzata a `0` solo una volta, al momento della *compilazione*. Durante la prima chiamata ad `incrementa()`, essa assume pertanto valore `1`. Poiché `x` è `static`, il suo valore è persistente e non viene perso in uscita dalla funzione. Ne deriva che alla seconda chiamata di `incrementa()` essa assume valore `2` e, infine, `3` alla terza chiamata.

Quando si specifica un valore iniziale per una variabile automatic, detto valore è assegnato alla variabile ogni volta che viene eseguito il blocco in cui la variabile stessa è dichiarata. Una inizializzazione come:

```
{
    int x = 1;
    ....
```

non è che una forma abbreviata della seguente:

```
{
    int x;

    x = 1;
    ....
```

Quanto detto non è vero per le variabili `static`. Il valore iniziale di `1` nella seguente riga di codice:

```
    static int x = 1;
```

viene assegnato alla variabile `x` una sola volta, in fase di compilazione: il compilatore riserva spazio per la variabile e vi memorizza il valore iniziale. Quando il programma è eseguito, il valore iniziale della variabile è già presente in essa.

Se il programmatore non inizializza esplicitamente una variabile `static`, il compilatore le assegna automaticamente il valore `NULL`, cioè lo zero.

Va poi sottolineato che l'accessibilità di una variabile `static` è comunque limitata (come per le varibili automatic) al blocco di codice in cui è dichiarata. Nel programma riportato per esempio, la variabile `x` non è accessibile né in `main()` né in qualunque altra funzione eventualmente definita, ma solamente all'interno di `incrementa()`.

Infine, è opportuno ricordare che un array dichiarato in una funzione deve necessariamente essere dichiarato `static` se inizializzato contestualmente alla dichiarazione.

Le variabili external

Sono variabili *external* tutte quelle dichiarate al di fuori delle funzioni. Esse hanno durata estesa a tutto il tempo di esecuzione del programma, ed in ciò appaiono analoghe alle variabili `static`, ma differiscono da queste ultime in quanto la loro accessibilità è globale a tutto il codice del programma. In altre parole, è possibile leggere o modificare il contenuto di una variabile external in qualsiasi funzione. Vediamo, come sempre, un esempio:

```
#include <stdio.h>

int x = 123;

void incrementa(void)
{
    ++x;
    printf("%d\n",x);
}

void main(void)
{
    printf("%d\n",x);
    incrementa();
    printf("%d\n",x);
}
```

L'output del programma è il seguente:

```
123
124
124
```

Infatti la variabile `x`, essendo definita al di fuori di qualunque funzione, è accessibile sia in `main()` che in `incrementa()` e il suo valore è conservato per tutta la durata dell'esecuzione.

Se una variabile external (o globale) ha nome identico a quello di una variabile automatic (o locale), quest'ultima "nasconde" la prima. Il codice che segue:

```
#include <stdio.h>

int x = 123;

void main(void)
{
    printf("%d\n",x);
    {
        int x = 321;

        printf("%d\n",x);
```

```
    }
    printf("%d\n",x);
}
```

produce il seguente output:

```
123
321
123
```

Infatti la `x` locale dichiarata nel blocco di codice interno a `main()` nasconde la `x` globale, dichiarata fuori dalla funzione; tuttavia la variabile locale cessa di esistere alla fine del blocco, pertanto quella globale è nuovamente accessibile.

Anche le variabili external, come quelle `static`, sono inizializzate dal compilatore al momento della compilazione, ed è loro attribuito valore `0` se il programmatore non indica un valore iniziale contestualmente alla dichiarazione.

Come abbiamo visto, le variabili external devono essere dichiarate al di fuori delle funzioni, senza necessità di specificare alcuna particolare parola chiave. Tuttavia, esse possono (ma non è obbligatorio) essere dichiarate anche all'interno delle funzioni che le referenziano, questa volta necessariamente precedute dalla parola chiave `extern`:

```
#include <stdio.h>

int x = 123;

void main(void)
{
    extern int x;            // riga facoltativa; se c'e' non puo' reinizializzare x

    printf("%d\n",x);
}
```

In effetti il compilatore non richiede che le variabili external vengano dichiarate all'interno delle funzioni, ma in questo caso è necessario che tali variabili siano state dichiarate al di fuori della funzione e in linee di codice precedenti quelle della funzione stessa, come negli esempi precedenti. Se tali condizioni non sono rispettate il compilatore segnala un errore di simbolo non definito:

```
#include <stdio.h>

int x = 123;

void main(void)
{
    printf("%d\n",x,y);                  // errore! y non e' stata ancora dichiarata
}

int y = 321;
```

Il codice dell'esempio è compilato correttamente se si dichiara `extern` la `y` in `main()`:

```
#include <stdio.h>

int x = 123;

void main(void)
{
    extern int x;                                                // facoltativa
    extern int y;                                               // obbligatoria!
```

```
    printf("%d\n",x,y);
}

int y = 321;
```

Il problema può essere evitato dichiarando tutte le variabili globali in testa al sorgente, ma se una variabile external e una funzione che la referenzia sono definite in due file sorgenti diversi[40], è necessario comunque dichiarare la variabile nella funzione.

E' opportuno limitare al massimo l'uso delle funzioni external: il loro utilizzo indiscriminato, infatti, può generare risultati catastrofici. In un programma qualsiasi è infatti piuttosto facile perdere traccia del significato delle variabili, soprattutto quando esse siano numerose. Inoltre le variabili globali sono generate al momento della compilazione ed esistono durante tutta l'esecuzione, incrementando così lo spazio occupato dal file eseguibile e la quantità memoria utilizzata dallo stesso. Infine, con esse non è possibile utilizzare nomi localmente significativi (cioè significativi per la funzione nella quale vengono di volta in volta utilizzate) e si perde la possibilità di mantenere ogni funzione una entità a se stante, indipendente da tutte le altre.

Va infine osservato che una variabile external può essere anche `static`:

```
#include <stdio.h>

static int x = 123;

void main(void)
{
    printf("%d\n",x);
}
```

Dichiarando `static` una variabile globale se ne limita la visibilità al solo file in cui essa è dichiarata: nel caso di un codice suddiviso in più sorgenti, le funzioni definite in altri file non saranno in grado di accedere alla `x` neppure qualora essa venga dichiarata con `extern` al loro interno. Naturalmente è ancora possibile dichiarare `extern` la variabile nelle funzioni definite nel medesimo file:

```
#include <stdio.h>

static int x = 123;

void main(void)
{
    extern int x;

    printf("%d\n",x);
}
```

Come facilmente desumibile dall'esempio, la parola chiave `static` non deve essere ripetuta.

LE COSTANTI

Le costanti, in senso lato, sono dati che il programma non può modificare. Una costante è, ad esempio, la sequenza di caratteri `"Ciao Ciao!\n"` vista in precedenza: per la precisione, si tratta di una costante stringa. Essa non può essere modificata perché non le è associato alcun nome simbolico a cui

[40] Suddividere il codice di un programma molto "corposo" in più file sorgenti può facilitarne la manutenzione, ma soprattutto consente, in caso di modifche, di ricompilare solo le parti effettivamente modificate, a tutto vantaggio dell'efficienza del processo di programmazione.

fare riferimento in un'operazione di assegnazione. Una costante è un valore esplicito, che può essere assegnato ad una variabile, ma al quale non può essere mai assegnato un valore diverso da quello iniziale.

Ad esempio, una costante di tipo character (carattere) è un singolo carattere racchiuso tra apici.

```
char c1, c2 = 'A';
c1 = 'b';
c2 = c1;
'c' = c2;              //ERRORE! impossibile assegnare un valore a una costante
```

Una costante intera con segno è un numero intero:

```
int unIntero = 245, interoNegativo = -44;
```

Una costante intera senza segno è un numero intero seguito dalla lettera `U`, maisucola o minuscola, come ci insegna il nostro `CIAO2.C`:

```
unsigned int anni = 31U;
```

Per esprimere una costante di tipo `long` occorre posporle la lettera `L`, maiuscola o minuscola[41].

```
long abitanti = 987553L;
```

Omettere la `L` non è un reato grave... il compilatore segnala con un warning che la costante è `long` e procede tranquillamente. In effetti, questo è l'atteggiamento tipico del compilatore C: quando qualcosa non è chiaro tenta di risolvere da sé l'ambiguità, e si limita a segnalare al programmatore di avere incontrato qualcosa di... poco convincente. Il compilatore C "presume" che il programmatore sappia quel che sta facendo e non si immischia nelle ambiguità logiche più di quanto sia strettamente indispensabile.

Una `U` (o `u`) individua una costante `unsigned`; le costanti `unsigned long` sono identificate, ovviamente, da entrambe le lettere `U` e `L`, maiuscole o minuscole, in qualsivoglia ordine. Le costanti appartenenti ai tipi integral possono essere espresse sia in notazione decimale (come in tutti gli esempi visti finora), sia in notazione esadecimale (anteponendo i caratteri `0x` o `0X` al valore) sia in notazione ottale (anteponendo uno `0` al valore).

```
char beep = 07;                                                  // ottale; 7
unsigned long uLong = 12UL;                        // decimale; 12 unsigned long
unsigned maxUInt = 0xFFFFU;                       // esadecimale; 65535 unsigned
```

Una costante di tipo floating point in doppia precisione (`double`) può essere espressa sia in notazione decimale che in notazione esponenziale: in questo caso si scrive la mantissa seguita dalla lettera `E` maiuscola o minuscola, a sua volta seguita dall'esponente. Per indicare che la costante è in singola precisione (`float`), occorre posporle la lettera `F`, maiuscola o minuscola. Per specificare una costante `long double` occorre la lettera `L`.

```
float varF = 1.0F;
double varD = 1.0;
double varD_2 = 1.;            // lo 0 dopo il punto decimale puo' essere omesso
long double varLD = 1.0L;         // non e' un long int! C'e' il punto decimale!
double varD_3 = 2.34E-2;                                              // 0.0234
```

[41] E' preferibile utilizzare la `L` maiuscola, poiché, nella lettura dei listati, quella minuscola può facilmente essere scambiata con la cifra `1`.

Dagli esempi si deduce immediatamente che la virgola è espressa, secondo la convenzione anglosassone, con il punto (`"."`).

Il C non riconosce le stringhe come tipo di dato, ma ammette l'utilizzo di costanti stringa (seppure con qualche limite, di cui si dirà): esse sono sequenze di caratteri racchiuse tra virgolette, come si è visto in più occasioni. Quanti byte occupa una stringa? Il numero dei caratteri che la compongono... più uno (pag. 25). In effetti le stringhe sono sempre chiuse da un byte avente valore zero binario[42], detto terminatore di stringa. Il `NULL` finale è generato automaticamente dal compilatore, non deve essere specificato esplicitamente.

Attenzione: le sequenze di caratteri particolari, come "`\n`", sono considerate un solo carattere (ed occupano un solo byte). I caratteri che non rientrano tra quelli presenti sulla tastiera possono essere rappresentati con una *backslash* (barra inversa) seguita da una "`x`" e dal codice ASCII esadecimale a due cifre del carattere stesso. Ad esempio, la stringa `"\x07\x0D\x0A"` contiene un "beep" (il carattere ASCII 7) e un ritorno a capo (i caratteri ASCII 13 e 10, questi ultimi equivalenti alla sequenza "`\n`"[43].

I codici ASCII possono essere utilizzati anche per esprimere un singolo carattere:

```
char beep = '\x07';
```

E' del tutto equivalente assegnare ad una variabile char un valore decimale, ottale o esadecimale o, ancora, il valore espresso con `\x` tra apici. Attenzione, però: la rappresentazione ASCII di un carattere è cosa ben diversa dal suo valore ASCII; `7`, `07`, `0x07` e `'\x07'` sono tra loro equivalenti, ma diversi da `'7'`.

La differenza tra un singolo carattere rispetto ad una stringa di un solo carattere sta negli apici, che sostistuiscono le virgolette. Inoltre, `'\x07'` occupa un solo byte, mentre `"\x07"` ne occupa due, uno per il carattere ASCII 7 e uno per il `NULL` che chiude ogni stringa.

Non esistono costanti di tipo `void`.

Le costanti manifeste

Supponiamo di scrivere un programma per la gestione dei conti correnti bancari. E' noto (e se non lo era ve lo dico io) che nei calcoli finanziari la durata dell'anno è assunta pari a 360 giorni. Nel sorgente del programma si potrebbero perciò incontrare calcoli come il seguente:

```
interesse = importo * giorniDeposito * tassoUnitario / 360;
```

il quale impiega, quale divisore, la costante intera `360`.

E' verosimile che nel programma la costante `360` compaia più volte, in diversi contesti (principalmente in formule di calcolo finanziario). Se in futuro fosse necessario modificare il valore della costante (una nuova normativa legale potrebbe imporre di assumere la durata dell'anno finanziario pari a 365 giorni) dovremmo ricercare tutte le occorrenze della costante `360` ed effettuare la sostituzione con `365`. In un sorgente di poche righe tutto ciò non rappresenterebbe certo un guaio, ma immaginando un codice di diverse migliaia di righe suddivise in un certo numero di file sorgenti, con qualche centinaio di occorrenze della costante, è facile prevedere quanto gravoso potrebbe rivelarsi il compito, e quanto grande sarebbe la possibilità di non riuscire a portarlo a termine senza errori.

---

[42] Nel senso che tutti i suoi 8 bit sono impostati a zero; non va confuso col carattere `'0'`.

[43] Le espressioni di controllo generate da un carattere preceduto dalla backslash sono anche dette sequenze ANSI.

Il preprocessore C consente di aggirare l'ostacolo mediante la direttiva `#define`, che associa tra loro due sequenze di caratteri in modo tale che, prima della compilazione ed in modo del tutto automatico, ad ogni occorrenza della prima (detta *manifest constant*) è sostituita la seconda.

Il nome della costante manifesta ha inizio col primo carattere *non-blank*[44] che segue la direttiva `#define` e termina con il carattere che precede il primo successivo non-spazio; tutto quanto segue quest'ultimo è considerato stringa di sostituzione.

Complicato? Solo in apparenza...

```
#define    GG_ANNO_FIN   360              //durata in giorni dell'anno finanziario

....

    interesse = importo * giorniDeposito * tassoUnitario / GG_ANNO_FIN;
```

L'esempio appena visto risolve il nostro problema: modificando la direttiva `#define` in modo che al posto del `360` compaia il `365` e ricompilando il programma, la sostituzione viene effettuata automaticamente in tutte le righe in cui compare `GG_ANNO_FIN`.

Va sottolineato che la direttiva `#define` non crea una variabile, né è associata ad un tipo di dato particolare: essa informa semplicemente il preprocessore che la costante manifesta, ogniqualvolta compaia nel sorgente in fase di compilazione, deve essere rimpiazzata con la stringa di sostituzione. Gli esempi che seguono forniscono ulteriori chiarimenti: in essi sono definite costanti manifeste che rappresentano, rispettivamente, una costante stringa, una costante in virgola mobile, un carattere esadecimale e una costante long integer, ancora esadecimale.

```
#define    NOME_PROG      "Conto 1.0"                          //nome del programma
#define    PI_GRECO       3.14                               //pi greco arrotondato
#define    RETURN         0x0D                                     //ritorno a capo
#define    VIDEO_ADDRESS  0xB8000000L                      //indirizzo del buffer video
```

Le costanti manifeste possono essere definite utilizzando altre costanti manifeste, purché definite in precedenza:

```
#define    N_PAG_VIDEO    8                      //numero di pagine video disponibili
#define    DIM_PAG_VIDEO  4000                      //4000 bytes in ogni pagina video
#define    VIDEO_MEMORY   (N_PAG_VIDEO * DIM_PAG_VIDEO)         //spazio memoria video
```

Una direttiva `#define` può essere suddivisa in più righe fisiche mediante l'uso della backslash:

```
#define    VIDEO_MEMORY    \
           (N_PAG_VIDEO * DIM_PAG_VIDEO)
```

L'uso delle maiuscole nelle costanti manifeste non è obbligatorio; esso tuttavia è assai diffuso in quanto consente di individuarle più facilmente nella lettura dei sorgenti.

Come tutte le direttive al preprocessore, anche la `#define` non si chiude mai con il punto e virgola (un eventuale punto e virgola verrebbe inesorabilmente considerato parte della stringa di sostituzione); inoltre il *crosshatch* ("`#`", cancelletto) deve trovarsi in prima colonna.

La direttiva `#define`, implementando una vera e propria tecnica di sostituzione degli argomenti, consente di definire, quali costanti manifeste, vere e proprie formule, dette *macro*, indipendenti dai tipi di dato coinvolti:

```
#define    min(a,b)    ((a < b) ? a : b)  // macro per il calcolo del minimo tra due
```

[44] Per non-blank o non-spazio si intende qualsiasi carattere diverso da spazi bianchi e tabulazioni.

Come si vede, nella macro `min(a,b)` non è data alcuna indicazione circa il tipo di `a` e `b`: essa utilizza l'operatore `? :`, che può essere applicato ad ogni tipo di dato[45]. Il programmatore è perciò libero di utilizzarla in qualunque contesto.

Le macro costituiscono dunque uno strumento molto potente, ma anche pericoloso: in primo luogo, la mancanza di controlli (da parte del compilatore) sui tipi di dato può impedire che siano segnalate incongruenze logiche di un certo rilievo (sommare le pere alle mele, come si dice...). In secondo luogo, le macro prestano il fianco ai cosiddetti *side-effect*, o effetti collaterali. Il C implementa un particolare operatore, detto di *autoincremento*[46], che accresce di una unità il valore della variabile a cui è anteposto: se applicato a uno dei parametri coinvolti nella macro, esso viene applicato più volte al parametro, producendo risultati indesiderati:

```
int var1, var2;
int minimo;

....

minimo = min(++var1, var2);
```

La macrosotituzione effettuata dal preprocessore trasforma l'ultima riga dell'esempio nella seguente:

```
minimo = ((++var1 < var2) ? ++var1 : var2);
```

E' facile vedere che esso si limita a sostituire alla macro `min` la definizione data con la `#define`, sostituendo altresì i parametri `a` e `b` con i simboli utilizzati al loro posto nella riga di codice, cioè `++var1` e `var2`. In tal modo `var1` è incrementata due volte se dopo il primo incremento essa risulta ancora minore di `var2`, una sola volta nel caso opposto. Se `min()` fosse una funzione il problema non potrebbe verificarsi (una chiamata a funzione non è una semplice sostituzione di stringhe, ma un'operazione tradotta in linguaggio macchina dal compilatore seguendo precise regole); tuttavia una funzione non accetterebbe indifferentemente argomenti di vario tipo, e occorrerebbe definire funzioni diverse per effettuare confronti, di volta in volta, tra integer, tra floating point, e così via. Un altro esempio di effetto collaterale è discusso a pag. 463.

Diamo un'occhiata all'esempio che segue:

```
#define  PROG_NAME    "PROVA"
....
    printf(PROG_NAME);
....
#undef  PROG_NAME
....
    printf(PROG_NAME);                        // Errore! PROG_NAME non esiste piu'...
```

Quando una definizione generata con una `#define` non serve più, la si può annullare con la direttiva `#undef`. Ogni riferimento alla definizione annullata, successivamente inserito nel programma, dà luogo ad una segnalazione di errore da parte del compilatore.

Da notare che `PROG_NAME` è passata a `printf()` senza porla tra virgolette, in quanto esse sono già parte della stringa di sostituzione, come si può vedere nell'esempio. Se si fossero utilizzate le virgolette, `printf()` avrebbe scritto `PROG_NAME` e non `PROVA`: il preprocessore, infatti, ignora tutto

---

[45] Degli operatori C parleremo diffusamente a pag. 61 e seguenti. Il significato di `? :` può essere, per il momento, dedotto dall'esempio. Per ora merita attenzione il fatto che molti compilatori implementano `max()` e `min()` proprio come macro, definite in uno dei file `.H` di libreria.

[46] Anche l'operatore, `++`, verrà descritto ampiamente (pag. 64).

quanto è racchiuso tra virgolette o apici. In altre parole, esso non ficca il naso nelle costanti stringa e in quelle di tipo carattere.

Vale la pena di citare anche la direttiva `#ifdef...#else...#endif`, che consente di includere o escludere dalla compilazione un parte di codice, a seconda che sia, o meno, definita una determinata costante manifesta:

```
#define DEBUG
....
#ifdef DEBUG
....                                    // questa parte del sorgente e' compilata
#else
....                        // questa no (lo sarebbe se NON fosse definita DEBUG)
#endif
```

La direttiva `#ifndef` e' analoga alla `#ifdef`, ma lavora con logica inversa:

```
#define DEBUG
....
#ifndef DEBUG
....                                // questa parte del sorgente NON e' compilata
#else
....                    // questa si (NON lo sarebbe se NON fosse definita DEBUG)
#endif
```

Le direttive `#ifdef` e `#ifndef` risultano particolarmente utili per scrivere codice portabile (vedere pag. 461): le parti di sorgente differenti in dipendenza dal compilatore, dal sistema o dalla macchina possono essere escluse o incluse nella compilazione con la semplice definizione di una costante manifesta in testa al sorgente.

Le costanti simboliche

E' di recente diffusione, tra i programmatori C, la tendenza a limitare quanto più possibile l'uso delle costanti manifeste, in parte proprio per evitare la possibilità di effetti collaterali, ma anche per considerazioni relative alla logica della programmazione: le costanti manifeste creano problemi in fase di *debugging*[47], poiché non è possibile sapere dove esse si trovino nella memoria dell'elaboratore (come tutte le costanti, non hanno indirizzo conoscibile); inoltre non sempre è possibile distinguere a prima vista una costante manifesta da una variabile, se non rintracciando la `#define` (l'uso delle maiuscole e minuscole è libero tanto nelle costanti manifeste quanto nei nomi di variabili, pertanto nulla garantisce che un simbolo espresso interamente con caratteri maiuscoli sia effettivamente una costante manifesta).

Il C consente di definire delle costanti simboliche dichiarandole come vere e proprie variabili, ma anteponendo al dichiaratore di tipo la parola chiave `const`. Ecco un paio di esempi:

```
    const int  ggAnnoFin = 360;
    const char return = 0x0D;
```

E' facile vedere che si tratta di dichiarazioni del tutto analoghe a quelle di variabili; tuttavia la presenza di `const` forza il compilatore a considerare costante il valore contenuto nell'area di memoria associata al nome simbolico. Il compilatore segnala come illegale qualsiasi tentativo di modificare il

[47] La fase, cioè, di ricerca e correzione degli errori di programmazione. Questa è effettuata con l'aiuto di sofisticati programmi, detti debuggers, che sono spesso in grado di visualizzare il contenuto delle variabili associandovi il nome simbolico; cosa peraltro impossibile con le costanti manifeste, che ne sono prive.

valore di una costante, pertanto ogni costante dichiarata mediante `const` deve essere inizializzata contestualmente alla dichiarazione stessa.

```
const int unIntCostante = 14;

....

unIntCostante = 26;   //errore: non si puo' modificare il valore di una costante
```

Il principale vantaggio offerto da `const` è che risulta possibile accedere (in sola lettura) al valore delle costanti così dichiarate mediante l'indirizzo delle medesime (come accade per tutte le aree di memoria associate a nomi simbolici): ancora una volta rimandiamo gli approfondimenti alla trattazione dei puntatori (pag. 16).

Infine, le costanti simboliche possono essere gestite dai debugger proprio come se fossero variabili.

ENTITÀ COMPLESSE

I tipi di dato discussi in precedenza sono intrinseci al compilatore: quelli, cioè, che esso è in grado di gestire senza ulteriori costruzioni logiche da parte del programmatore; possiamo indicarli come tipi elementari.

Spesso, però, essi non sono sufficienti a rappresentare in modo esauriente le realtà oggetto di elaborazione: In un semplice programma che gestisca in modo grafico il monitor del computer può essere comodo rappresentare un generico punto luminoso (*pixel*) del monitor stesso come un'entità unica, individuata mediante parametri che consentano, attraverso il loro valore, di distinguerla dalle altre dello stesso tipo: si tratta di un'entità complessa.

Infatti ogni pixel può essere descritto, semplificando un po', mediante tre parametri caratteristici: le coordinate (che sono due, ascissa e ordinata, trattandosi di uno spazio bidimensionale) e il colore.

Il C mette a disposizione del programmatore alcuni strumenti atti a rappresentare entità complesse in modo più prossimo alla percezione che l'uomo ne ha, di quanto consentano i tipi di dato finora visti. Non si tratta ancora della possibilità di definire veri e propri tipi di dato "nuovi" e di gestirli come se fossero intrinseci al linguaggio[48], ma è comunque un passo avanti...

Le strutture

Tra gli strumenti cui si è fatto cenno appare fondamentale la struttura (*structure*), mediante la quale si definisce un modello (*template*) che individua un'aggregazione di tipi di dato fondamentali.

Ecco come potremmo descrivere un pixel con l'aiuto di una struttura:

```
struct pixel {
    int x;
    int y;
    int colour;
};
```

---

[48] In tal senso strumenti molto potenti sono offerti dal C++, con il quale si possono "inventare" nuovi tipi di dato, definendone anche le modalità di manipolazione, e gestirli, se ben progettati, senza alcuna differenza rispetto a quelli elementari intrinseci.

Quello dell'esempio è una dichiarazione di template di struttura: si apre con la parola chiave `struct` seguita dal nome (*tag*) che intendiamo dare al nostro modello; questo è a sua volta seguito da una graffa aperta. Le righe che seguono, vere e proprie dichiarazioni di variabili, individuano il contenuto della struttura e si concludono con una graffa chiusa seguita dal punto e virgola.

Ecco un altro esempio di dichiarazione di template, dal quale risulta chiaro che una struttura può comprendere differenti tipi di dato:

```
struct ContoCorrente {
    char   intestatario[50];
    char   data_accensione[9];
    int    cod_filiale;
    double saldo;
    double tasso_interesse;
    double max_fido;
    double tasso_scoperto;
};
```

E' meglio focalizzare sin d'ora che la dichiarazione di un template di struttura non comporta che il compilatore riservi dello spazio di memoria per allocare i campi[49] della struttura stessa. La dichiarazione di template definisce semplicemente la "forma" della struttura, cioè il suo modello.

Di solito le dichiarazioni di template di struttura compaiono all'inizio del sorgente, anche perché i templates devono essere stati dichiarati per poter essere utilizzati: solo dopo avere definito l'identifictore (tag) e il modello (template) della struttura, come negli esempi di poco fa, è possibile dichiarare ed utilizzare oggetti di quel tipo, vere e proprie variabili `struct`.

```
#include <stdio.h>

struct concorso {
    int serie;
    char organizzatore;
    int partecipanti;
};

void main(void)
{
    struct concorso c0, c1;

    c0.serie = 2;
    c0.organizzatore = 'F';
    c0.partecipanti = 482;
    c1.serie = 0;
    c1.organizzatore = 'G';
    c1.partecipanti = 33;
    printf("Serie della concorso 0: %d\n",c0.serie);
    printf("Organizzatore della concorso 1: %c\n",c1.organizzatore);
}
```

Nel programma dell'esempio viene dichiarato un template di struttura, avente tag `concorso`. Il template è poi utilizzato in `main()` per dichiarare due strutture di tipo `concorso`: solo a questo punto sono creati gli oggetti `concorso` e viene loro riservata memoria. Gli elementi, o campi, delle due strutture sono inizializzati con dati di tipo opportuno; infine alcuni di essi sono visualizzati con la solita `printf()`.

Cerchiamo di evidenziare alcuni concetti fondamentali, a scanso di equivoci. La dichiarazione di template non presenta nulla di nuovo: parola chiave `struct`, tag, graffa aperta, campi, graffa chiusa, punto e virgola. Una novità è invece rappresentata dalla dichiarazione delle strutture `c0` e `c1`: come si

[49] Le variabili comprese in una struttura si dicono campi.

vede essa è fortemente analoga a quelle di comuni variabili, con la differenza che le variabili dichiarate non appartengono al tipo `int`, `float`, o a uno degli altri tipi di dati sin qui trattati. Esse appartengono ad un tipo di dato nuovo, definito da noi: il tipo `struct concorso`.

Finora si è indicato con "dichiarazione di template" l'operazione che serve a definire l'aspetto della struttura, e con "dichiarazione di struttura" la creazione degli oggetti, cioè la dichiarazione delle variabili struttura. E' forse una terminologia prolissa, ma era indispensabile per chiarezza. Ora che siamo tutti diventati esperti di strutture potremo essere un poco più concisi e indicare con il termine "struttura", come comunemente avviene, tanto i template che le variabili di tipo `struct`[50].

In effetti, la dichiarazione:

```
struct concorso {
    int serie;
    char organizzatore;
    int partecipanti;
};
```

crea semplicemente un modello che può essere usato come riferimento per ottenere variabili dotate di quelle particolari caratteristiche. Ciascuna variabile conforme a quel modello contiene, nell'ordine prefissato, un `int`, un `char` e un secondo `int`. A ciascuna di queste variabili, come per quelle di qualsiasi altro tipo, il compilatore alloca un'area di memoria di dimensioni sufficienti, alla quale associa il nome simbolico che compare nella dichiarazione

```
    struct concorso c0;
```

cioè `c0`. In quest'ultima dichiarazione, l'identificatore `concorso` indica il modello particolare al quale si deve conformare la variabile dichiarata. Esso è, in pratica, un'abbreviazione di

```
{
    int serie;
    char organizzatore;
    int partecipanti;
};
```

e come tale può venire usato nel programma. In altre parole, è possibile riferirsi all'intera dichiarazione di struttura semplicemente usandone il tag.

Una variabile di tipo `struct` può essere dichiarata contestualmente al template:

```
struct concorso {
    int serie;
    char organizzatore;
    int partecipanti;
} c0, c1;
```

Il template può essere normalmente utilizzato per dichiarare altre strutture nel programma[51].

---

[50] Nella pratica comune ci si riferisce di solito alla "struttura `concorso`" allo stesso modo che alla "struttura `c0`", anche se nel primo caso si intende "il modello della struttura il cui identificatore è `concorso`" e nel secondo "la variabile di nome `c0`, il cui tipo è la `struct` avente modello `concorso`". I distratti sono avvertiti.

[51] Per completezza va osservato che dichiarando la variabile struttura e contemporaneamente definendone il template la creazione di un tag può essere omessa:

```
struct {
    int serie;
    char organizzatore;
```

Tornando a quel che avviene nella `main()` dell'esempio, ai campi delle strutture dichiarate sono stati assegnati valori con una notazione del tipo

```
nome_della_variabile_struttura.nome_del_campo = valore;
```

ed in effetti l'operatore punto (".") è lo strumento offerto dal C per accedere ai singoli campi delle variabili `struct`, tanto per assegnarvi un valore, quanto per leggerlo (e lo si vede dalle `printf()` che seguono).

Abbiamo parlato delle strutture viste negli esempi precedenti come di variabili di tipo `struct concorso`. In effetti definire un template di struttura significa arricchire il linguaggio di un nuovo tipo di dato, non intrinseco, ma al quale è possibile applicare la maggior parte dei concetti e degli strumenti disponibili con riferimento ai tipi di dato intrinseci.

Le strutture possono quindi essere gestite mediante array e puntatori, proprio come comuni variabili C. La dichiarazione di un array di strutture si prsenta come segue:

```
    struct concorso c[3];
```

Si nota immediatamente la forte somiglianza con la dichiarazione di un array di tipo intrinseco: il valore tra parentesi quadre specifica il numero di elementi, cioè, in questo caso, di strutture che formano l'array. Ogni elemento è, appunto, una struttura conforme al template `concorso`; l'array ha nome `c`. Per accedere ai singoli elementi dell'array è necessario, come prevedibile, specificare il nome dell'array seguito dall'indice, tra quadre, dell'elemento da referenziare. La differenza rispetto ad un array "comune", ad esempio di tipo `int`, sta nel fatto che accedere ad una struttura non significa ancora accedere ai dati che essa contiene: per farlo occorre usare l'operatore punto, come mostrato poco sopra. Un esempio chiarirà le idee:

```
#include <stdio.h>

struct concorso {
    int serie;
    char organizzatore;
    int partecipanti;
};

void main(void)
{
    register i;
    struct concorso c[3];

    c[0].serie = 2;
    c[0].organizzatore = 'F';
    c[0].partecipanti = 482;
    c[1].serie = 0;
    c[1].organizzatore = 'G';
    c[1].partecipanti = 33;
    c[2].serie = 3;
    c[2].organizzatore = 'E';
    c[2].partecipanti = 107;
    for(i = 0; i < 3; i++)
```

```
    int parteciapnti;
} c0, c1;
```

In questo caso, non esistendo un tag mediante il quale fare riferimento al template, è necessario riscrivere il template ogni volta che si dichiara altrove una variabile avente quelle stesse caratteristiche. Da evitare, assolutamente.

```
        printf("%d    %c    %d\n",c[i].serie,c[i].organizzatore,c[i].partecipanti);
}
```

Con riferimento ad un array di strutture, la sintassi usata per referenziare i campi di ciascuna struttura elemento dell'array è simile a quella utilizzata per array di tipi intrinseci. Ci si riferisce, ad esempio, al campo `serie` dell'elemento di posto `0` dell'array con la notazione `c[0].serie`; è banale osservare che `c[0]` accede all'elemento dell'array, mentre `.serie` accede al campo voluto di quell'elemento.

Si può pensare all'esempio presentato sopra immaginando di avere tre fogli di carta, ciascuno contenente un elemento dell'array `c`. In ciascun foglio sono presenti tre righe di informazioni che rappresentano, rispettivamente, i 3 campi della struttura. Se i 3 fogli vengono mantenuti impilati in ordine numerico crescente, si ottiene una rappresentazione "concreta" dell'array, in quanto è possibile conoscere sia il contenuto dei tre campi di ogni elemento, sia la relazione tra i vari elementi dell'array stesso.

I più attenti hanno sicuramente[52] notato che, mentre le operazioni di assegnamento, lettura, etc. con tipi di dato intrinseci vengono effettuate direttamente sulla variabile dichiarata, nel caso delle strutture esse sono effettuate sui campi, e non sulla struttura come entità direttamente accessibile. In realtà le regole del C non vietano di accedere direttamente ad una struttura intesa come un'unica entità, ma si tratta di una pratica poco seguita[53]. E' infatti assai più comodo ed efficiente utilizzare i puntatori.

Anche nel caso dei puntatori le analogie tra strutture e tipi intrinseci sono forti. La dichiarazione di un puntatore a struttura, infatti, è:

```
struct concorso *cPtr;
```

dove `cPtr` è il puntatore, che può contenere l'indirizzo di una struttura di template `concorso`. L'espressione `*cPtr` restituisce una `struct concorso`, esattamente come in una dichiarazione quale

```
int *iPtr;
```

`*iPtr` restituisce un `int`. Attenzione, però: per accedere ai campi di una struttura referenziata mediante un puntatore non si deve usare l'operatore punto, bensì l'operatore "freccia", formato dai caratteri "meno" ("`-`") e "maggiore" ("`>`") in sequenza, con una sintassi del tipo:

```
nome_del_puntatore_alla_variabile_di_tipo_struttura->nome_del_campo = valore;
```

Vediamo un esempio.

```
struct concorso *cPtr;

....
cPtr->serie = 2;
....
printf("Serie: %d\n",cPtr->serie);
```

I puntatori a struttura godono di tutte le proprietà dei puntatori a tipi intrinseci, tra le quali particolarmente interessante appare l'aritmetica dei puntatori (vedere pag. 33). Incrementare un puntatore a struttura significa sommare implicitamente al suo valore tante unità quante ne occorrono per "scavalcare" tutta la struttura referenziata e puntare quindi alla successiva. In generale, sommare un intero

---

[52] Un po' di ottimismo non guasta...

[53] Forse anche perché fino a qualche anno fa erano pochi i compilatori in grado di implementare tale sintassi.

ad un puntatore a struttura equivale sommare quell'intero moltiplicato per la dimensione della struttura[54]. E' appena il caso di sottolineare che la dimensione di un puntatore a struttura e la dimensione della struttura puntata sono due concetti differenti, come già si è detto per le variabili di tipo intrinseco. Un puntatore a struttura occupa sempre 2 o 4 byte, a seconda che sia `near`, oppure `far` o `huge`, indipendentemente dalla dimensione della struttura a cui punta. Con la dichiarazione di un puntatore a struttura, dunque, il compilatore non alloca memoria per la struttura stessa.

Rivediamo il programma d'esempio di poco fa, modificandolo per utilizzare un puntatore a struttura:

```
#include <stdio.h>

struct concorso {
    int serie;
    char organizzatore;
    int partecipanti;
};

void main(void)
{
    struct concorso c[3], *cPtr;

    c[0].serie = 2;
    c[0].organizzatore = 'F';
    c[0].partecipanti = 482;
    c[1].serie = 0;
    c[1].organizzatore = 'G';
    c[1].partecipanti = 33;
    c[2].serie = 3;
    c[2].organizzatore = 'E';
    c[2].partecipanti = 107;
    for(cPtr = c; cPtr < c+3; ++cPtr)
        printf("%d   %c   %d\n",cPtr->serie,cPtr->organizzatore,cPtr->partecipanti);
}
```

Come si può notare, la modifica consiste essenzialmente nell'avere dichiarato un puntatore a `struct concorso`, `cPtr`, e nell'averlo utilizzato in luogo della notazione `c[i]` per accedere agli elementi dell'array. Le dichiarazioni dell'array e del puntatore sono state raggruppate in un'unica istruzione, ma sarebbe stato possibile separarle: il codice

```
    struct concorso c[3];
    struct concorso *cPtr;
```

avrebbe avuto esattamente lo stesso significato, sebbene in forma meno compatta e, forse, più leggibile.

Nel ciclo `for` dell'esempio, il puntatore `cPtr` è inizializzato a `c` e poiché il nome di un array è puntatore all'array stesso, `cPtr` punta al primo elemento di `c`, cioè `c[0]`. Durante la prima iterazione sono visualizzati i valori dei 3 campi di `c[0]`; all'iterazione successiva `cPtr` viene incrementato per puntare al successivo elemento di `c`, cioè `c[1]`, e quindi l'espressione

```
cPtr->
```

è ora equivalente a

---

[54] La dimensione di una struttura può essere ricavata mediante l'operatore `sizeof()` (vedere pag. 68), passandogli quale argomento il tag preceduto dalla parola chiave `struct`, oppure il nome di una variabile struttura: basandoci sugli esempi visti sin qui, `sizeof(struct concorso)` e `sizeof(c0)` restituiscono entrambe la dimensione della struttura `concorso` (che nel nostro caso è pari a 5 byte).

c[1].

All'iterazione successiva, l'espressione

cPtr->

diviene equivalente a

c[2].

dal momento che cPtr è stato incrementato ancora una volta.

A proposito di puntatori, è forse il caso di evidenziare che una struttura può contare tra i suoi campi puntatori a qualsiasi tipo di dato. Sono perciò ammessi anche puntatori a struttura, persino puntatori a struttura identificata dal medesimo tag. In altre parole, è perfettamente lecito scrivere:

```
struct TextLine {
    char *line;
    int  cCount;
    struct TextLine *prevTL;
    struct TextLine *nextTL;
};
```

Quella dell'esempio è una struttura (o meglio, un template di struttura) che potrebbe essere utilizzata per una rudimentale gestione delle righe di un testo, ad esempio in un programma di *word processing*. Essa contiene due puntatori a struttura dello stesso template: nell'ipotesi che ogni riga di testo sia gestita attraverso una struttura TextLine, prevTL è valorizzato con l'indirizzo della struct TextLine relativa alla riga precedente nel testo, mentre nextTL punta alla struct TextLine della riga successiva[55]. E' proprio mediante un utilizzo analogo a questo dei puntatori che vengono implementati oggetti quali le liste. Uno dei vantaggi immediatamente visibili che derivano dall'uso descritto dei due puntatori prevTL e nextTL consiste nella possibilità di implementare algoritmi di ordinamento delle righe di testo che agiscano solo sui puntatori: è sufficiente modificare il modo in cui le righe di testo sono legate l'una all'altra da un punto di vista logico, senza necessità alcuna di modificarne l'ordine fisico in memoria.

E' ovvio che, come al solito, un puntatore non riserva lo spazio per l'oggetto a cui punta. Nell'ipotesi di puntatori near, l'espressione sizeof(struct TextLine) restituisce 8. La memoria necessaria a contenere la riga di testo e le strutture TextLine stesse deve essere allocata esplicitamente.

Nel caso degli array, al contrario, la memoria è allocata staticamente dal compilatore (anche qui nulla di nuovo): riscriviamo il template in modo da gestire la riga di testo come un array di caratteri, avente dimensione massima prestabilita (in questo caso 80):

```
struct TextLine {
    char line[80];
    int  cCount;
    struct TextLine *prevTL;
    struct TextLine *nextTL;
};
```

questa volta l'espressione sizeof(struct TextLine) restituisce 86.

Va anche precisato che una struttura può contenere un'altra struttura (e non solo il puntatore ad essa), purché identificata da un diverso tag:

---

[55] Un prevTL e un nextTL contenenti NULL segnalano che le righe gestite dalle strutture di cui fanno parte sono, rispettivamente, la prima e l'ultima del testo. Se una struct TextLine presenta entrambi i puntatori NULL, allora essa gestisce l'unica riga di testo. Se è NULL anche il puntatore alla riga, line, allora il testo è vuoto.

```
struct TextParms {
    int textLen;
    int indent;
    int justifyType;
};

struct TextLine {
    char line[80];
    struct TextParms lineParms;
    struct TextLine *prevTL;
    struct TextLine *nextTL;
};
```

In casi come questo le dichiarazioni delle due strutture possono perfino essere nidificate:

```
struct TextLine {
    char line[80];
    struct TextParms {
        int textLen;
        int indent;
        int justifyType;
    } lineParms;
    struct TextLine *prevTL;
    struct TextLine *nextTL;
};
```

Da quanto appena detto appare evidente che una struttura non può mai contenere una struttura avente il proprio stesso tag identificativo: per il compilatore sarebbe impossibile risolvere completamente la definizione della struttura, in quanto essa risulterebbe definita in funzione di se stessa. In altre parole è illecita una dichiarazione come:

```
struct ST {
    int number;                                                         // OK
    float *fPtr;                                                        // OK
    struct ST inner;       // NO! il tag e' il medesimo e questo non e' un puntatore
};
```

Anche agli elementi di strutture nidificate si accede tramite il punto ("`.`") o la freccia ("`->`"): con riferimento ai templates appena riportati, è possibile, ad esempio, dichiarare un array di strutture `TextLine`:

```
    struct TextLine tl[100];             // dichiara un array di strutture TextLine
```

Alla riga di testo gestita dal primo elemento dell'array si accede, come già sappiamo, con l'espressione `tl[0].line`. Per visualizzare la riga successiva (gestita dall'elemento di `tl` il cui indirizzo è contenuto in `nextTL`) vale la seguente:

```
    printf("Prossima riga: %s\n",tl[0].nextTL->line);
```

Infatti `tl[0].nextTL` accede al campo `nextTL` di `tl[0]`: l'operatore utilizzato è il punto, proprio perchè `tl[0]` è una struttura e non un puntatore a struttura. Ma `nextTL` è, al contrario, un puntatore, perciò per referenziare l'elemento `line` della struttura che si trova all'indirizzo che esso contiene è necessario usare la "freccia". Supponiamo ora di voler conoscere l'indentazione (rientro rispetto al margine) della riga appena visualizzata: è ormai noto che ai campi della struttura "puntata" da `nextTL` si accede con l'operatore `->`; se il campo referenziato è, a sua volta, una struttura (`lineParms`), i campi di questa sono "raggiungibili" mediante il punto.

```
printf("Indentazione della prossima riga: %d\n",tl[0].nextTL->lineParms.indent);
```

Insomma, la regola generale (che richiede di utilizzare il punto se l'elemento fa parte di una struttura referenziata direttamente e la freccia se l'elemento è raggiungibile attraverso il puntatore alla struttura) rimane valida e si applica pedestremente ad ogni livello di nidificazione.

Vale infine la pena di chiarire che le strutture, pur costituendo un potente strumento per la rappresentazione informatica di entità complesse (quali record di archivi, etc.), sono ottimi "aiutanti" anche quando si desideri semplificare il codice ed incrementarne l'efficienza; se, ad esempio, occorre passare molti parametri ad una funzione, e questa è richiamata molte volte (si pensi al caso di un ciclo con molte iterazioni), può essere conveniente definire una struttura che raggruppi tutti quei parametri, così da poter passare alla funzione un parametro soltanto: il puntatore alla struttura stessa (vedere pag. 87).

Le unioni

Da quanto detto circa le strutture, appare evidente come esse costituiscano uno strumento per la rappresentazione di realtà complesse, in quanto sono in grado di raggrupparne i molteplici aspetti quantitativi[56]. In particolare, ogni singolo campo di una struttura permette di gestire uno degli aspetti che, insieme, descrivono l'oggetto reale.

Il concetto di unione deriva direttamente da quello di struttura, ma con una importante differenza: i campi di una `union` rappresentano diversi modi di vedere, o meglio, rappresentare, l'oggetto che la `union` stessa descrive. Consideriamo l'esempio seguente:

```
struct FarPtrWords {
    unsigned offset;
    unsigned segment;
};

union Far_c_Ptr {
    char far *ptr;
    struct FarPtrWords words;
};
```

La dichiarazione di un template di `union` è del tutto analoga a quella di un template di struttura: l'unica differenza è costituita dalla presenza della parola chiave `union` in luogo di `struct`.

La differenza è però enorme a livello concettuale: la `struct FarPtrWords` comprende due campi, entrambi di tipo `unsigned int`. Non ci vuole molto a capire che essa occupa 4 byte e descrive un puntatore di tipo "non `near`", scomposto nelle due componenti di indirizzo[57].

I due campi della `union Far_c_Ptr`, invece, sono rispettivamente un puntatore a 32 bit e una `struct FarPtrWords`. Contrariamente a quanto ci si potrebbe aspettare, la `union` non occupa 8 byte, bensì solo 4: puntatore e `struct FarPtrWords` sono due modi alternativi di interpretarli o, in altre parole, di accedere al loro contenuto. La `union Far_c_Ptr` è un comodo strumento per gestire un puntatore come tale, o nelle sue parti offset e segmento, a seconda della necessità. L'area di memoria in cui il dato si trova è sempre la stessa, ma il campo `ptr` la referenzia come un tutt'uno, mentre la struttura `FarPtrWord` consente di accedere ai primi due byte o agli ultimi due, separatamente.

---

[56] Per quelli qualitativi ci si deve accontentare di una loro "traduzione" in termini quantitativi.

[57] Perché prima l'offset e poi il segmento? E' sempre la solita storia: i processori Intel memorizzano i byte più significativi di una variabile nelle locazioni di memoria aventi indirizzo maggiore (tecnica *backwords*). Perciò di un dato a 32 bit è memorizzato prima il quarto byte, poi il terzo (ed è la seconda word), poi il secondo, infine il primo (ed è la prima word).

Si può pensare ad una union come ad un insieme di "maschere" attraverso le quali interpretare il contenuto di un'area di memoria.

Vediamo la sintassi, senza preoccuparci dell'espressione (char far *): si tratta di un *cast* (pag. 65) e non riguarda in modo diretto l'argomento "union":

```
    union FarPtr fp;

    fp.ptr = (char far *)0xB8000000L;
    printf("ptr: %Fp\n",fp.ptr);
    printf("ptr: %X:%X\n",fp.words.segment,fp.words.offset);
```

L'accesso ai membri di una union segue le medesime regole dell'accesso ai membri di una struct, cioè mediante l'operatore punto (o l'operatore "freccia" se si lavora con un puntatore). E' interessante notare che inizializzando il campo ptr viene inizializzato anche il campo word, in quanto condividono la stessa memoria fisica. Il medesimo campo ptr è poi utilizzato per ricavare il valore del puntatore, mentre il campo words consente di accedere alle componenti segment ed offset. Entrambe le printf() visualizzano

```
B800:0000
```

Se nel codice dell'esempio si sostituisce la riga di inizializzazione del campo ptr con le righe seguenti:

```
    fp.words.offset = 0;
    fp.words.segment = 0xB800;
```

le due printf() visualizzano ancora il medesimo output.

La sintassi che consente di accedere ai due campi della struct FarPtrWords, a prima vista, può apparire strana, ma in realtà essa è perfettamente coerente con le regole esposte con riferimento alle strutture: dal momento che ai campi di una union si accede mediante l'operatore punto, sono giustificate le scritture fp.ptr e fp.words ma l'operatore punto si utilizza anche per accedere ai membri di una struttura, perciò sono lecite le scritture word.offset e word.segment; ciò spiega fp.word.offset e fp.word.segment.

Nell'esempio di union analizzato, i membri sono due ed hanno uguale dimensione (entrambi 4 byte). Va precisato che i membri di una union possono essere più di due; inoltre essi possono essere di dimensioni differenti l'uno dall'altro, nel qual caso il compilatore, allocando la union, le riserva una quantità di memoria sufficiente a contenere il più "ingombrante" dei suoi membri, e li "sovrappone" a partire dall'inizio dell'area di memoria occupata dalla union stessa. Esempio:

```
union Far_c_Ptr {
    char far *ptr;
    struct FarPtrWords words;
    unsigned pOffset;
}
```

Il terzo elemento della union è un unsigned int, e come tale occupa 2 byte. Questi coincidono con i primi due byte di ptr (e della struct), e rappresentano pertanto la word offset del puntatore rappresentato dalla union.

I puntatori a union si comportano esattamente come i puntatori a struct.

Gli enumeratori

Gli enumeratori sono un ulteriore strumento che il C rende disponibile per rappresentare più agevolmente i dati gestiti dai programmi. In particolare essi consentono di descrivere con nomi simbolici gruppi di oggetti ai quali è possibile associare valori numerici interi.

Come noto, le variabili di un programma possono rappresentare non solo oggetti quantificabili, come un importo valutario, ma anche qualità non numerabili (come un colore o il sesso di un individuo) la cui caratteristica principale è il fatto di essere mutuamente esclusive. Normalmente si tende a gestire tali qualità "inventando" una codifica che permette di assegnare valori di tipo integral (gli smemorati tornino a pag. 12) ai loro differenti modi di manifestarsi (ad esempio: al colore nero può essere associato il valore zero, al rosso il valore `1`, e così via; si può utilizzare il carattere `'M'` per "maschile" e `'F'` per "femminile, etc.). Spesso si ricorre alle direttive `#define`, che consentono di associare, mediante la sostituzione di stringhe a livello di preprocessore, un valore numerico ad un nome descrittivo (vedere pag. 44 e seguenti).

L'uso degli enumeratori può facilitare la stesura dei programmi, lasciando al compilatore il compito di effettuare la codifica dei diversi valori assumibili dalle variabili che gestiscono modalità qualitative, e consentendo al programmatore di definire ed utilizzare nomi simbolici per riferirsi a tali valori. Vediamo un esempio:

```
enum SEX {
    ignoto,                                                    // beh, non si sa mai...
    maschile,
    femminile
};
```

La dichiarazione di un enumeratore ricorda da vicino quella di una struttura: anche in questo caso viene definito un template; la parola chiave `enum` è seguita dal tag, cioè dal nome che si intende dare al modello di enumeratore; vi sono le parentesi graffe aperta e chiusa, quest'ultima seguita dal punto e virgola. La differenza più evidente rispetto alla dichiarazione di un template di struttura consiste nel fatto che laddove in questo compaiono le dichiarazioni dei campi (vere e proprie definizioni di variabili con tanto di indicatore di tipo e punto e virgola), nel template di `enum` vi è l'elenco dei nomi simbolici corrispondenti alle possibili manifestazioni della qualità che l'enumeratore stesso rappresenta. Detti nomi simbolici sono separati da virgole; la virgola non compare dopo l'ultimo nome elencato.

Anche la dichiarazione di una variabile di tipo `enum` ricorda da vicino quella di una variabile struttura:

```
    enum SEX sesso;
    ....
    sesso = maschile;
    ....
    if(sesso == maschile)
        printf("MASCHIO");
    else
        if(sesso == femminile)
            printf("FEMMINA");
        else
            printf("BOH?");
```

Il codice riportato chiarisce le modalità di dichiarazione, inizializzazione e, in generale, di utilizzo di una variabile di tipo `enum`.

E' inoltre possibile notare come in C, a differenza di quanto avviene in molti altri linguaggi, l'operatore di assegnamento e quello di confronto per uguaglianza hanno grafia differente, dal momento che quest'ultimo si esprime con il doppio segno di uguale.

Ovviamente il compilatore, di soppiatto, assegna dei valori ai nomi simbolici elencati nel template dell'`enum`: per default al primo nome è associato il valore `0`, al secondo `1`, e così via. E'

comunque possibile assegnare valori a piacere, purché integral, ad uno o più nomi simbolici; ai restanti il valore viene assegnato automaticamente dal compilatore, incrementando di uno il valore associato al nome precedente.

```
enum SEX {
    ignoto = -1,
    maschile,
    femminile
};
```

Nell'esempio, al nome `ignoto` è assegnato esplicitamente valore `-1`: il compilatore assegna valore `0` al nome `maschile` e `1` a `femminile`. I valori esplicitamente assegnati dal programmatore non devono necessariamente essere consecutivi; la sola condizione da rispettare è che si tratti di valori interi.

Il vantaggio dell'uso degli enumeratori consiste nella semplicità di stesura e nella migliore leggibilità del programma, che non deve più contenere dichiarazioni di costanti manifeste né utilizzare variabili intere per esprimere modalità qualitative. Inoltre, la limitazione del fabbisogno di costanti manifeste rappresenta di per sé un vantaggio di carattere tecnico, in quanto consente di limitare i rischi connessi al loro utilizzo, in particolare i cosiddetti *side effect* o effetti collaterali (pag. 45).

I campi di bit

Se una variabile di tipo intero può assumere solo un limitato numero di valori, è teoricamente possibile memorizzarla utilizzando un numero di bit inferiore a quello assegnatole dal compilatore: basta infatti 1 bit per memorizzare un dato che può assumere solo due valori, 2 bit per un dato che può assumere quattro valori, 3 bit per uno che può assumere otto valori, e così via.

Il C non ha tipi intrinseci di dati con un numero di bit inferiori a 8 (il `char`), ma consente di "impaccare" più variabili nel numero di bit strettamente necessario mediante i cosiddetti campi di bit.

Un esempio di uso di questo strumento può essere ricavato con riferimento alla gestione di una cartella clinica. Supponiamo di voler gestire, per ogni paziente, le seguenti informazioni: il sesso (maschile o femminile), lo stato vitale (vivente, defunto, in coma), il tipo di medicinale somministrato (sedici categorie, come antibiotici e sulfamidici), la categoria di ricovero (otto possibili sistemazioni, da corsia a camera di lusso). In questa ipotesi sarebbe possibile codificare il sesso mediante un solo bit, lo stato vitale con 2, il tipo di cura con 4, la sistemazione in ospedale con 3: in totale 10 bit, senz'altro disponibili in un'unica variabile di tipo intero.

L'uso dei campi di bit prevede la dichiarazione di un template: anche in questo caso la somiglianza con le strutture è palese.

```
struct CartellaClinica {
    unsigned sesso: 1;
    unsigned stato: 2;
    unsigned cura: 4;
    unsigned letto: 3;
};
```

La dichiarazione utilizza la parola chiave `struct`, proprio come se si trattasse di un template di struttura; le dichiarazioni dei campi sono introdotte da uno specificatore di tipo e chiusa dal punto e virgola; la differenza qui consiste nell'indicazione dell'ampiezza in bit di ogni singolo campo, effettuata posponendo al nome del campo il carattere *due punti* ("`:`") seguito dal numero di bit da assegnare al campo stesso. I due punti servono, infatti, a indicare la definizione di un campo di bit, la cui ampiezza viene specificata dal numero seguente; se il numero totale di bit non è disponibile in un'unica variabile intera, il compilatore alloca anche la successiva word in memoria.

I campi di bit del tipo `CartellaClinica` sono tutti dichiarati `unsigned int`: in tal modo tutti i bit sono utilizzabili per esprimere i valori che di volta in volta i campi stessi assumeranno. In realtà, i campi di bit possono anche essere dichiarati `int`, ma in questo caso il loro bit più significativo rappresenta il segno e non è quindi disponibile per memorizzare il valore. Un campo dichiarato `int` ed ampio un solo bit può esprimere solo i valori `0` e `-1`.

I campi di bit sono referenziabili esattamente come i campi di una comune struttura:

```
enum SEX {
    maschile,
    femminile
};

struct CartellaClinica {
    unsigned sesso: 1;
    unsigned stato: 2;
    unsigned cura: 4;
    unsigned letto: 3;
};

char *sessi[] = {
    "MASCHILE",
    "FEMMINILE"
};
....
    struct CartellaClinica Paziente;
    ....
    Paziente.sesso = maschile;
    ....
    printf("Sesso del paziente: %s\n",sessi[Paziente.sesso]);
```

E' importante ricordare come sia compito del programmatore assicurarsi che i valori memorizzati nei campi di bit non occupino più bit di quanti ne sono stati loro riservati in fase di definizione del template, dal momento che le regole del C non assicurano che venga effettuato un controllo nelle operazioni di assegnamento di valori ai campi. Se si assegna ad un campo di bit un valore maggiore del massimo previsto per quel campo, può accadere che i bit più significativi di quel valore siano scritti nei campi successivi: è bene, ancora una volta, verificare il comportamento del proprio compilatore (circa le dipendenze del codice dal compilatore utilizzato, vedere pag. 463).

Naturalmente un campo di bit può essere utilizzato anche per memorizzare un'informazione di tipo quantitativo: ad esempio, la `struct CartellaClinica` potrebbe essere ridefinita mediante l'aggiunta di un campo atto a memorizzare il numero di ricoveri subiti dal paziente; impiegando 6 bit tale valore è limitato a 63.

```
struct CartellaClinica {
    unsigned sesso: 1;
    unsigned stato: 2;
    unsigned cura: 4;
    unsigned letto: 3;
    unsigned ricoveri: 6;
};
```

Nella nuova definizione, tutti i 16 bit delle due word occupate in memoria dalla `struct CartellaClinica` sono utilizzati.

GLI OPERATORI

Come tutti i linguaggi di programmazione, il C dispone di un insieme di operatori, cioè di simboli che rappresentano particolari operazioni sul valore di un dato (che viene comunemente detto *operando*).

Alcuni operatori C sono perfettamente equivalenti a quelli omologhi di altri linguaggi, altri sono peculiari; tuttavia, prima di esaminarne le principali caratteristiche, è bene chiarire il significato di due concetti: *precedenza* e *associatività*.

Quando un operatore agisce su più operandi o in un'espressione sono definite più operazioni, tali concetti assumono notevole importanza, perché consentono di interpretare correttamente l'espressione stessa, stabilendo quali operazioni devono essere effettuate prima delle altre. Consideriamo, quale esempio, una somma:

```
a = b + c;
```

Nell'espressione sono presenti due operatori: l'uguale (operatore di assegnamento) ed il "più" (operatore di somma). E' facile comprendere che l'espressione ha significato solo se viene dapprima calcolata la somma dei valori contenuti in `b` e `c`, e solo successivamente il risultato è assegnato ad `a`. Possiamo dire che la `precedenza` dell'operatore di assegnamento è minore di quella dell'operatore di somma.

Consideriamo ora una serie di assegnamenti:

```
a = b = c = d;
```

Il compilatore C la esegue assegnando il valore di `d` a `c`; poi il valore di `c` a `b`; infine, il valore di `b` ad `a`. Il risultato è che il valore di `d` è assegnato in cascata alle altre variabili; in pratica, che l'espressione è stata valutata da destra a sinistra, cioè che l'operatore di assegnamento gode di associatività da destra a sinistra.

In altre parole, la precedenza si riferisce all'ordine in cui il compilatore valuta gli operatori, mentre l'associatività concerne l'ordine in cui sono valutati operatori aventi la stessa precedenza (non è detto che l'ordine sia sempre da destra a sinistra).

Le parentesi tonde possono essere sempre utilizzate per definire parti di espressioni da valutare prima degli operatori che si trovano all'esterno delle parentesi. Inoltre, quando vi sono parentesi tonde annidate, vale la regola che la prima parentesi chiusa incontrata si accoppia con l'ultima aperta e che vengono sempre valutate per prime le operazioni più interne. Così, ad esempio, l'espressione

```
a = 5 * (a + b / (c - 2));
```

è valutata come segue: dapprima è calcolata la differenza tra `c` e `2`, poi viene effettuata la divisione di `b` per tale differenza. Il risultato è sommato ad `a` ed il valore ottenuto è moltiplicato per `5`. Il prodotto, infine, è assegnato ad `a`. In assenza delle parentesi il compilatore avrebbe agito in maniera differente, infatti:

```
a = 5 * a + b / c - 2;
```

è valutata sommando il prodotto di `a` e `5` al quoziente di `b` diviso per `c`; al risultato è sottratto `2` ed il valore così ottenuto viene assegnato ad `a`.

Vale la pena di presentare l'insieme degli operatori C, riassumendone in una tabella le regole di precedenza ed associatività; gli operatori sono elencati in ordine di precedenza decrescente.

OPERATORI C

OPERATORE	DESCRIZIONE	ASSOCIATIVITÀ
`()`	chiamata di funzione	da sx a dx
`[]`	indici di array	
`.`	appartenenza a struttura	
`->`	appartenenza a struttura refernziata da puntatore	
`!`	NOT logico	da dx a sx
`~`	complemento a uno	
`-`	meno unario (negazione)	
`++`	autoincremento	
`--`	autodecremento	
`&`	indirizzo di	
`*`	indirezione	
`(`*`tipo`*`)`	cast (conversione di tipo)	
`sizeof()`	dimensione di	
`*`	moltiplicazione	da sx a dx
`/`	divisione	
`%`	resto di divisione intera	
`+`	addizione	da sx a dx
`-`	sottrazione	
`<<`	scorrimento a sinistra di bit	da sx a dx
`>>`	scorrimento a destra di bit	
`<`	minore di	da sx a dx
`<=`	minore o uguale a	
`>`	maggiore di	

<table>
<tr><td><code>>=</code></td><td>maggiore o uguale a</td><td></td></tr>
<tr><td><code>==</code></td><td>uguale a</td><td>da sx a dx</td></tr>
<tr><td><code>!=</code></td><td>diverso da (NOT uguale a)</td><td></td></tr>
<tr><td><code>&</code></td><td>AND su bit</td><td>da sx a dx</td></tr>
<tr><td><code>^</code></td><td>XOR su bit</td><td>da sx a dx</td></tr>
<tr><td><code>|</code></td><td>OR su bit</td><td>da sx a dx</td></tr>
<tr><td><code>&&</code></td><td>AND logico</td><td>da sx a dx</td></tr>
<tr><td><code>||</code></td><td>OR logico</td><td>da sx a dx</td></tr>
<tr><td><code>? :</code></td><td>espressione condizionale</td><td>da dx a sx</td></tr>
<tr><td><code>=, etc.</code></td><td>operatori di assegnamento (semplice e composti)</td><td>da dx a sx</td></tr>
<tr><td><code>,</code></td><td>virgola (separatore di espressioni)</td><td>da sx a dx</td></tr>
</table>

Come si vede, alcuni operatori possono assumere significati diversi. Il loro modo di agire sugli operandi è quindi talvolta desumibile senza ambiguità solo conoscendo il contesto di azione, cioè le specifiche espressioni in cui sono utilizzati. Di seguito è fornita una descrizione dettagliata degli operatori di cui ancora non si è detto in precedenza, elencati in ordine di precedenza decrescente, come da tabella, ma, al tempo stesso, raggruppati per analogia di significato. Circa l'operatore di chiamata a funzione si veda pag. 85.

Not logico

Il *not logico* si indica con il punto esclamativo. Esso consente di negare logicamente il risultato di un confronto, cioè di "capovolgerlo". Perciò, se ad esempio

```
(a > b)
```

è vera, allora

```
!(a > b)
```

risulta falsa. Ancora, l'espressione seguente

```
if(!(a = b)) ....
```

equivale a

```
a = b;
if(!a) ....
```

che, a sua volta, è l'equivalente di

```
a = b;
if(!(a != 0)) ....
```

cioè, in definitiva,

```
a = b;
if(a == 0) ....
```

Si noti che l'operatore "`!=`", pur essendo formato da due simboli, è per il compilatore un unico *token*[58], la cui grafia, comunque, è perfettamente coerente con il significato dell'operatore "`!`" (vedere quanto detto circa gli operatori logici, pag. 70).

COMPLEMENTO A UNO

L'operatore di *complemento a uno* è rappresentato con la tilde ("`~`"). Il complemento ad uno di un numero si ottiene invertendo tutti i bit che lo compongono: ad esempio, con riferimento a dati espressi con un solo byte, il complemento a uno di `0` è `255`, mentre quello di `2` è `253`. Infatti, rappresentando il byte come una stringa di 8 bit, nel primo caso si passa da `00000000` a `11111111`, mentre nel secondo da `00000010` si ottiene `11111101`. Pertanto

```
a = 2;
printf("%d\n",~a);
```

produce la visualizzazione proprio del numero `253`.

L'operatore di complemento a uno (o negazione binaria) non va confuso né con l'operatore di negazione logica, di cui si è appena detto, né con quello di negazione algebrica o meno unario ("`-`", vedere di seguito), dei quali si è detto poco sopra: del resto, la differenza tra i tre è evidente. Il primo "capovolge" i singoli bit di un valore, il secondo rende nullo un valore non nullo e viceversa, mentre il terzo capovolge il segno di un valore, cioè rende negativo un valore positivo e viceversa.

NEGAZIONE ALGEBRICA

Il segno meno ("`-`") può essere utilizzato come negazione algebrica, cioè per esprimere numeri negativi o, più esattamente, per invertire il segno di un valore: in tal caso esso ha precedenza maggiore di tutti gli operatori aritmetici (vedere pag. 68), per cui

```
a = -b * c;
```

è valutata moltiplicando `c` per il valore di `b` cambiato di segno. Si osservi che le negazione algebrica di un valore non modifica il valore stesso, ma lo restituisce con segno opposto e identico modulo: nell'esempio appena riportato, il valore in `b` non viene modificato.

AUTOINCREMENTO E AUTODECREMENTO

Gli operatori di (auto)incremento e (auto)decremento sommano e, rispettivamente, sottraggono `1` alla variabile a cui sono applicati. L'espressione

```
++a;
```

---

[58] Il token è un'entità minima riconoscibile dal compilatore come parte a se stante di una istruzione.

incrementa di `1` il valore di `a`, mentre

```
--a;
```

lo decrementa. E' molto importante ricordare che essi possono essere *prefissi* o *suffissi*; possono, cioè, sia precedere che seguire la variabile a cui sono applicati. Il loro significato rimane il medesimo (sommare o sottrarre `1`), ma il loro livello di precedenza cambia. Nell'espressione

```
a = ++b;
```

ad `a` viene assegnato il valore di `b` incrementato di `1`, perché, in realtà, dapprima è incrementata la variabile `b` e successivamente il suo nuovo valore è assegnato ad `a`. Invece, con

```
a = b++;
```

ad `a` è assegnato il valore di `b` e solo successivamente questa è incrementata. Analoghe considerazioni valgono nel caso dell'operatore di decremento. Ancora: nella

```
if(a > ++b) ....
```

la condizione è valutata dopo avere incrementato `b`, mentre nella

```
if(a > b++) ....
```

dapprima è valutata la condizione e poi viene incrementata `b`.

La differenza tra operatore prefisso e suffisso, però, scompare quando l'autoincremento della variabile sia parametro di una chiamata a funzione: con riferimento ad una riga come

```
printf("%d\n",++a);
```

spesso non è possibile sapere a priori se `a` viene incrementata prima di passarne il valore a `printf()`, o se, al contrario, l'incremento è effettuato in seguito. Ci si potrebbe aspettare che la scrittura `++a` determini l'incremento prima della chiamata, mentre `a++` lo determini dopo; tuttavia il C non stabilisce una regola univoca. Ciò significa che ogni compilatore può regolarsi come meglio crede. E questo a sua volta significa che possono esserci compilatori che fissano a priori un modo univoco di procedere, ed altri che invece decidono caso per caso in fase di compilazione, sulla base, ad esempio, di opzioni di ottimizzazione del codice in funzione della velocità, della dimensione, e così via. E' dunque indispensabile consultare molto attentamente la documentazione del compilatore o, meglio ancora, evitare possibili ambiguità dedicando all'incremento della variabile un'istruzione separata dalla chiamata a funzione, anche in vista di un possibile *porting* del programma ad altri compilatori (al riguardo vedere anche pag. 463).

Gli operatori "`++`" e "`--`" modificano sempre il valore della variabile[59] a cui sono applicati.

CAST E CONVERSIONI DI TIPO

In una espressione è sempre possibile avere operandi di tipo diverso. Non è poi così strano dividere, ad esempio, un numero in virgola mobile per un numero intero, oppure, anche se a prima vista può sembrare meno ovvio, moltiplicare un intero per un carattere. In ogni caso, comunque, il risultato

[59] Proprio per questo, quindi, non possono essere applicati alle costanti.

dell'operazione deve essere di un unico tipo, di volta in volta ben determinato: in tali casi è sempre necessario, perciò, procedere a conversioni di tipo su almeno uno degli operandi coinvolti.

Il C, al riguardo, fissa un ordine "gerarchico" dei tipi di dato intrinseci, e stabilisce due semplici regole che consentono di conoscere sempre a priori come verranno effettuate le necessarie conversioni.

L'ordine gerachico dei tipi, decrescente da sinistra a destra, è il seguente:

```
long double > double > float > long > int > short > char
```

Ne risulta che ogni tipo è di "grado" superiore ad ogni altro tipo elencato alla sua destra e di grado inferiore a quello dei tipi elencati alla sua sinistra. Sulla scorta di tale gerarchia, la prima regola stabilisce che nelle espressioni che *non* coinvolgono operatori di assegnamento, in ogni coppia di operandi l'operando di grado inferiore è convertito nel tipo dell'operando avente grado superiore. Così, ad esempio, in una operazione di confronto tra un `float` e un `long`, quest'ultimo è convertito in `float` prima che sia effettuato il confronto.

La seconda regola riguarda invece le operazioni di assegnamento: l'espressione a destra dell'operatore di assegnamento è sempre convertita nel tipo della variabile che si trova a sinistra del medesimo, indipendentemente dal livello gerarchico dei dati coinvolti.

Naturalmente le due regole possono trovare contemporanea applicazione quando ad una variabile sia assegnato il risultato di un'espressione che coinvolge operandi di tipi differenti:

```
int iVar;
long lVar;
float fVar;
char cVar;

....
iVar = fVar + lVar * cVar;
```

Nell'esempio, l'operatore di moltiplicazione ha precedenza rispetto a quello di somma, perciò viene dapprima calcolato il prodotto di `lVar` per `cVar`, dopo avere convertito `cVar` in `long`. Il valore ottenuto è poi sommato a quello contenuto in `fVar`, ma solo dopo averlo convertito in `float`. Il risultato, infine, viene convertito in `int` ed assegnato a `iVar`.

Si tenga presente che le conversioni effettuate in modo automatico dal compilatore C implicano un troncamento della parte più significativa del valore convertito quando esso viene "degradato" ad un livello inferiore, ed un'aggiunta di bit nulli quando è "promosso" ad un tipo di livello superiore. Nel secondo caso il valore originario del dato può sempre venire conservato; nel primo, al contrario, esiste il rischio di perdere una parte (la più significativa) del valore convertito.

L'affermazione risulta palese se si pensa, ad esempio, al caso di una conversione da `int` a `long` ed una viceversa: consideriamo due variabili, la prima di tipo int (16 bit) e la seconda di tipo `long` (32 bit), contenenti, rispettivamente, i valori `5027` (che in codice binario è `0001001110100011`) e `2573945` (in binario `00000000001001110100011001111001`): la conversione della prima in `long` implica l'aggiunta di 16 bit nulli alla sinistra di quelli "originali". Lo spazio occupato è ora di 32 bit, ma il valore di partenza non viene modificato. Nel convertire il `long` in `int`, al contrario, vengono eliminati i 16 bit più significativi (quelli più a sinistra): i 16 bit rimanenti sono `0100011001111001`, che equivalgono, in notazione decimale, a `18041`.

Conversioni di tipo automatiche sono effettuate anche quando il tipo dei parametri passati ad una funzione non corrisponde al tipo dei parametri che la funzione "desidera". Inoltre, in questo caso, i `char` sono sempre convertiti in `int`, anche se la funzione si aspetta di ricevere proprio un `char`[60]. Va

---

[60] Qualcuno, probabilmente, se ne domanda il perché. Ebbene, il motivo non è legato ad una improbabile mania del compilatore di convertire tutto quello che gli capiti a tiro, bensì esclusivamente alla natura tecnica del passaggio di parametri ad una funzione, sempre effettuato tramite una particolare area di memoria, lo *stack*, organizzata e gestita in word (vedere pag. 158).

anche sottolineato che il compilatore, in genere, emette un messaggio di warning quando la conversione di tipo generata in modo automatico comporta il rischio di perdere una parte del valore coinvolto.

Vi sono però spesso situazioni in cui il compilatore non è in grado di effettuare la conversione in modo automatico; ad esempio quando sono coinvolti tipi di dato non intrinseci, definiti dal programmatore (quali strutture, campi di bit, etc.). Altre volte, invece, si desidera semplicemente esplicitare una conversione che il compilatore potrebbe risolvere da sé, al fine di rendere più chiaro il codice o per evitare il warning ad essa correlato.

In tutti questi casi si può ricorrere all'operatore di *cast*, il quale forza un qualunque valore ad appartenere ad un certo tipo. La notazione è la seguente:

```
(tipo)espressione
```

dove `tipo` può essere una qualsiasi delle parole chiave del C utilizzate nelle dichiarazioni di tipo ed `espressione` dev'essere una qualsiasi espressione sintatticamente corretta. Ad esempio:

```
int iVar;

iVar = (int)3.14159;
```

La conversione illustrata può essere automaticamente eseguita dal compilatore, ma l'esplicitarla mediante l'operatore di cast incrementa la chiarezza del codice ed evita il messaggio di warning. Un altro caso in cui si effettua spesso il cast è l'inizializzazione di un puntatore `far` o `huge` con una costante a 32 bit:

```
char far *colVbuf = (char far *)0xB8000000L;      // ptr buffer video testo col.
```

La conversione automatica, in questo caso, non comporterebbe alcun errore, dal momento che la costante assegnata al puntatore è un dato a 32 bit, esattamente come il puntatore stesso: il compilatore emetterebbe però una segnalazione di warning, per evidenziare al programmatore che un dato di tipo `long` viene assegnato ad un puntatore `far` a carattere: una questione di forma, insomma. Di fatto la costante potrebbe essere scritta anche senza la "`L`" che ne indica inequivocabilmente la natura `long`, ma in quel caso il compilatore segnalerebbe, con un altro warning, che vi è una costante che, per il valore espresso, deve essere considerata `long` senza che ciò sia stato esplicitamente richiesto.

Più significativo può essere l'esempio seguente:

```
struct FARPTR {
    unsigned offset;
    unsigned segment;
};

    ....
    char far *cFptr;
    struct FARPTR fPtr;
    ....
    (char far *)fPtr = cFptr;
```

In questo caso la struttura di tipo `FARPTR` è utilizzata per accedere separatamente alla parte segmento e alla parte offset di un puntatore `far`. In pratica, il valore contenuto nel puntatore `far` è copiato nell'area di memoria occupata dalla struttura: si tratta di un'operazione che potrebbe provocare l'emissione di un messaggio di errore e l'interruzione della compilazione. La presenza dell'operatore di cast tranquillizza il compilatore; dal canto nostro sappiamo che struttura e puntatore occupano entrambi 32 bit, perciò siamo tranquilli a nostra volta.

OPERATORE sizeof()

Il compilatore C rende disponibile un operatore, `sizeof()`, che restituisce come `int` il numero di byte[61] occupato dal tipo di dato o dalla variabile indicati tra le parentesi. Esempietto:

```
int pippo;
long pluto;
float num;
int bytes_double;
....
printf("pippo occupa %d bytes\n",sizeof(pippo));
printf("infatti un int ne occupa %d\n",sizeof(int));
printf("un long occupa %d bytes\n",sizeof(long));
printf("...e un fload %d\n",sizeof(float));
bytes_double = sizeof(double);
printf("Il double occupa %d bytes\n",bytes_double);
```

Si noti che `sizeof()` non è una funzione, ma un operatore: esso è dunque intrinseco al compilatore e non fa parte di alcuna libreria. Inoltre esso restituisce sempre un valore di tipo `int`, indipendentemente dal tipo di dato o di variabile specificato tra le parentesi.

OPERATORI ARITMETICI

Gli operatori aritmetici del C sono i simboli di addizione ("`+`"), sottrazione ("`-`"), divisione ("`/`") e moltiplicazione ("`*`"), quest'ultimo da non confondere con l'operatore di indirezione (pag. 17) che utilizza il medesimo simbolo. Anche l'utilizzo di tali operatori appare piuttosto scontato; è comunque opportuno sottolineare che tra di essi valgono le normali regole di precedenza algebrica, per cui le operazioni di moltiplicazione e divisione si calcolano, in assenza di parentesi, prima di quelle di addizione e sottrazione. Così, ad esempio, l'espressione

```
a = b + c * 4 - d / 2;
```

è calcolata come

```
a = b + (c * 4) - (d / 2);
```

Vedere anche l'operatore di negazione algebrica, pag. 64.

RESTO DI DIVISIONE INTERA

Quando si effettua una divisione tra due interi, il C restituisce solamente la parte intera del risultato. Se esiste un resto, questo è perso. Ad esempio, l'espressione

```
a = 14 / 3;
```

assegna 4 ad `a`.

Se interessa conoscere il resto della divisione, è necessario utilizzare l'operatore "`%`":

```
a = 14 % 3;
```

[61] In C un dato on può mai occupare una frazione di byte.

assegna ad a il valore 2, cioè il resto dell'operazione; in pratica, l'operatore "%" è complementare all'operatore "/", ma è applicabile esclusivamente tra valori rientranti nella categoria degli integral (pag. 12).

SHIFT SU BIT

Pur essendo classificato normalmente tra i linguaggi di alto livello, il C manifesta spesso la propria natura di linguaggio orientato al sistema: gli operatori su bit di cui dispone sono una delle caratteristiche che contribuiscono a renderlo particolarmente vicino alla macchina. Tali operatori consentono di agire sui dati integral considerandoli semplici sequenze di bit.

Particolarmente interessanti risultano due operatori che permettono di traslare, cioè di "fare scorrere", di un certo numero di posizioni a destra o sinistra i bit di un valore: si tratta dei cosiddetti operatori di *shift*. In particolare, lo shift a sinistra si esprime col simbolo "<<", mentre quello a destra (indovinate un po') con ">>". Esempio:

```
a = 1;
printf("%d\n",a <<= 2);
printf("%d\n",a >> 1);
```

Il frammento di codice riportato produce la visualizzazione dei numeri 4 e 2; infatti, il numero 1 in forma binaria è 00000001. Traslando a sinistra i bit di due posizioni, si ottiene 00000100, che è, appunto, 4. Come si è detto a pag. 73, l'operatore di assegnamento può essere composto con gli operatori su bit: ne segue che la seconda riga di codice modifica il valore di a, assegnandole il suo stesso valore traslato a sinistra di due posizioni. La seconda chiamata a printf() visualizza il valore 2, restituito dall'espressione che trasla a destra di una posizione i bit del valore presente in a (4), ma questa volta a non è modificata.

Va osservato che l'operazione di shift rende privi di significato i primi o gli ultimi bit del valore (a seconda che la traslazione avvenga verso sinistra o, rispettivamente, verso destra)[62]: quegli spazi sono riempiti con bit di valore opportuno. Nel caso di shift a sinistra non vi è mai problema: i bit lasciati liberi sono riempiti con bit a zero; ma nel caso di uno shift a destra le cose si complicano.

Se l'integral su cui è effettuato lo shift è senza segno, o è signed ma positivo, allora anche in questo caso sono utilizzati bit nulli come riempitivo. Se, al contrario, l'integral è di tipo signed ed è negativo, allora va tenuto presente che il suo bit più significativo, cioè quello all'estrema sinistra, è usato proprio per esprimere il segno. Alcuni processori estendono il segno, cioè riempiono i bit lasciati liberi dallo shift con bit a uno; altri invece inseriscono comunque bit nulli. Pertanto, a seconda del calcolatore su cui è eseguita, una operazione di shift a sinistra come la seguente:

```
signed char sc;

sc = -1;                                                     // In bits e' 11111111
sc >>= 4;              // rimane 11111111 con E.S.; diventa 00001111 senza E.S.
```

può avere quale effetto un valore finale per sc pari ancora a -1, se il processore effettua l'estensione del segno, oppure pari a 15 se non vi è estensione di segno. Cautela, dunque: consultare la documentazione della macchina[63] prima di azzardare ipotesi.

---

[62] Detti bit, per farla breve, vengono "spinti fuori" dallo spazio a loro disposizione e si perdono nel nulla.

[63] L'estensione del segno dipende dal processore e non dal compilatore. Questo si limita infatti a utilizzare le istruzioni assembler di shift su bit per codificare opportunamente le istruzioni C che coinvolgono gli operatori di shift. Come è noto, ogni processore ha il "proprio" assembler, pertanto il comportamento della macchina dipende dal

OPERATORI LOGICI DI TEST

Gli operatori logici di test possono essere suddivisi in due gruppi: quelli normalmente usati nei confronti tra valori e quelli utilizzati per collegare i risultati di due confronti. Ecco una breve serie di esempi relativi al primo gruppo:

```
(a == b)                                    // VERA se a e' UGUALE a b
(a != b)                                   // VERA se a e' diversa da b
(a < b)                              // VERA se a e' strettamente minore di b
(a > b)                            // VERA se a e' strettamente maggiore di b
(a <= b)                                 // VERA se a e' minore o uguale a b
(a >= b)                               // VERA se a e' maggiore o uguale a b
```

La grafia di detti operatori ed il loro significato appaiono scontati, ad eccezione, forse, dell'operatore di uguaglianza "==": in effetti i progettisti del C, constatato che nella codifica dei programmi i confronti per uguaglianza sono, generalmente, circa la metà degli assegnamenti, hanno deciso[64] di distinguere i due operatori "raddoppiando" la grafia del secondo per esprimere il primo. Ne segue che

```
a = b;
```

assegna ad a il valore di b, mentre

```
(a == b)
```

esprime una condizione che è vera se le due variabili sono uguali. La differente grafia dei due operatori consente di mortificare, ancora una volta, la povera regola KISS (pag. 2), rendendo possibile scrivere condizioni come

```
if(a = b) ....
```

Per quanto appena detto, è ovvio che tale scrittura non può significare "se `a` è uguale a `b`": si tratta infatti, in realtà, di un modo estremamente succinto per dire

```
a = b;
if(a) ....
```

che, a sua volta, equivale a

```
a = b;
if(a != 0) ....
```

cioè "assegna `b` ad `a`, e se il risultato (cioè il nuovo valore di `a`) è diverso da `0`...", dal momento che il C, ogni qualvolta sia espressa una condizione senza secondo termine di confronto assume che si voglia verificane la non-nullità. Carino, vero?

Veniamo al secondo gruppo. Gli operatori logici normalmente usati per collegare i risultati di due o più confronti sono due: si tratta del *prodotto logico* ("`&&`", o *and*) e della *somma logica* ("`||`", o *or*).

---

significato che tali istruzioni assembly hanno per quel particolare processore. I processori Intel effettuano l'estensione del segno.

[64] Una decisione davvero insignificante? No. Vedremo tra poco il perché.

```
(a < b && c == d)                              // AND: vera se entrambe sono VERE
(a < b || c == d)                               // OR: vera se ALMENO UNA e' VERA
```

E' possibile scrivere condizioni piuttosto complesse, ma vanno tenute presenti le regole di precedenza ed associatività. Ad esempio, poiché tutti gli operatori del primo gruppo hanno precedenza maggiore di quelli del secondo, la

```
(a < b && c == d)
```

è equivalente alla

```
((a < b) && (c == d)
```

Nelle espressioni in cui compaiono sia "`&&`" che "`||`" va ricordato che il primo ha precedenza rispetto al secondo, perciò

```
(a < b || c == d && d > e)
```

equivale a

```
((a < b) || ((c == d) && (d < e)))
```

Se ne trae, se non altro, che in molti casi usare le parentesi, anche quando non indispensabile, è sicuramente utile, dal momento che incrementa in misura notevole la leggibilità del codice e abbatte la probabilità di commettere subdoli errori logici.

OPERATORI LOGICI SU BIT

Gli operatori logici su bit consentono di porre in relazione due valori mediante un confronto effettuato bit per bit. Consideriamo l'operatore di prodotto logico, o *and su bit*. Quando due bit sono posti in AND, il risultato è un bit nullo a meno che entrambi i bit valgano `1`. La tabella illustra tutti i casi possibili nel prodotto logico di due bit, a seconda dei valori che cisacuno di essi può assumere. L'operazione consistente nel porre in AND due valori è spesso indicata col nome di "mascheratura", in quanto essa ha l'effetto di nascondere in modo selettivo alcuni bit: in particolare viene convenzionalmente chiamato "maschera" il secondo valore. Se nella maschera è presente uno zero, nel risultato c'è sempre uno zero in quella stessa posizione, mentre un 1 nella maschera lascia inalterato il valore del bit originario. Supponiamo, ad esempio, di voler considerare solo gli 8 bit meno significativi di un valore a 16 bit:

```
unsigned word;
char byte;

word = 2350;
byte = word & 0xFF;
```

Il valore `2350`, espresso in 16 bit, risulta `0000100100101110`, mentre `FFh` è `0000000011111111`. L'operazione di prodotto logico è rappresentabile come

```
0000100100101110 &
0000000011111111 =
------------------
0000000000101110
```

ed il risultato è `46`. Dall'esempio si trae inoltre che il simbolo dell'operatore di and su bit è il carattere "`&`": il contesto in cui viene utilizzato consente facilmente di distinguerlo a prima vista dall'operatore *address of*, che utilizza il medesimo simbolo pur avendo significato completamente diverso (pag. 17).

Più sottile appare la differenza dall'operatore di and logico, sebbene questo abbia grafia differente ("`&&`", pag. 70). L'and su bit agisce proprio sui singoli bit delle due espressioni, mentre l'and logico collega i valori logici delle medesime (vero o falso). Ad esempio, l'espressione

```
((a > b) && c)
```

restituisce un valore diverso da `0` se `a` è maggiore di `b` e, contemporaneamente, `c` è diversa da `0`. Invece, l'espressione

```
((a > b) & c)
```

restituisce un valore diverso da `0` se `a` è maggiore di `b` e, contemporaneamente, `c` è dispari. Infatti un'espressione vera restituisce `1`, e tutti i valori dispari hanno il bit meno significativo ad `1`, pertanto il prodotto logico ha un bit ad `1` (quello meno significativo, ed è dunque diverso da `0`) solo se entrambe le condizioni sono vere.

L'operatore di *or su bit* è invece utilizzato per calcolare quella che viene comunemente indicata come somma logica di due valori. Quando due bit vengono posti in OR, il risultato è sempre `1`, tranne il caso in cui entrambi i bit sono a `0`. Il comportamento dell'operatore di somma logica è riassunto nella tabella. Si noti che il concetto di maschera può essere validamente applicato anche alle operazioni di OR tra due valori, in particolare quando si voglia assegnare il valore `1` ad uno o più bit di una variabile. Infatti la presenza di un `1` nella maschera porta ad `1` il corrispondente bit del risultato, mentre uno `0` nella maschera lascia inalterato il bit del valore originario (questo comportamento è l'esatto opposto di quello dell'operatore "`&`").

L'operazione di OR sui bit dei valori `2350` e `255` (`FFh`) è rappresentabile come segue:

```
0000100100101110 |
0000000011111111 =
------------------
0000100111111111
```

AND	**0**	**1**
0	0	0
1	0	1

e restituisce `2599`. Il simbolo dell'operatore di or su bit è "`|`", e non va confuso con quello dell'operatore di or logico ("`||`", pag. 70); del resto tra i due operatori esistono differenze di significato del tutto analoghe a quelle accennate poco fa circa gli operatori di and su bit e di and logico.

Esiste un terzo operatore logico su bit: l'operatore di *xor su bit*, detto anche "or esclusivo". Il suo simbolo è un accento circonflesso ("`^`"). Un'operazione di XOR tra due bit fornisce risultato `0` quando i due bit hanno uguale valore (cioè sono entrambi `1` o entrambi `0`), mentre restituisce 1 quando i bit hanno valori opposti (il primo `1` ed il secondo `0`, o viceversa): la tabella evidenzia quanto affermato. Se ne trae che la presenza di un `1` in una maschera utilizzata in XOR, dunque, inverte il bit corrispondente del valore originario. Rispolverando ancora una volta (con la solenne promessa che sarà l'ultima) l'esempio del valore `2350` mascherato con un `255`, si ha:

XOR	**0**	**1**
0	0	1
1	1	0

OR	**0**	**1**
0	0	1
1	1	1

```
0000100100101110 ^
0000000011111111 =
------------------
0000100111010001
```

Il risultato è `2513`.

OPERATORE CONDIZIONALE

L'operatore condizionale, detto talvolta operatore ternario in quanto lavora su tre operandi[65], ha simbolo `"? :"` e può essere paragonato ad una forma abbreviata della struttura di controllo `if....else` (pag. 75). La sua espressione generale è:

```
espressione1 ? espressione2 : espressione3
```

la quale significa: "se `espressione1` è vera (cioè il suo valore è diverso da `0`) restituisci `espressione2`, altrimenti restituisci `espressione3`".

Ad esempio, l'istruzione

```
printf("%c\n",(carat >= ' ') ? carat : '.');
```

visualizza il valore di `carat` come carattere solo se questo segue, nella codifica ASCII, lo spazio o è uguale a questo. Negli altri casi è visualizzato un punto.

L'operatore condizionale consente di scrivere codice più compatto ed efficiente di quanto sia possibile fare con la `if....else`, penalizzando però la leggibilità del codice.

ASSEGNAMENTO

L'operatore di assegnamento per eccellenza è l'uguale (`"="`), che assegna alla variabile alla propria sinistra il risultato dell'espressione alla propria destra. Data l'intuitività del suo significato ed utilizzo, non è il caso di dilungarsi su di esso: vale piuttosto la pena di considerarne l'utilizzo combinato con operatori aritmetici.

In tutti i casi in cui vi sia un'espressione del tipo

```
a = a + b;
```

in cui, cioè, la variabile a sinistra dell'uguale compaia anche nell'espressione che gli sta a destra, è possibile utilizzare una forma abbreviata che si esprime "componendo" l'operatore di assegnamento con l'uguale e l'operatore dell'espressione. Si parla allora di operatori di assegnamento composti, in contrapposizione all'operatore di assegnamento semplice (il segno di uguale). Come al solito un esempio è più chiaro di qualunque spiegazione; l'espressione riportata poco sopra diventa:

```
a += b;
```

Formalizzando il tutto, un assegnamento del tipo

```
variabile = variabile operatore espressione
```

può essere scritta (ma non si è obbligati a farlo)

```
variabile operatore = espressione
```

Ecco l'elenco di tutti gli operatori di assegnamento composti:

---

[65] E' l'unico operatore C a presentare tale modalità di utilizzo.

```
+=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

Essi consentono di ottenere espressioni forse un po' criptiche, ma sicuramente assai concise.

SEPARATORE DI ESPRESSIONI

In una sola istruzione C è possibile raggruppare più espressioni, non collegate tra loro da operatori logici, ma semplicemente elencate in sequenza per rendere più compatto (ma meno leggibile) il codice. Esempio:

```
int i, j, k;

i = 0, j = 2, k =6;
```

La riga che inizializza le tre variabili è equivalente alle tre inizializzazioni eseguite in tre diverse righe. La virgola (`","`) agisce da separatore di espressioni e fa sì che queste siano eseguite in sequenza da sinistra a destra. Consideriamo ora l'istruzione che segue:

```
printf("%d\n",i = 5, j = 4, k = 8);
```

Che cosa visualizza `printf()`? Inutile tirare a indovinare, esiste una regola ben precisa. L'operatore "virgola" restituisce sempre il risultato dell'ultima espressione valutata; in questo caso il valore `8`, che è passato a `printf()` come parametro.

L'operatore di separazione di espressioni viene spesso utilizzato quando sia necessario inizializzare più contatori in entrata ad un ciclo.

Il flusso elaborativo

Qualsiasi programma può venire codificato in un linguaggio di programmazione usando tre sole modalità di controllo del flusso elaborativo: l'esecuzione sequenziale, l'esecuzione condizionale e i cicli.

L'esecuzione sequenziale è la più semplice delle tre e spesso non viene pensata come una vera e propria modalità di controllo; infatti è logico attendersi che, in assenza di ogni altra specifica, la prossima istruzione ad essere eseguita sia quella che nella codifica segue quella attuale.

Le altre due strutture di controllo richiedono invece qualche approfondimento.

Le istruzioni di controllo condizionale

Il linguaggio C dispone di due diversi strumenti per condizionare il flusso di esecuzione dei programmi. Vale la pena di analizzarli compiutamente.

if...else

L'esecuzione condizionale nella forma più semplice è specificata tramite la parola chiave `if`, la quale indica al compilatore che l'istruzione seguente deve essere eseguita se la condizione, sempre specificata tra parentesi, è vera. Se la condizione non è verificata, allora l'istruzione non è eseguita e il flusso elaborativo salta all'istruzione successiva. L'istruzione da eseguire al verificarsi della condizione può essere una sola linea di codice, chiusa dal punto e virgola, oppure un blocco di linee di codice, ciascuna conclusa dal punto e virgola e tutte quante comprese tra parentesi graffe. Esempietto:

```
if(a == b)
    printf("a è maggiore di b\n");
if(a == c) {
    printf("a è maggiore di c\n");
    a = c;
}
```

Nel codice riportato, se il valore contenuto in `a` è uguale a quello contenuto in `b` viene visualizzata la stringa `"a è maggiore di b"`; in caso contrario la chiamata a `printf()` non è eseguita e l'elaborazione prosegue con la successiva istruzione, che è ancora una `if`. Questa volta, se `a` è uguale a `c` viene eseguito il blocco di istruzioni comprese tra le parentesi graffe, altrimenti esso è saltato "a pié pari" e il programma prosegue con la prima istruzione che segue la graffa chiusa.

Come regola generale, una condizione viene espressa tramite uno degli operatori logici del C (vedere pag. 70) ed è sempre racchiusa tra parentesi tonde.

La `if` è completata dalla parola chiave `else`, che viene utilizzata quando si devono definire due possibilità alternative; inoltre più strutture `if...else` possono essere annidate qualora serva effettuare test su più "livelli" in cascata:

```
if(a == b)
    printf("a è maggiore di b\n");
else {
    printf("a è minore o uguale a b\n")
    if(a < b)
        printf("a è proprio minore di b\n");
    else
        printf("a è proprio uguale a b\n");
}
```

Quando è presente la `else`, se la condizione è vera viene eseguito solo ciò che sta tra la `if` e la `else`; in caso contrario è eseguito solo il codice che segue la `else` stessa. L'esecuzione dei due blocchi di codice è, in altre parole, alternativa.

E' estremamente importante ricordare che ogni `else` viene dal compilatore riferita all'ultima `if` incontrata: quando si annidano costruzioni `if...else` bisogna quindi fare attenzione alla costruzione logica delle alternative. Cerchiamo di chiarire il concetto con un esempio.

Supponiamo di voler codificare in C il seguente algoritmo: se `a` è uguale a `b` allora si controlla se `a` è maggiore di `c`. Se anche questa condizione è vera, si visualizza un messaggio. Se invece la prima delle due condizioni è falsa, cioè `a` non è uguale a `b`, allora si assegna a `c` il valore di `b`. Vediamo ora un'ipotesi di codifica:

```
if(a == b)
    if(a > c)
        printf("a è maggiore di c\n");
else
    c = b;
```

I rientri dal margine sinistro delle diverse righe evidenziano che le intenzioni sono buone: è immediato collegare, da un punto di vista visivo, la `else` alla prima `if`. Peccato che il compilatore non si interessi affatto alle indentazioni: esso collega la `else` alla seconda `if`, cioè all'ultima `if` incontrata. Bisogna correre ai ripari:

```
if(a == b)
    if(a > c)
        printf("a è maggiore di c\n");
    else;
else
    c = b;
```

Quella appena vista è una possibilità. Introducendo una `else` "vuota" si raggiunge lo scopo, perché questa è collegata all'ultima `if` incontrata, cioè la seconda. Quando il compilatore incontra la seconda `else`, l'ultima `if` non ancora "completa", risalendo a ritroso nel codice, è la prima delle due. I conti tornano... ma c'è un modo più elegante.

```
if(a == b) {
    if(a > c)
        printf("a è maggiore di c\n");
}
else
    c = b;
```

In questo caso le parentesi graffe indicano chiaramente al compilatore qual è la parte di codice che dipende direttamente dalla prima `if` e non vi è il rischio che la `else` sia collegata alla seconda, dal momento che questa è interamente compresa nel blocco tra le graffe e quindi è sicuramente "completa".

Come si vede, salvo alcune particolarità, nulla diversifica la logica della `if` del C da quella delle `if` (o equivalenti parole chiave) disponibili in altri linguaggi di programmazione[66].

---

[66] Tra l'altro, in C la `if` non è mai seguita da una parola chiave tipo "`then`", o simili. Può sembrare banale sottolinearlo, ma chi viene ad esempio dalla programmazione in Basic sembra convincersene con qualche difficoltà.

switch

La `if` gestisce ottimamente quelle situazioni in cui, a seguito della valutazione di una condizione, si presentano due sole possibilità alternative. Quando le alternative siano più di due, si è costretti a utilizzare più istruzioni `if` nidificate, il che può ingarbugliare non poco la struttura logica del codice e menomarne la leggibilità.

Quando la condizione da valutare sia esprimibile mediante un'espressione restituente un `int` o un `char`, il C rende disponibile l'istruzione `switch`, che consente di valutare un numero qualsiasi di alternative per il risultato di detta espressione. Diamo subito un'occhiata ad un caso pratico:

```
#define EOF   -1
#define LF    10
#define CR    13
#define BLANK ' '
....
    char c;
    long ln = 0L, cCount = 0L;
    ....
    switch(c = fgetc(inFile)) {
        case EOF:
            return;
        case LF:
            if(++ln == MaxLineNum)
                return;
        case BLANK:
            cCount++;
        case NULL:
        case CR:
            break;
        default:
            *ptr++ = c;
    }
```

Il frammento di codice riportato fa parte di una funzione che legge il contenuto di un file carattere per carattere ed esegue azioni diverse a seconda del carattere letto: in particolare, la funzione `fgetc()` legge un carattere dal file associato al descrittore[67] `inFile` e lo restituisce. Tale carattere è memorizzato nella variabile `c`, dichiarata di tipo `char`. L'operazione di assegnamento è, in C, un'espressione che restituisce il valore assegnato, pertanto il valore memorizzato nella variabile `c` è valutato dalla `switch`, che esegue una delle possibili alternative definite. Se si tratta del valore definito dalla costante manifesta `EOF` la funzione termina; se si tratta del carattere definito come `LF` viene valutato quante righe sono già state scandite per decidere se terminare o no; se si tratta di un `LF` o di un `BLANK` è incrementato un contatore; i caratteri definiti come `CR` e `NULL` (il solito zero binario) vengono semplicemente ignorati; qualsiasi altro carattere è copiato in un buffer il cui puntatore è incrementato di conseguenza.

E' meglio scendere in maggiori dettagli. Per prima cosa va osservato che l'espressione da valutare deve trovarsi tra parentesi tonde. Inoltre il corpo della `switch`, cioè l'insieme delle alternative, è racchiuso tra parentesi graffe. Ogni singola alternativa è definita dalla parola chiave `case`, seguita da una costante (non sono ammesse variabili o espressioni non costanti) intera (o `char`), a sua volta seguita dai due punti ("`:`"). Tutto ciò che segue i due punti è il codice che viene eseguito qualora l'espressione

---

[67] In C, uno dei modi per manipolare il contenuto di un file consiste nell'aprire uno *stream*, cioè un "canale di flusso" col file stesso mediante un'apposita funzione, che restituisce il puntatore ad una struttura i cui campi sono utilizzati poi da altre funzioni, tra cui la `fgetc()`, per compiere operazioni sul file stesso. Tale puntatore è detto "descrittore" del file: vedere pag. 116.

valutata assuma proprio il valore della costante tra la case e i due punti, fino alla prima istruzione break incontrata (se incontrata!), la quale determina l'uscita dalla switch, cioè un salto alla prima istruzione che segue la graffa chiusa. La parola chiave default seguita dai due punti introduce la sezione di codice da eseguire qualora l'espressione non assuma nessuno dei valori specificati dalle diverse case. Ma non finisce qui.

Tra le parentesi graffe deve essere specificata almeno una condizione: significa che la switch potrebbe essere seguita anche da una sola case o dalla default, e che quindi possono esistere delle switch prive di default o di case. La default, comunque, se presente è unica. Complicato? Più a parole che nei fatti...

Torniamo all'esempio: cosa accade se c vale EOF? viene eseguito tutto ciò che segue i due punti, cioè l'istruzione return. Questa ci "catapulta" addirittura fuori dalla funzione eseguita in quel momento, quindi della switch non siparla proprio più...

Se invece c vale LF, l'esecuzione salta alla if che segue immediatamente la seconda case. Se la condizione valutata dalla if è vera... addio funzione; altrimenti l'esecuzione prosegue con l'istruzione immediatamente successiva. E' molto importante sottolineare che, a differenza di quanto si potrebbe pensare, la presenza di altre case non arresta l'esecuzione e non produce l'uscita dalla switch: viene quindi incrementata la variabile cCount. Solo a questo punto l'istruzione break determina l'uscita dalla switch.

L'incremento della cCount è invece la prima istruzione eseguita se c vale BLANK, ed è anche... l'ultima perché subito dopo si incontra la break. Se c vale CR o NULL si incontra immediatamente la break, e quindi si esce subito dalla switch. Da ciò si vede che quando in una switch è necessario trattare due possibili casi in modo identico è sufficiente accodare le due case. Infine, se in c non vi è nessuno dei caratteri esaminati, viene eseguito ciò che segue la default.

E' forse superfluo precisare che le break, se necessario, possono essere più di una e possono dipendere da altre condizioni valutate all'interno di una case, ad esempio mediante una if. Inoltre una case può contenere un'intera switch, nella quale ne può essere annidata una terza... tutto sta a non perdere il filo logico dei controlli. Esempio veloce:

```
switch(a) {
    case 0:
        switch(b) {
            case 25:
                ....
                break;
            case 30:
            case 31:
                ....
            case 40:
                ....
                break;
            default:
                ....
        }
        ....
        break;
    case 1:
        ....
        break;
    case 2:
        ....
}
```

Se a è uguale a 0 viene eseguita la seconda switch, al termine della quale si rientra nella prima (e sempre nella parte di codice dipendente dalla case per 0). La prima switch, inoltre, non ha la

`default`: se `a` non vale `0`, né `1`, né `2` l'esecuzione salta direttamente alla prima istruzione che segue la graffa che la chiude.

I blocchi di istruzioni dipendenti da una `case`, negli esempi visti, non sono mai compresi tra graffe. In effetti esse non sono necessarie (ma lo sono, ripetiamolo, per aprire e chiudere la `switch`), però, se presenti, non guastano. In una parola: sono facoltative.

goto

Il C supporta un'istruzione che ha il formato generale:

```
goto etichetta;
....
etichetta:
```

oppure:

```
etichetta:
....
goto etichetta;
```

L'`etichetta` può essere un qualsiasi nome (sì, anche `Pippo` o `PLUTO`) ed è seguita dai due punti ("`:`"). L'istruzione `goto` è detta "di salto incondizionato", perché quando viene eseguita il controllo passa immediatamente alla prima istruzione che segue i due punti che chiudono l'`etichetta`. E' però possibile saltare ad una etichetta solo se si trova all'interno della stessa funzione in cui si trova la `goto`; non sono consentiti salti interfunzione.

Per favore, non usate mai la `goto`. Può rendere meno chiaro il flusso elaborativo alla lettura del listato ed è comunque sempre[68] possibile ottenere lo stesso risultato utilizzando un'altra struttura di controllo tra quelle disponibili, anche se talvolta è meno comodo.

La giustificazione più usuale all'uso di `goto` in un programma C è relativa alla possibilità di uscire immediatamente da cicli annidati al verificarsi di una data condizione, ma anche in questi casi è preferibile utilizzare metodi alternativi.

I CICLI

Il linguaggio C dispone anche di istruzioni per il controllo dei cicli: con esse è possibile forzare l'iterazione su blocchi di codice più o meno ampi.

while

Mediante l'istruzione `while` è possibile definire un ciclo ripetuto finché una data condizione risulta vera. Vediamo subito un esempio:

```
while(a < b) {
    printf("a = %d\n",a);
    ++a;
}
```

[68] Sempre vuol dire... proprio sempre!

Le due righe comprese tra le graffe sono eseguite finché la variabile a, incremento dopo incremento, diventa uguale a b. A questo punto l'esecuzione prosegue con la prima istruzione che segue la graffa chiusa.

Vale la pena di addentrarsi un poco nell'algoritmo, esaminando con maggiore dettaglio ciò che accade. Come prima operazione viene valutato se a è minore di b (la condizione deve essere espressa tra parentesi tonde). Se essa risulta vera vengono eseguiti la printf() e l'autoincremento di a, per ritornare poi al confronto tra a e b. Se la condizione è vera il ciclo è ripetuto, altrimenti si prosegue, come già accennato, con quanto segue la parentesi graffa chiusa.

Se ne trae, innanzitutto, che se al primo test la condizione non è vera, il ciclo non viene eseguito neppure una volta. Inoltre è indispensabile che all'interno delle graffe accada qualcosa che determini le condizioni necessarie per l'uscita dal ciclo: in questo caso i successivi incrementi di a rendono falsa, prima o poi, la condizione da cui tutto il ciclo while dipende.

Esiste però un altro metodo per abbandonare un ciclo al verificarsi di una certa condizione: si tratta dell'istruzione break[69]. Esempio:

```
while(a < b) {
    printf("a = %d\n",a);
    if(++a == 100)
        break;
    --c;
}
```

In questo caso a è incrementata e poi confrontata con il valore 100: se uguale, il ciclo è interrotto, altrimenti esso prosegue con il decremento di c. E' anche possibile escludere dall'esecuzione una parte del ciclo e forzare il ritorno al test:

```
while(a < b) {
    if(a++ < c)
        continue;
    printf("a = %d\n",a);
    if(++a == 100)
        break;
    --c;
}
```

Nell'ultimo esempio presentato, a viene confrontata con c ed incrementata. Se, prima dell'incremento essa è minore di c il flusso elaborativo ritorna al test dell'istruzione while; la responsabile del salto forzato è l'istruzione continue, che consente di iniziare da capo una nuova iterazione. In caso contrario viene chiamata printf() e, successivamente, viene effettuato il nuovo test con eventuale uscita dal ciclo.

I cicli while possono essere annidati:

```
while(a < b) {
    if(a++ < c)
        continue;
    printf("a = %d\n",a);
    while(c < x)
        ++c;
    if(++a == 100)
        break;
```

[69] L'istruzione break, se usata in una struttura di controllo switch (pag. 77), determina l'uscita dalla stessa. Si può dire che la break, laddove lecita, ha sempre lo scopo di interrompere la fase elaborativa corrente per proseguire con il normale flusso del programma: essa esercita infatti la medesima funzione nei cicli do...while (vedere di seguito) e for (pag. 81).

```
        --c;
    }
```

All'interno del ciclo per `(a > b)` ve n'è un secondo, per `(c < x)`. Già nella prima iterazione del ciclo "esterno", se la condizione `(c < x)` è vera si entra in quello "interno", che viene interamente elaborato (cioè `c` è incrementata finché assume valore pari ad `x`) prima che venga eseguita la successiva istruzione del ciclo esterno. In pratica, ad ogni iterazione del ciclo esterno avviene una serie completa di iterazioni nel ciclo interno.

Va sottolineato che eventuali istruzioni `break` o `continue` presenti nel ciclo interno sono relative esclusivamente a quest'ultimo: una `break` produrrebbe l'uscita dal ciclo interno e una `continue` il ritorno al test, sempre del ciclo interno.

Si può ancora notare, infine, che il ciclo per `(c < x)` si compone di una sola istruzione: proprio per questo motivo è stato possibile omettere le parentesi graffe.

do...while

I cicli di tipo `do...while` sono, come si può immaginare, "parenti stretti" dei cicli di tipo `while`. Vediamone subito uno:

```
    do {
        if(a++ < c)
            continue;
        printf("a = %d\n",a);
        while(c < x)
            ++c;
        if(++a == 100)
            break;
        --c;
    } while(a < b);
```

Non a caso è stato riportato qui uno degli esempi utilizzati poco sopra con riferimento all'istruzione `while`: in effetti i due cicli sono identici in tutto e per tutto, tranne che per un particolare. Nei cicli di tipo `do...while` il test sulla condizione è effettuato al termine dell'iterazione, e non all'inizio: ciò ha due conseguenze importanti.

In primo luogo un ciclo `do...while` è eseguito sempre almeno una volta, infatti il flusso elaborativo deve percorrere tutto il blocco di codice del ciclo prima di giungere a valutare per la prima volta la condizione. Se questa è falsa il ciclo non viene ripetuto e l'elaborazione prosegue con la prima istruzione che segue la `while`, ma resta evidente che, comunque, il ciclo è già stato compiuto una volta.

In secondo luogo l'istruzione `continue` non determina un salto a ritroso, bensì in avanti. Essa infatti forza in ogni tipo di ciclo un nuovo controllo della condizione; nei cicli `while` la condizione è all'inizio del blocco di codice, e quindi per poterla raggiungere da un punto intermedio di questo è necessario un salto all'indietro, mentre nei cicli `do...while` il test è a fine codice, e viene raggiunto, ovviamente, con un salto in avanti.

Per ogni altro aspetto del comportamento dei cicli `do...while`, in particolare l'istruzione `break`, valgono le medesime considerazioni effettuate circa quelli di tipo `while`.

for

Tra le istruzioni C di controllo dei ciclo, la `for` è sicuramente la più versatile ed efficiente. La `for` è presente in tutti (o quasi) i linguaggi, ma in nessuno ha la potenza di cui dispone in C. Infatti, in generale, i cicli di tipo `while` e derivati sono utilizzati nelle situazioni in cui non è possibile conoscere a

priori il numero esatto di iterazioni, mentre la for, grazie alla sua logica "*punto di partenza; limite; passo d'incremento*", si presta proprio ai casi in cui si può determinare in partenza il numero di cicli da compiere.

Nella for del C è ancora valida la logica a tre coordinate, ma, a differenza della quasi totalità dei linguaggi di programmazione, esse sono reciprocamente svincolate e non necessarie. Ciò significa che, se in Basic[70] la for agisce su un'unica variabile, che viene inizializzata e incrementata (o decrementata) sino al raggiungimento di un limite prestabilito, in C essa può manipolare, ad esempio, tre diverse variabili (o meglio, tre espressioni di diverso tipo); inoltre nessuna delle tre espressioni deve necessariamente essere specificata: è perfettamente lecita una for priva di condizioni di iterazione.

A questo punto, tanto vale esaurire le banalità formali, per concentrarsi poi sulle possibili modalità di definizione delle tre condizioni che pilotano il ciclo. Sia subito detto, dunque, che anche la for vuole che le condizioni siano specificate tra parentesi tonde e che se il blocco di codice del ciclo comprende più di una istruzione sono necessarie le solite graffe, aperta e chiusa. Anche nei cicli for possiamo utilizzare le istruzioni break e continue: la prima per "saltar fuori" dal ciclo; la seconda per tornare "a bomba" alla valutazione del test. Anche i cicli for possono essere annidati, e va tenuto presente che il ciclo più interno compie una serie completa di iterazioni ad ogni iterazione di quello che immediatamente lo contiene.

E vediamo, finalmente, qualche ciclo for dal vivo: nella sua forma banale, quasi "Basic-istica", può assumere il seguente aspetto:

```
for(i = 1; i < k; i++) {
    ....
}
```

Nulla di particolare. Prima di effettuare la prima iterazione, la variabile i è inizializzata a 1. Se essa risulta minore della variabile k il ciclo è eseguito una prima volta. Al termine di ogni iterazione essa è incrementata e successivamente confrontata con la k; se risulta minore di quest'ultima il ciclo è ripetuto.

Vale la pena di evidenziare che le tre coordinate logiche stanno tutte quante all'interno delle parentesi tonde e sono separate tra loro dal punto e virgola (";"); solo la sequenza (;;) deve obbligatoriamente essere presente in un ciclo for.

In effetti possiamo avere una for come la seguente:

```
for( ; ; ) {
    ....
}
```

Qual è il suo significato? Nulla è inizializzato. Non viene effettuato alcun test. Non viene modificato nulla. Il segreto consiste nel fatto che l'assenza di test equivale a condizione sempre verificata: la for dell'esempio definisce quindi un'iterazione infinita. Il programma rimane intrappolato nel ciclo finché si verifica una condizione che gli consenta di abbandonarlo in altro modo, ad esempio con l'aiuto di una break.

Ma si può fare di meglio...

```
for(i = 0; string[i]; )
    ++i;
```

Il ciclo dell'esempio calcola la lunghezza della stringa (terminatore nullo escluso). Infatti i è inizializzata a 0 e viene valutato se il carattere ad offset 0 in string è nullo; se non lo è viene eseguita l'unica istruzione del ciclo, che consiste nell'incrementare i. A questo punto è valutato se è nullo il byte

[70] Un linguaggio a caso? No... il Basic lo conoscono (quasi) tutti...

ad offset `1` in string, e così iterando finché `string[i]` non è proprio il `NULL` finale. L'esempio appena presentato è del tutto equivalente a

```
for(i = 0; string[i]; i++);
```

Il punto e virgola che segue la parentesi tonda indica che non vi sono istruzioni nel ciclo. Le sole cose da fare sono, perciò, la valutazione della condizione e l'incremento di `i` finché, come nel caso precedente, `string[i]` non punta al `NULL` che chiude la stringa. Se poi volessimo includere nel calcolo anche il `NULL`, ecco come fare:

```
for(i = 0; string[i++]; );
```

Sissignori, tutto qui. Anche questo ciclo non contiene alcuna istruzione; tuttavia, in questo caso, l'incremento di `i` fa parte della condizione e (trattandosi di un postincremento; vedere pag. 64) viene effettuato dopo la valutazione, quindi anche (per l'ultima volta) quando `string[i]` punta al `NULL`. E che dire della prossima?

```
for( ; *string++; ) {
    ....
}
```

Nulla di particolare, in fondo: viene verificato se `*string` è un byte non nullo e `string` è incrementato. Se la verifica dà esito positivo viene eseguito il codice del ciclo. Viene poi nuovamente effettuata la verifica, seguita a ruota dall'incremento, e così via. Quanti si sono accorti che questo ciclo `for` è assolutamente equivalente a un ciclo `while`? Eccolo:

```
while(*string++) {
    ....
}
```

In effetti si potrebbe dire che l'istruzione `while`, in C, è assolutamente inutile, in quanto può essere sempre sostituita dalla `for`, la quale, anzi, consente generalmente di ottenere una codifica più compatta ed efficiente dell'algoritmo. La maggiore compattezza deriva dalla possibilità di utilizzare contestualmente alla condizione, se necessario, anche un'istruzione di inizializzazione ed una di variazione. La maggiore efficienza invece dipende dal comportamento tecnico del compilatore, il quale, se possibile, gestisce automaticamente i contatori dei cicli `for` come variabili `register` (vedere pag. 36).

Gli esempi potrebbero continuare all'infinito, ma quelli presentati dovrebbero essere sufficienti per evidenziare, almeno a grandi linee, le caratteristiche salienti dei cicli definiti mediante l'istruzione `for`. E' forse il caso di sottolineare ancora una volta che il contenuto delle parentesi tonde dipende fortemente dal ciclo che si vuole eseguire e dall'assetto elaborativo che gli si vuole dare, ma l'uso dei due punto e virgola è obbligatorio. Il primo e l'ultimo parametro non devono essere necessariamente inizializzare ed incrementare (o decrementare) il contatore (o il medesimo contatore), così come il parametro intermedio non deve per forza essere una condizione da valutare. Ciascuno di questi parametri può essere una qualunque istruzione C o può venire omesso. Il compilatore, però, interpreta sempre il parametro di mezzo come una condizione da verificare, indipendentemente da ciò che è in realtà: detto parametro è quindi sempre valutato come vero o falso[71], e da esso dipendono l'ingresso nel ciclo e le successive iterazioni.

[71] E' bene ricordare che per il compilatore C è falso lo zero binario e vero qualunque altro valore.

LE FUNZIONI

La funzione è l'unità elaborativa fondamentale dei programmi C. Dal punto di vista tecnico essa è un blocco di codice a sé stante, isolato dal resto del programma, in grado di eseguire un particolare compito. Essa riceve dati e fornisce un risultato: ciò che avviene al suo interno è sconosciuto alla rimanente parte del programma, con la quale non vi è mai alcuna interazione.

Ogni programma C si articola per funzioni: esso è, in altre parole, un insieme di funzioni. Tuttavia, nonostante l'importanza che le funzioni hanno all'interno di un qualunque programma C, l'unica regola relativa al loro numero e al loro nome è che deve essere presente almeno una funzione ed almeno una delle funzioni deve chiamarsi `main()` (vedere pag. 8). L'esecuzione del programma inizia proprio con la prima istruzione contenuta nella funzione `main()`; questa può chiamare altre funzioni, che a loro volta ne possono chiamare altre ancora. L'unico limite è rappresentato dalla quantità di memoria disponibile.

Tutte le funzioni sono reciprocamente indipendenti e si collocano al medesimo livello gerarchico, nel senso che non vi sono funzioni più importanti di altre o dotate, in qualche modo, di diritti di precedenza: la sola eccezione a questa regola è rappresentata proprio da `main()`, in quanto essa deve obbligatoriamente esistere ed è sempre chiamata per prima.

Quando una funzione ne chiama un'altra, il controllo dell'esecuzione passa a quest'ultima che, al termine del proprio codice, o in corrispondenza dell'istruzione `return` lo restituisce alla chiamante. Ogni funzione può chiamare anche se stessa, secondo una tecnica detta ricorsione: approfondiremo a dovere l'argomento a pag. 100.

In generale, è utile suddividere l'algoritmo in parti bene definite, e codificare ciascuna di esse mediante una funzione dedicata; ciò può rivelarsi particolarmente opportuno soprattutto per quelle parti di elaborazione che devono essere ripetute più volte, magari su dati differenti. La ripetitività non è però l'unico criterio che conduce ad individuare porzioni di codice atte ad essere racchiuse in funzioni: l'importante, come si è accennato, è isolare compiti logicamente indipendenti dal resto del programma; è infatti usuale, in C, definire funzioni che nel corso dell'esecuzione vegono chiamate una volta sola.

Vediamo più da vicino una chiamata a funzione:

```
#include <stdio.h>

void main(void);

void main(void)
{
    printf("Esempio di chiamata.\n");
}
```

Nel programma di esempio abbiamo una chiamata alla funzione di libreria `printf()`[72]. Ogni compilatore C è accompagnato da uno o più file, detti librerie, contenenti funzioni già compilate e pronte all'uso, che è possibile chiamare dall'interno dei programmi: `printf()` è una di queste. In un programma è comunque possibile definire, cioè scrivere, un numero illimitato di funzioni, che potranno

[72] E dov'è la chiamata a `main()`? Non c'è proprio! E' il compilatore che provvede a generare il codice eseguibile necessario a chiamarla automaticamente alla partenza del programma. Ciò non vieta, tuttavia, di chiamare `main()`, se necessario, dall'interno di qualche altra funzione o persino dall'interno di se stessa.

essere chiamate da funzioni dello stesso programma[73]. L'elemento che caratterizza una chiamata a funzione è la presenza delle parentesi tonde aperta e chiusa alla destra del suo nome. Per il compilatore C, un nome seguito da una coppia di parentesi tonde è sempre una chiamata a funzione. Tra le parentesi vengono indicati i dati su cui la funzione lavora: è evidente che se la funzione chiamata non necessita ricevere dati dalla chiamante, tra le parentesi non viene specificato alcun parametro:

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

void main(void);

void main(void)
{
    char ch;

    printf("Premere un tasto:\n");

    ch = getch();
    printf("E' stato premuto %c\n",ch);
}
```

Nell'esempio è utilizzata la funzione `getch()`, che sospende l'esecuzione del programma ed attende la pressione di un tasto: come si vede essa è chiamata senza specificare alcun parametro tra le parentesi.

Inoltre `getch()` restituisce il codice ASCII del tasto premuto alla funzione chiamante: tale valore è memorizzato in `ch` mediante una normale operazione di assegnamento. In generale, una funzione può restituire un valore alla chiamante; in tal caso la chiamata a funzione è trattata come una qualsiasi espressione che restituisca un valore di un certo tipo: nell'esempio appena visto, infatti, la chiamata a `getch()` potrebbe essere passata direttamente a `printf()` come parametro.

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

void main(void);

void main(void)
{
    printf("Premere un tasto:\n");
    printf("E' stato premuto %c\n",getch());
}
```

Dal momento che in C la valutazione di espressioni nidificate avviene sempre dall'interno verso l'esterno, in questo caso dapprima è chiamata `getch()` e il valore da essa restituito è poi passato a `printf()`, che viene perciò chiamata solo al ritorno da `getch()`.

Dal punto di vista elaborativo la chiamata ad una funzione è il trasferimento dell'esecuzione al blocco di codice che la costituisce. Della funzione chiamante, la funzione chiamata conosce esclusivamente i parametri che quella le passa; a sua volta, la funzione chiamante conosce, della funzione chiamata, esclusivamente il tipo di parametri che essa si aspetta e riceve, se previsto, un valore (uno ed uno solo) di ritorno. Tale valore può essere considerato il risultato di un'espressione e come tale, lo si è visto, passato ad un altra funzione o memorizzato in una variabile, ma può anche essere ignorato:

[73] In realtà è possibile definire anche funzioni che non verranno utilizzate da quel programma. Il caso tipico è quello dei programmi TSR (Terminate and Stay Resident), il cui scopo è caricare in memoria e rendere residente un insieme di routine che verranno chiamate da altri programmi o da eventi di sistema. Di TSR si parla e straparla a pag. 275 e seguenti.

`printf()` restituisce il numero di caratteri visualizzati, ma negli esempi precedenti tale valore è stato ignorato (semplicemente non utilizzandolo in alcun modo) poiché non risultava utile nell'elaborazione effettuata.

Sotto l'aspetto formale, dunque, è lecito attendersi che ogni funzione richieda un certo numero di parametri, di tipo noto, e restituisca o no un valore, anch'esso di tipo conosciuto a priori. In effetti le cose stanno proprio così: numero e tipo di *parametri* e tipo del *valore di ritorno* sono stabiliti nella *definizione* della funzione.

DEFINIZIONE, PARAMETRI E VALORI RESTITUITI

La definizione di una funzione coincide, in pratica, con il codice che la costituisce. Ogni funzione, per poter essere utilizzata, deve essere definita: in termini un po' brutali potremmo dire che essa deve esistere, nello stesso sorgente in cui è chiamata oppure altrove (ad esempio in un altro sorgente o in una libreria, sotto forma di codice oggetto). Quando il compilatore incontra una chiamata a funzione non ha infatti alcuna necessità di conoscerne il corpo elaborativo: tutto ciò che gli serve sapere sono le regole di interfacciamento tra funzione chiamata e funzione chiamante, per essere in grado di verificare la correttezza formale della chiamata. Dette "regole" altro non sono che tipo e numero dei parametri richiesti dalla funzione chiamata e il tipo del valore restituito. Essi devono perciò essere specificati con precisione nella dichiarazione di ogni funzione. Vediamo:

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

int conferma(char *domanda, char si, char no)
{
    char risposta;

    do {
        printf("%s?",domanda);
        risposta = getch();
    while(risposta != si && risposta != no);
    if(risposta == si)
        return(1);
    return(0);
}
```

Quella dell'esempio è una normale definizione di funzione. Riprendiamo i concetti già accennati a pagina 8 con maggiore dettaglio: la definizione si apre con la dichiarazione del tipo di dato restituito dalla funzione. Se la funzione non restituisce nulla, il tipo specificato deve essere `void`.

Immediatamente dopo è specificato il nome della funzione: ogni chiamata deve rispettare scrupolosamente il modo in cui il nome è scritto qui, anche per quanto riguarda l'eventuale presenza di caratteri maiuscoli. La lunghezza massima del nome di una funzione varia da compilatore a compilatore; in genere è almeno pari a 32 caratteri. Il nome deve iniziare con un carattere alfabetico o con un *underscore* ("_") e può contenere caratteri, underscore e numeri (insomma, le regole sono analoghe a quelle già discusse circa i nomi delle variabili: vedere pag. 15).

Il nome è seguito dalle parentesi tonde aperta e chiusa, tra le quali devono essere elencati i parametri che la funzione riceve dalla chiamante. Per ogni parametro deve essere indicato il tipo ed il nome con cui è referenziato all'interno della funzione: se i parametri sono più di uno occorre separarli con virgole; se la funzione non riceve alcun parametro, tra le parentesi deve essere scritta la parola chiave `void`. Questo è l'elenco dei cosiddetti *parametri formali*; le variabili, costanti o espressioni passate alla funzione nelle chiamate sono invece indicate come *parametri attuali*[74].

[74] L'aggettivo *attuali* è di uso comune, ma deriva da una pessima traduzione dell'inglese *actual*, che significa *vero*, *reale*.

Si noti che dopo la parentesi tonda chiusa non vi è alcun punto e virgola (";"): essa è seguita (nella riga sottostante per maggiore leggibilità) da una graffa aperta, la quale indica il punto di partenza del codice eseguibile che compone la funzione stessa. Questo è concluso dalla graffa chiusa, ed è solitamente indicato come *corpo della funzione*.

Il corpo di una funzione è una normale sequenza di dichiarazioni di variabili, di istruzioni, di chiamate a funzione: l'unica cosa che esso non può contenere è un'altra definizione di funzione: proprio perché tutte le funzioni hanno pari livello gerarchico, non possono essere nidifcate, cioè definite l'una all'interno di un'altra.

L'esecuzione della funzione termina quando è incontrata l'ultima istruzione presente nel corpo oppure l'istruzione `return`: in entrambi i casi l'esecuzione ritorna alla funzione chiamante. Occorre però soffermarsi brevemente sull'istruzione `return`.

Se la funzione non è dichiarata `void` è obbligatorio utilizzare la `return` per uscire dalla funzione (anche quando ciò avvenga al termine del corpo), in quanto essa rappresenta l'unico strumento che consente di restituire un valore alla funzione chiamante. Detto valore deve essere indicato, opzionalmente tra parentesi tonde, alla destra della `return` e può essere una costante, una variabile o, in generale, un'espressione (anche una chiamata a funzione). E' ovvio che il tipo del valore specificato deve essere il medesimo restituito dalla funzione.

Se invece la funzione è dichiarata `void`, e quindi non restituisce alcun valore, l'uso dell'istruzione `return` è necessario solo se l'uscita deve avvenire (ad esempio in dipendenza dal verificarsi di certe condizioni) prima della fine del corpo (tuttavia non è vietato che l'utima istruzione della funzione sia proprio una `return`). A destra della `return` non deve essere specificato alcun valore, bensì direttamente il solito punto e virgola.

Perché una funzione possa essere chiamata, il compilatore deve conoscerne, come si è accennato, le regole di chiamata (parametri e valore restituito): è necessario, perciò, che essa sia definita prima della riga di codice che la richiama. In alternativa, può essere inserito nel sorgente il solo *prototipo* della funzione stessa: con tale termine si indica la prima riga della definizione, chiusa però dal punto e virgola. Nel caso dell'esempio, il prototipo di `conferma()` è il seguente:

```
int conferma(char *domanda, char si, char no);
```

Si vede facilmente che esso è sufficiente al compilatore per verificare che le chiamate a `conferma()` siano eseguite correttamente[75].

I prototipi sono inoltre l'unico strumento disponibile per consentire al compilatore di "fare conoscenza" con le funzioni di libreria richiamate nei sorgenti: infatti, essendo disponibili sotto forma di codice oggetto precompilato, esse non vengono mai definite. Le due direttive `#include` (pag. 8) in testa al codice dell'esempio presentato, che determinano l'inclusione nel sorgente dei file `STDIO.H` e `CONIO.H`, hanno proprio la finalità di rendere disponibili al compilatore i prototipi delle funzioni di libreria `printf()` e `getch()`.

E' forse più difficile elencare ed enunciare in modo chiaro e completo tutte le regole relative alla definizione delle funzioni e alla dichiarazione dei prototipi, di quanto lo sia seguirle nella pratica reale di programmazione. Innanzitutto non bisogna dimenticare che definire una funzione significa "scriverla" e che scrivere funzioni significa, a sua volta, scrivere un programma C: l'abitudine alle regole descritte si acquisisce in poco tempo. Inoltre, come al solito, il compilatore è piuttosto elastico e non si cura più di tanto di certi particolari: ad esempio, se una funzione restituisce un `int`, la dichiarazione del tipo restituito può essere omessa. Ancora: l'elenco dei parametri formali può ridursi all'elenco dei soli tipi, a patto di dichiarare i parametri stessi prima della graffa aperta, quasi come se fossero variabili qualunque. Infine, molti compilatori si fidano ciecamente del programmatore e non si turbano affatto se incontrano

[75] Vi sono però anche ragioni tecniche, e non solo formali, che rendono opportuni tali controlli da parte del compilatore: esse sono legate soprattutto alla gestione dello stack, l'area di memoria attraverso la quale i parametri sono resi disponibili alla funzione.

una chiamata ad una funzione del tutto sconosciuta, cioè non (ancora) definita né prototipizzata. Le regole descritte, però, sono quelle che meglio garantiscono una buona leggibilità del codice ed il massimo livello di controllo sintattico in fase di compilazione. Esse sono, tra l'altro, quasi tutte obbligatorie nella programmazione in C++, linguaggio che, pur derivando in maniera immediata dal C, è caratterizzato dallo *strong type checking*, cioè da regole di rigoroso controllo sulla coerenza dei tipi di dato.

Abbiamo detto che le funzioni di un programma sono tutte indipendenti tra loro e che ogni funzione non conosce ciò che accade nelle altre. In effetti le sole caratteristiche di una funzione note al resto del programma sono proprio i parametri richiesti ed il valore restituito; essi sono, altresì, l'unico modo possibile per uno scambio di dati tra funzioni.

E' però estremamente importante ricordare che una funzione non può mai modificare i parametri attuali che le sono passati, in quanto ciò che essa riceve è in realtà una copia dei medesimi. In altre parole, il passaggio dei parametri alle funzioni avviene *per valore* e non *per riferimento*. Il nome di una variabile identifica un'area di memoria: ebbene, quando si passa ad una funzione una variabile, non viene passato il riferimento a questa, cioè il suo indirizzo, bensì il suo valore, cioè una copia della variabile stessa. La funzione chiamata, perciò, non accede all'area di memoria associata alla variabile, ma a quella associata alla copia: essa può dunque modificare a piacere i parametri ricevuti senza il rischio di mescolare le carte in tavola alla funzione chiamante. Le copie dei parametri attuali sono, inoltre, locali alla funzione medesima e si comportano pertanto come qualsiasi variabile automatica (pag. 34).

L'impossibilità, per ciascuna funzione, di accedere a dati non locali ne accentua l'indipendenza da ogni altra parte del programma. Una eccezione è rappresentata dalle variabili globali (pag. 39), visibili per tutta la durata del programma e accessibili in qualsiasi funzione.

Vi è poi una seconda eccezione: i puntatori. A dire il vero essi sono un'eccezione solo in apparenza, ma di fatto consentono comportamenti contrari alla regola, appena enunciata, di inaccessibilità a dati non locali. Quando un puntatore è parametro formale di una funzione, il parametro attuale corrispondente rappresenta l'indirizzo di un'area di memoria: coerentemente con quanto affermato, alla funzione chiamata è passata una copia del puntatore, salvaguardando il parametro attuale, ma tramite l'indirizzo contenuto nel puntatore la funzione può accedere all'area di memoria "originale", in quanto, è bene sottolinearlo, solo il puntatore viene duplicato, e non l'area di RAM referenziata. E' proprio tramite questa apparente incongruenza che le funzioni possono modificare le stringhe di cui ricevano, quale parametro, l'indirizzo (o meglio, il puntatore).

```
#include <stdio.h>

#define  MAX_STR  20              // max. lung. della stringa incluso il NULL finale

void main(void);
char *setstring(char *string,char ch,int n);

void main(void)
{
    char string[MAX_STR];

    printf("[%s]\n",setstring(string,'X',MAX_STR));
}

char *setstring(char *string,char ch,int n)
{
    string[--n] = NULL;
    while(n)
        string[--n] = ch;
    return(string);
}
```

Nel programma di esempio è definita la funzione `setstring()`, che richiede tre parametri formali: nell'ordine, un puntatore a carattere, un carattere ed un intero. La prima istruzione di `setstring()` decrementa l'intero e poi lo utilizza come offset rispetto all'indirizzo contenuto nel

puntatore per inserire un NULL in quella posizione. Il ciclo while percorre a ritroso lo spazio assegnato al puntatore copiando, ad ogni iterazione, ch in un byte dopo avere decrementato n. Quando n è zero, tutto lo spazio allocato al puntatore è stato percorso e la funzione termina restituendo il medesimo indirizzo ricevuto come parametro. Ciò consente a main() di passarla come parametro a printf(), che visualizza, tra parentesi quadre, la stringa inizializzata da setstring(). Si nota facilmente che questa ha modificato il contenuto dell'area di memoria allocata in main().

Un'altra caratteristica interessante della gestione dei parametri attuali in C è il fatto che essi sono passati alla funzione chiamata a partire dall'ultimo, cioè da destra a sinistra. Tale comportamento, nella maggior parte delle situazioni, è trasparente per il programmatore, ma possono verificarsi casi in cui è facile essere tratti in inganno:

```
#include <stdio.h>

void main(void);
long square(void);

long number = 8;

void main(void)
{
    extern long number;

    printf("%ld squared = %ld\n",number,square());
}

long square(void)
{
    number *= number;
    return(number);
}
```

Il codice riportato non è certo un esempio di buona programmazione, ma evidenzia con efficacia che printf() riceve i parametri in ordine inverso a quello in cui sono elencati nella chiamata. Eseguendo il programma, infatti l'output ottenuto è

```
64 squared = 64
```

laddove ci si aspetterebbe un 8 al posto del primo 64, ma se si tiene conto della modalità di passaggio dei parametri, i conti tornano (beh... almeno dal punto di vista tecnico!). Il primo parametro che printf() riceve è il valore restituito da square(). Questa agisce direttamente sulla variabile globale number, sostituendone il valore con il risultato dell'elevamento al quadrato, e la restituisce. Successivamente printf() riceve la copia della stessa variabile, che però è già stata modificata da square(). L'esempio evidenzia, tra l'altro, la pericolosità intrinseca nelle variabili definite a livello globale. Vediamo ora un altro caso, più realistico.

```
#include <stdio.h>
#include <io.h>
#include <errno.h>

....
    int h1, h2;
    ....
    printf("dup2() restituisce %d; errore DOS %d\n",dup2(h1,h2),errno);
```

La funzione dup2(), il cui prototitpo è in IO.H, effettua un'operazione di redirezione di file (non interessa, ai fini dell'esempio, entrare in dettaglio) e restituisce 0 in caso di successo, oppure -1 qualora si verifichi un errore. Il codice di errore restituito dal sistema operativo è disponibile nella

variabile globale errno, dichiarata in ERRNO.H[76]. Lo scopo della printf() è, evidentemente, quello di visualizzare il valore restituito da dup2() e il codice di errore DOS corrispondente allo stato dell'operazione, ma il risultato ottenuto è invece che, accanto al valore di ritorno di dup2() sia visualizzato il valore che errno conteneva prima della chiamata alla dup2() stessa: infatti, essendo i parametri passati a printf() a partire dall'ultimo, la copia di errno è generata prima che si realizzi effettivamente la chiamata a dup2().

Questa strana tecnica di passaggio "a ritroso" dei parametri ha uno scopo estremamente importante: consentire la definizione di funzioni in grado di accettare un numero variabile di parametri.

Abbiamo sottomano un esempio pratico: la funzione di libreria printf(). Ai più attenti non dovrebbe essere sfuggito che, negli esempi sin qui presentati, essa riceve talvolta un solo parametro (la stringa di formato), mentre in altri casi le sono passati, oltre a detta stringa (sempre presente), altri parametri (i dati da visualizzare) di differente tipo.

Il carattere introduttivo di queste note rende inutile un approfondimento eccessivo dell'argomento[77]: è però interessante sottolineare che, in generale, quando una funzione accetta un numero variabile di parametri, è dichiarata con uno o più parametri formali "fissi" (i primi della lista), almeno uno dei quali contiene le informazioni che servono alla funzione per stabilire quanti parametri attuali le siano effettivamente passati ed a quale tipo appartengano. Nel caso di printf() il parametro fisso è la stringa di formato (o meglio, il puntatore alla stringa); questa contiene, se nella chiamata sono passati altri parametri, un indicatore di formato per ogni parametro addizionale (i vari "%d", "%s", e così via). Analizzando la stringa, printf() può scoprire quanti altri parametri ha ricevuto dalla funzione chiamante, e il loro tipo.

D'accordo, ma per fare questo era proprio necessario implementare il passaggio a ritroso dei parametri? La risposta è sì, ma per capirlo occorre scendere un poco in dettagli di carattere tecnico. Il passaggio dei parametri avviene attraverso lo *stack*, un'area di memoria gestita in base al principio LIFO (Last In, First Out; cioè: l'ultimo che entra è il primo ad uscire): ciò significa che l'ultimo dato scritto nello stack è sempre il primo ad esserne estratto. Tornando alla nostra printf(), a questo punto è chiaro che preparandone una chiamata, il compilatore copia nello stack in ultima posizione proprio il puntatore alla stringa di formato, ma questo è anche il primo dato a cui il codice di printf() può accedere. In altre parole, la funzione conosce con certezza la posizione nello stack del primo parametro attuale, in quanto esso vi è stato copiato per ultimo: analizzandolo può sapere quanti altri, in sequenza, ne deve estrarre dallo stack.

Ecco il prototipo standard di printf():

```
int printf(const char *format, ...);
```

Come si vede, è utilizzata l'ellissi ("...", tre punti) per indicare che da quel parametro in poi il numero ed il tipo dei parametri formali non è noto a priori. In questi casi, il compilatore, nell'analizzare la congruenza tra parametri formali ed attuali nelle chiamate, è costretto ad accettare quel che "passa" il convento (...è il caso di dirlo).

In C è comunque possibile definire funzioni per le quali il passaggio dei parametri è effettuato "in avanti", cioè dal primo all'ultimo, nel medesimo ordine della dichiarazione: è sufficiente anteporre al nome della funzione la parola chiave pascal[78].

---

[76] Molte funzioni C utilizzano questo sistema per gestire situazioni di errore in operazioni basate su chiamate al DOS.

[77] Esistono funzioni di libreria specializzate nella gestione di parametri in numero variabile, che possono essere richiamate dalle funzioni definite dal programmatore per conoscere quanti parametri attuali sono stati passati, e così via. Si tratta del gruppo di funzioni va_start(), va_arg(), va_end().

[78] La parola chiave è stata scelta per analogia con il linguaggio Pascal, in cui il passaggio dei parametri avviene sempre da sinistra a destra, cioè "in avanti".

```
char *pascal funz_1(char *s1,char *s2);        // funz. che restituisce un ptr a char
void pascal funz_2(int a);                                          // funzione void
int far pascal funz_3(void);                            // funz. far che restit. un int
char far * far pascal funz_4(char c,int a);       // funz. far che restit. un far ptr
```

L'esempio riporta alcuni prototipi di funzioni dichiarate `pascal`: l'analogia con i "normali" prototipi di funzioni è evidente, dal momento che l'unica differenza è proprio rappresentata dalla presenza della nuova parola chiave. Come si vede, anche le funzioni che non prendono parametri possono essere dichiarate `pascal`; tuttavia una funzione `pascal` non può mai essere dichiarata con un numero variabile di parametri. A questo limite si contrappone il vantaggio di una sequenza di istruzioni assembler di chiamata un po' più efficiente[79]. In pratica, tutte le funzioni con un numero fisso di parametri possono essere tranquillamente dichiarate `pascal`, sebbene ciò, è ovvio, non sia del tutto coerente con la filosofia del linguaggio C. Per un esempio notevole di funzione di libreria dichiarata `pascal` vedere pag. 499; si osservi inoltre che in ambiente Microsoft Windows quasi tutte le funzioni sono dichiarate `pascal`.

Per complicare le cose, aggiungiamo che molti compilatori accettano una opzione di command line per generare chiamate `pascal` come default (per il compilatore Borland essa è `-p`):

```
bcc -p pippo.c
```

Con il comando dell'esempio, tutte le funzioni dichiarate in `PIPPO.C` e nei file `.H` da esso inclusi sono chiamate in modalità `pascal`, eccetto `main()` (che è sempre chiamata in modalità C) e le funzioni dichiarate `cdecl`. Quest'ultima parola chiave ha scopo esattamente opposto a quello di pascal, imponendo che la funzione sia chiamata in modalità C (cioè col passaggio in ordine inverso dei parametri) anche se la compilazione avviene con l'opzione di modalità `pascal` per default.

```
char *cdecl funz_1(char *s1,char *s2);         // funz. che restituisce un ptr a char
void cdecl funz_2(int a);                                           // funzione void
int far cdecl funz_3(void);                             // funz. far che restit. un int
char far * far cdecl funz_4(char c,...);          // funz. far che restit. un far ptr
```

L'esempio riprende i prototipi esaminati poco fa, introducendo però una modifica all'ultimo di essi: la funzione `funz_4()` accetta un numero variabile di parametri. E' opportuno dichiarare esplicitamente `cdecl` tutte le funzioni con numero di parametri variabile, onde consentirne l'utilizzo anche in programmi compilati in modalità `pascal`.

PUNTATORI A FUNZIONI

Credevate di esservene liberati? Ebbene no! Rieccoci a parlare di puntatori... Sin qui li abbiamo presentati come variabili un po' particolari, che contengono l'indirizzo di un dato piuttosto che un dato vero e proprio. E' giunto il momento di rivedere tale concetto, di ampliarlo, in quanto possono essere dichiarati puntatori destinati a contenere l'indirizzo di una funzione.

Un puntatore a funzione è dunque un puntatore che non contiene l'indirizzo di un intero, o di un carattere, o di un qualsiasi altro tipo di dato, bensì l'indirizzo del primo byte del codice di una funzione. Vediamone la dichiarazione:

```
int (*funcPtr)(char *string);
```

---

[79] Per la chiamata di una funzione "normale", il compilatore genera delle istruzioni `PUSH` per i parametri da passare, una istruzione `CALL` e le istruzioni `POP` necessarie a rimuovere dallo stack i parametri passati. La funzione si chiude con una `RET`. Se la medesima funzione è dichiarata `pascal`, il compilatore genera ancora le `PUSH` e la `CALL`, ma non le `POP`, in quanto la funzione, essendo sempre fisso il numero di parametri, può provvedere da sé alla pulizia dello stack terminando con una `RET n`, dove `n` esprime il numero di byte da eliminare dallo stack.

Nell'esempio funcPtr è un puntatore ad una funzione che restituisce un int e accetta quale parametro un puntatore a char. La sintassi può apparire complessa, ma un esame più approfondito rivela la sostanziale analogia con i puntatori che già conosciamo. Innanzitutto, l'asterisco che precede il nome funcPtr ne rivela inequivocabilmente la natura di puntatore. Anche la parola chiave int ha un ruolo noto: indica che l'indirezione del puntatore restituisce un intero. Trattandosi di un puntatore a funzione, funcPtr è seguito dalle parentesi tonde contenenti la lista dei parametri della funzione. Sono proprio queste parentesi a indicare che funcPtr è puntatore a funzione. Restano da spiegare le parentesi che racchiudono *funcPtr: esse sono indispensabili per distinguere la dichiarazione di un puntatore a funzione da un prototipo di funzione. Se riscriviamo la dichiarazione dell'esempio omettendo la prima coppia di parentesi, otteniamo

```
int *funcPtr(char *string);
```

cioè il prototipo di una funzione che restituisce un puntatore ad intero e prende come parametro un puntatore a carattere.

Poco fa si è detto che l'indirezione di funcPtr restituisce un intero. Che significato ha l'indirezione di un puntatore a funzione? Quando si ha a che fare con puntatori a "dati", il concetto è piuttosto semplice: l'indirezione rappresenta il dato che si trova all'indirizzo contenuto nel puntatore stesso. Ma all'indirizzo contenuto in un puntatore a funzione si trova una parte del programma, cioè vero e proprio codice eseguibile: allora ha senso parlare di indirezione di un puntatore a funzione solo con riferimento al dato restituito dalla funzione che esso indirizza. Ma perché una funzione possa restituire qualcosa deve essere eseguita: e proprio qui sta il bello, dal momento che l'indirezione di un puntatore a funzione rappresenta una chiamata alla funzione indirizzata. Vediamo funcPtr all'opera:

```
#include <string.h>

...
    int iVar;
    char *cBuffer;
    ....
    funcPtr = strlen;
    ....
    iVar = (*funcPtr)(cBuffer);
    ....
```

Nell'esempio, a funcPtr è assegnato l'indirizzo della funzione di libreria strlen(), il cui prototipo si trova in STRING.H, che accetta quale parametro un puntatore a stringa e ne restituisce la lunghezza (sotto forma di intero). Se ne traggono alcune interessanti indicazioni: per assegnare ad un puntatore a funzione l'indirizzo di una funzione basta assegnargli il nome di quest'ultima. Si noti che il simbolo strlen non è seguito dalle parentesi, poiché in questo caso non intendiamo chiamare strlen() e assegnare a funcPtr il valore che essa restituisce, bensì assegnare a funcPtr l'indirizzo a cui strlen() si trova[80]. Inoltre, il tipo di dato restituito dalla funzione e la lista dei parametri devono corrispondere a quelli dichiarati col puntatore: tale condizione, in questo caso, è soddisfatta.

Infine, nell'esempio compare anche la famigerata indirezione del puntatore: come si vede, al parametro formale della dichiarazione è stato sostituito il parametro attuale (come in qualsiasi chiamata a funzione) e al posto dell'indicatore del tipo restituito troviamo, da destra a sinistra, l'operatore di assegnamento e la variabile che memorizza quel valore.

Va sottolineato che l'indirezione è perfettamente equivalente alla chiamata alla funzione indirizzata dal puntatore: in questo caso a

[80] Ma allora... il nome di una funzione è puntatore alla funzione stessa! Ricordate il caso degli array? Chi ha poca memoria può sbirciare a pagina 29.

```
iVar = strlen(cBuffer);
```

Allora perché complicarsi la vita con i puntatori? I motivi sono molteplici. A volte è indispensabile conoscere gli indirizzi di alcune routine per poterle gestire correttamente[81]. In altri casi l'utilizzo di puntatori a funzione consente di scrivere codice più efficiente: si consdieri l'esempio che segue.

```
if(a > b)
    for(i = 0; i < 1000; i++)
        funz_A(i);
else
    for(i = 0; i < 1000; i++)
        funz_B(i);
```

Il frammento di codice può essere razionalizzato mediante l'uso di un puntatore a funzione, evitando di scrivere due cicli `for` quasi identici:

```
void (*fptr)(int i);
....
if(a > b)
    fptr = funz_A;
else
    fptr = funz_B;
for(i = 0; i < 1000; i++)
    (*fptr)(i);
```

Più in generale, l'uso dei puntatori a funzione si rivela di grande utilità quando, nello sviluppare l'algoritmo, non si può determinare a priori quale funzione deve essere chiamata in una certa situazione, ma è possibile farlo solo al momento dell'esecuzione, dall'esame dei dati elaborati. Un esempio può essere costituito dalla cosiddetta programmazione per flussi guidati da tabelle, nella quale i dati in input consentono di individuare un elemento di una tabella contenente i puntatori alle funzioni richiamabili in quel contesto.

Per studiare nel concreto una applicazione del concetto appena espresso si può pensare ad una programma in grado di visualizzare un sorgente C eliminando tutti i commenti introdotti dalla doppia barra ("`//`", vedere pag. 14). In pratica si tratta di passare alla riga di codice successiva quando si incontra tale sequenza di caratteri: analizzando il testo carattere per carattere, bisogna visualizzare tutti i caratteri letti fino a che si incontra una barra. In questo caso, per decidere che cosa fare, occorre esaminare il carattere successivo: se è anch'esso una barra si passa alla riga successiva e si riprendono a visualizzare i caratteri; se non lo è, invece, deve essere visualizzato, ma preceduto da una barra, e l'elaborazione prosegue visualizzando i caratteri incontrati.

I possibili stati del flusso elaborativo, dunque, sono due: elaborazione normale, che prevede la visualizzazione del carattere, e attesa, indotto dall'individuazione di una barra. La situazione complessiva delle azioni da intraprendere può essere riassunta in una tabella, ogni casella della quale rappresenta le azioni da intraprendere quando si verifichi una data combinazione tra stato elaborativo attuale e carattere incontrato.

---

[81] E' il caso dei programmi che incorporano funzioni per la gestione degli interrupt di sistema. Essi devono memorizzare l'indirizzo degli interrupt ai quali sostituiscono le proprie routine, al fine di poterli utilizzare in caso di necessità e per riattivarli al termine della propria esecuzione. Non sembra però il caso di approfondire ulteriormente l'argomento, almeno per ora. Ma lo si farà a pag. 251 e seguenti.

AZIONI DA INTRAPRENDERE	**Carattere incontrato**	
Stato elaborativo	Barra "/"	Altro carattere
Elaborazione *normale*	Non visualizza il carattere Legge il carattere successivo Passa in stato "Attesa"	Visualizza il carattere Legge il carattere successivo Resta in stato "Normale"
Attesa carattere successivo	Non visualizza il carattere Legge la riga successiva Passa in stato "Normale"	Visualizza "/" e il carattere Legge il carattere successivo Passa in stato "Normale"

Circa il trattamento del carattere, le possibili situazioni sono tre: visualizzazione, non visualizzazione, e visualizzazione del carattere stesso preceduto da una barra. La scansione del file può proseguire in due modi diversi: carattere successivo o riga successiva. Infine, si può avere il passaggio dallo stato normale a quello di attesa, il viceversa, o il permanere nello stato normale. Si tratta di una situazione un po' intricata, ma facilmente trasformabile in algoritmo utilizzando proprio i puntatori a funzione.

Quello che ci occorre è, in primo luogo, un ciclo di controllo del flusso elaborativo: il guscio esterno del programma consiste nella lettura del file riga per riga e nell'analisi della riga letta carattere per carattere.

```
#include <stdio.h>

#define  MAXLIN    256

void main(void);

void main(void)
{
    char line[MAXLIN], *ptr;

    while(gets(line)) {
        for(ptr = line; *ptr; ) {
            ....
        }
        printf("\n");
    }
}
```

Ecco fatto. La funzione di libreria `gets()` legge una riga dallo standard input[82] e la memorizza nell'array di caratteri il cui indirizzo le è passato quale parametro. Dal momento che essa restituisce `NULL` se non vi è nulla da leggere, il ciclo `while()` è iterato sino alla lettura dell'ultima riga del file. Il ciclo `for()` scandisce la riga carattere per carattere e procede sino a quando è incontrato il `NULL` che chiude

[82] Lo standard input (`stdin`) è un "file" particolare, che il DOS identifica normalmente con la tastiera. Esso può però essere rediretto ad un file qualunque mediante il simbolo "<". Supponendo di chiamare il nostro programma `NOCOMENT`, è sufficiente lanciare il comando

```
nocoment < pippo.c
```

per visualizzarne il contenuto privo di commenti. Vedere pag. 116.

la riga. E' compito del codice all'interno del ciclo incrementare opportunamente `ptr`. All'uscita dal ciclo `for()` si va a capo[83].

A questo punto entrano trionfalmente in scena i puntatori a funzione. Per elaborare correttamente una singola riga ci occorrono quattro diverse funzioni, ciascuna in grado di manipolare un dato carattere come descritto in una delle quattro caselle della nostra tabella. Vediamole:

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

#define  NORMAL   0
#define  WAIT     1

char *hideLetterInc(char *ptr)          // non visualizza il carattere e restituisce
{                                       // il puntatore incrementato (tabella[0][0])
    extern int nextStatus;

    nextStatus = WAIT;
    return(ptr+1);
}

char *sayLetterInc(char *ptr)            // visualizza il carattere e restituisce il
{                                          // puntatore incrementato (tabella[0][1])
    extern int nextStatus;

    nextStatus = NORMAL;
    printf("%c",*ptr);
    return(ptr+1);
}

char *hideLetterNextLine(char *ptr)                // non visualizza il carattere e
{                                               // restituisce l'indirizzo del NULL
    extern int nextStatus;                            // terminator (tabella[1][0])

    nextStatus = NORMAL;
    return(ptr+(strlen(ptr));
}

char *sayBarLetterInc(char *ptr)         // visualizza il carattere preceduto da una
{                                               // barra e restituisce il puntatore
    extern int nextStatus;                          // incrementato (tabella[1][1])

    nextStatus = NORMAL;
    printf("/%c",*ptr);
    return(ptr+1);
}
```

Come si vede, il codice delle funzioni è estremamente semplice. Tuttavia, ciascuna esaurisce il compito descritto in una singola cella della tabella, compresa la "decisione" circa lo stato ("normale" o "attesa") che vale per il successivo carattere da esaminare: non ci resta che creare una tabella analoga a quella presentata poco fa, ma contenente i puntatori alle funzioni.

[83] La `gets()` sostituisce l'a capo ("\n")di fine riga letto dal file con un null terminator. Ciò rende necessario andare a capo esplicitamente.

FUNZIONI DA CHIAMARE	Carattere incontrato	
Stato elaborativo	Barra "/"	Altro carattere
Elaborazione *normale*	`hideLetterInc()`	`sayLetterInc()`
Attesa carattere successivo	`hideLetterNextLine()`	`sayBarLetterInc()`

Ed ecco la codifica C della tabella di puntatori a funzione:

```
char *hideLetterInc(char *ptr);
char *sayLetterInc(char *ptr);
char *hideLetterNextLine(char *ptr);
char *sayBarLetterInc(char *ptr);

char *(*funcs[2][2])(char *ptr) = {
    {hideLetterInc,      sayLetterInc},
    {hideLetterNextLine, sayBarLetterInc}
};
```

Lo stato di elaborazione è, per default, "Normale" e viene individuato dalle funzioni ad ogni carattere trattato, il quale è la seconda coordinata necessaria per individuare il puntatore a funzione opportuno all'interno della tabella. Ora siamo finalmente in grado di presentare il listato completo del programma.

```
#include <stdio.h>                                            // per printf() e gets()
#include <string.h>                                                    // per strlen()

#define  MAXLIN    256
#define  NORMAL      0
#define  WAIT        1
#define  BAR         0
#define  NON_BAR     1

void main(void);
char *hideLetterInc(char *ptr);
char *sayLetterInc(char *ptr);
char *hideLetterNextLine(char *ptr);
char *sayBarLetterInc(char *ptr);

extern int nextStatus = NORMAL;

void main(void)
{
    static char *(*funcs[2][2])(char *ptr) = {                  // e' static perche' e'
        {hideLetterInc, sayLetterInc},                    // dichiarato ed inizializzato
        {hideLetterNextLine, sayBarLetterInc}                         // in una funzione
    };
    char line[MAXLIN], *ptr;
    int letterType;

    while(gets(line)) {
        for(ptr = line; *ptr; ) {
            switch(*ptr) {
                case '/':
                    letterType = BAR;
                    break;
                default:
                    letterType = NON_BAR;
            }
```

```
            ptr = (*funcs[nextStatus][letterType])(ptr);
        }
        printf("\n");
    }
}

char *hideLetterInc(char *ptr)          // non visualizza il carattere e restituisce
{                                       // il puntatore incrementato (tabella[0][0])
    extern int nextStatus;

    nextStatus = WAIT;
    return(ptr+1);
}

char *sayLetterInc(char *ptr)            // visualizza il carattere e restituisce il
{                                          // puntatore incrementato (tabella[0][1])
    extern int nextStatus;

    nextStatus = NORMAL;
    printf("%c",*ptr);
    return(ptr+1);
}

char *hideLetterNextLine(char *ptr)                 // non visualizza il carattere e
{                                                // restituisce l'indirizzo del NULL
    extern int nextStatus;                                 // terminator (tabella[1][0])

    nextStatus = NORMAL;
    return(ptr+(strlen(ptr)));
}

char *sayBarLetterInc(char *ptr)         // visualizza il carattere preceduto da una
{                                                // barra e restituisce il puntatore
    extern int nextStatus;                               // incrementato (tabella[1][1])

    nextStatus = NORMAL;
    printf("/%c",*ptr);
    return(ptr+1);
}
```

Il contenuto del ciclo `for()` è sorprendentemente semplice[84]. Ma il cuore di tutto il programma è la riga

```
    ptr = (*funcs[nextStatus][letterType])(ptr);
```

in cui possiamo ammirare il risultato di tutti i nostri sforzi elucubrativi: una sola chiamata a funzione, realizzata attraverso un puntatore, a sua volta individuato nella tabella tramite le "coordinate" `nextStatus` e `letterType`, evita una serie di `if` nidificate e, di conseguenza, una codifica dell'algoritmo sicuramente meno essenziale ed efficiente.

L'esempio evidenzia inoltre quale sia la sintassi della dichiarazione e dell'utilizzo di un array di puntatori a funzione.

Forse può apparire non del tutto chiaro come sia forzata la lettura della riga successiva quando è individuato un commento: il test del ciclo `for()` determina l'uscita dal medesimo quando l'indirezione di

[84] E avrebbe potuto esserlo ancora di più, utilizzando una `if....else` in luogo della `switch`. Quest'ultima, però, rende più agevoli eventuali modifiche future al codice, qualora, ed esempio, si dovessero implementare elaborazioni diverse in corrispondenza di altri caratteri.

ptr è un byte nullo, e questa è proprio la situazione indotta dalla funzione hideLetterNextLine(), che restituisce un puntatore al null terminator della stringa contenuta in line.

Va ancora sottolineato che nextStatus è dichiarata come variabile globale per... pigrizia: dichiararla all'interno di main() avrebbe reso necessario passarne l'indirizzo alle funzioni richiamate mediante il puntatore, perché queste possano modificarne il valore. Nulla di difficile, ma non era il caso di complicare troppo l'esempio.

Infine, è meglio non montarsi la testa: quello presentato è un programma tutt'altro che privo di limiti. Infatti non è in grado di riconoscere una coppia di barre inserita all'interno di una stringa, e la considera erroneamente l'inizio di un commento; inoltre visualizza comunque tutti gli spazi compresi tra l'ultimo carattere valido di una riga e l'inizio del commento. L'ingrato compito di modificare il sorgente tenendo conto di queste ulteriori finezze è lasciato, come nei migliori testi, alla buona volontà del lettore[85].

Tanto per complicare un po' le cose, anche i puntatori a funzione possono essere near o far. Per chiarire che cosa ciò significhi, occorre ancora una volta addentrarsi in alcuni dettagli tecnici. I processori Intel seguono il flusso elaborativo, istruzione per istruzione, mediante due registri, detti CS e IP (*Code Segment* e *Instruction Pointer*): i due nomi ne svelano di per sé le rispettive funzioni. Il primo fissa un'origine ad un certo indirizzo, mentre il secondo esprime l'offset, a partire da quell'indirizzo, della prossima istruzione da eseguire. Se il primo byte di una funzione dista dall'origine meno di 65535 byte è sufficiente, per indirizzarla, un puntatore near, cioè a 16 bit, associato ad IP. In programmi molto grandi è normale che una funzione si trovi in un segmento di memoria diverso da quello corrente[86]: il suo indirizzo deve perciò essere espresso con un valore a 32 bit (un puntatore far, la cui word più significativa è associata a CS e quella meno significativa ad IP).

Bisogna sottolineare che le funzioni stesse possono essere dichiarate near o far. Naturalmente, dichiarare far una funzione non significa forzare il compilatore a creare un programma enorme per poterla posizionare "lontano": esso genera semplicemente un differente algoritmo di chiamata. Tutte le funzioni far sono chiamate salvando sullo stack sia CS che IP (l'indirizzo di rientro dalla funzione), indipendentemente dal fatto che il contenuto di CS debba essere effettivamente modificato. Nelle chiamate di tipo near, invece, viene salvato (e modificato) solo IP. In uscita dalla funzione i valori di CS ed IP sono estratti dallo stack e ripristinati, così da poter riprendere l'esecuzione dall'istruzione successiva alla chiamata a funzione. E' evidente che una chiamata di tipo far può eseguire qualunque funzione, ovunque essa si trovi, mentre una chiamata near può eseguire solo quelle che si trovano effettivamente all'interno del segmento definito da CS. Spesso si dichiara far una funzione proprio per renderla indipendente dalle dimensioni del programma, o meglio dal modello di memoria scelto per compilare il programma. L'argomento è sviluppato a pagina 146 con particolare riferimento alle chiamate intersegmento; per ora è sufficiente precisare che proprio dal modello di memoria dipende il tipo di chiamata che il compilatore genera per una funzione non dichiarata near o far in modo esplicito. In altre parole, una definizione come

```
int funzione(char *buf)
{
    ....
}
```

---

[85] Un suggerimento circa le stringhe? Eccolo: il carattere *virgolette* ("), gestito con una terza colonna nella tabella, potrebbe determinare un terzo stato elaborativo (la terza riga della tabella), che determina la normale visualizzazione di tutto ciò che si incontra e l'avanzamento del puntatore di una posizione soltanto. Anche le barre vengono visualizzate come se nulla fosse. Quando si incontra nuovamente il carattere *virgolette* si ritorna nello stato "Normale".

[86] Cioè oltre il fatidico limite dei 64 Kb che iniziano all'indirizzo contenuto in CS. La logica è del tutto analoga a quella descritta circa i puntatori "a dati" (pag. 16).

origina una funzione `near` o `far` a seconda del modello di memoria scelto. Analoghe considerazioni valgono per i puntatori: è ancora una volta il modello di memoria a stabilire se un puntatore dichiarato come segue

```
int (*fptr)(char *buf);
```

è `near` o `far`. Qualora si intenda dichiarare esplicitamente una funzione o un puntatore `far`, la sintassi è ovvia:

```
int far funzione(char *buf)
{
    ....
}
```

per la funzione, e

```
int (far *fptr)(char *buf);
```

per il puntatore. Dichiarazioni `near` esplicite sono assolutamente analoghe a quelle appena presentate.

Infine, le funzioni possono essere definite `static`:

```
static int funzione(char *buf)
{
    ....
}
```

per renderle accessibili (cioè "richiamabili") solo all'interno del sorgente in cui sono definite. Non è però possibile, nella dichiarazione di un puntatore a funzione, indicare che questa è `static`: la riga

```
static int (*fptr)(char *buf);
```

dichiara un puntatore `static` a funzione. Ciò appare comprensibile se si considera che, riferita ad una funzione, la parola chiave `static` ne modifica unicamente la visibilità, e non il tipo di dato restituito (vedere anche pag. 25 e pag. 41).

LA RICORSIONE

Abbiamo già accennato (pag. 85) che la ricorsione è realizzata da una funzione che richiama se stessa: si tratta di una tecnica di programmazione che può fornire soluzioni eleganti ed efficienti a problemi che, talvolta, possono essere affrontati anche mediante la semplice iterazione. Ogni funzione, `main()` compresa, può richiamare se stessa, ma è evidente che deve essere strutturata in maniera opportuna: non esistono, peraltro, strumenti appositi; occorre progettare attentamente l'algortimo.

Un esempio di problema risolvibile sia iterativamente che ricorsivamente è il calcolo del fattoriale di un numero. Il fattoriale di un numero intero positivo `n` (simbolo `n!`) è espresso come una serie di moltiplicazioni ripetute a partire da

```
n * (n - 1)
```

Il risultato di ogni moltiplicazione è quindi moltiplicato per un fattore di una unità inferiore rispetto a quello del moltiplicatore dell'operazione precedente. La formula di calcolo del fattoriale di `n` è pertanto:

```
n! = n * (n - 1) * (n - 2) * ... * 2 * 1
```

Inoltre, per definizione, `1! = 1` e `0! = 1`. Raggruppando tutti i fattori che, nella formula precedente, precedono `n`, si osserva che

```
(n - 1)! = (n - 1) * (n - 2) * ... * 2 * 1
```

e pertanto il fattoriale di un numero intero può essere anche espresso come il prodotto del medesimo per il fattoriale dell'intero che lo precede:

```
n! = n * (n - 1)!
```

I due esempi che seguono implementano il calcolo del fattoriale con due approcci radicalmente differenti: delle due definizioni, o meglio "rappresentazioni" di `n!` date poco sopra, la soluzione iterativa è fondata sulla prima, mentre la ricorsione traduce in concreto la seconda.

Un ciclo in grado di calcolare il fattoriale di un intero è il seguente:

```
    int n;
    long nfatt;
    ....
    for(nfatt = 1L; n > 1; n--)
        nfatt *= n;
```

Al termine delle iterazioni `nfatt` vale `n!`.

Vediamo ora la soluzione ricorsiva:

```
long fattoriale(long n)
{
    return((n < 2) ? 1 : n * fattoriale(n - 1));
}
```

La funzione `fattoriale()` restituisce `1` se il parametro ricevuto è minore di `2` (cioè vale `0` o `1`), mentre in caso contrario il valore restituito è il prodotto di `n` per il fattoriale di `n-1`, cioè `n!`: si noti che `fattoriale()` calcola il valore da restituire chiamando se stessa e "passandosi" quale parametro il parametro appena ricevuto, ma diminuito di uno.

Il termine "passandosi" è una semplificazione: in realtà `fattoriale()` non passa il parametro a se stessa, ma ad una ulteriore istanza di sé. Che significa? Nell'esecuzione del programma ogni chiamata a `fattoriale()` utilizza in memoria, per i dati[87], una differente area di lavoro, in quanto anche questo meccanismo utilizza lo stack per operare. Se una funzione definisce variabili locali ed effettua una ricorsione, la nuova istanza alloca le proprie variabili locali, senza conoscere l'esistenza di quelle dell'istanza ricorrente. E' evidente che se una istanza di una ricorsione potesse accedere a tutte le variabili, incluse quelle locali, definite in ogni altra istanza, la funzione non avrebbe un proprio spazio "riservato" in cui operare: ogni modifica a quasiasi variabile si rifletterebbe in tutte le istanze, e ciascuna di esse potrebbe quindi scompaginare il valore delle altre.

A volte, però, può essere utile che una istanza conosca qualcosa delle altre: ad esempio un contatore, che consenta di sapere in qualunque istanza quanto in profondità si sia spinta la ricorsione. Tale esigenza è soddisfatta dalle variabili `static`, in quanto esse sono locali alla funzione in cui sono definite, ma comuni a tutte le sue istanze. L'affermazione risulta del tutto comprensibile se si tiene conto che le variabili statiche sono accessibili solo alla funzione in cui sono definite, ma esistono e conservano il loro valore per tutta la durata del programma. Quando una funzione è chiamata per la prima volta ed assegna un valore ad una variabile statica, questa mantiene il proprio valore anche in una seconda istanza (e in tutte le successive) della stessa funzione; mentre delle variabili `automatic` è generata una nuova copia in ogni istanza, una variabile `static` è unica in tutte le istanze e poiché essa esiste e mantiene il

---

[87] E' ovvio che il codice eseguito è fisicamente il medesimo.

proprio valore anche in uscita dalla funzione, ogni istanza può conoscere non solo il valore che tale variabile aveva nell'istanza precedente, ma anche nell'istanza successiva (ovviamente dopo il termine di questa).

Le variabili globali, infine, sono accessibili a tutte le istanze ma, a differenza di quelle statiche, lo sono anche alle altre funzioni: in fondo non si tratta di una novità.

Si noti che la funzione `fattoriale()` deve essere chiamata una sola volta per ottenere il risultato ricercato:

```
    printf("10! = %ld\n",fattoriale(10));
```

Non serve alcuna iterazione, perché la ricorsione implementata internamente dalla funzione è sufficiente al calcolo del risultato.

Vediamo un altro esempio: la funzione `scanDirectory()` ricerca un file nell'albero delle directory, percorrendo tutte le sottodirectory di quella specificata come punto di partenza:

```
/********************************************************

    SCANDIR.C - Barninga Z! - 1994

    void cdecl scanDirectory(char *path,char *file);

    char *path;   path di partenza per la ricerca del file. Deve terminare con una
                  backslash ("\").
    char *file;   nome del file da ricercare in path ed in tutte le sue subdir. Puo'
                  contenere le wildcards "?" e "*".

    Visualizza i pathnames (a partire dal punto indicato da path) dei files trovati.

    Compilato con Borland C++ 3.1:

    bcc -c -mx scandir.c

    dove x specifica il modello di memoria e puo' essere: t, s, m, c, l, h.

********************************************************/
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <dir.h>

#define ALL_ATTR   (FA_ARCH+FA_HIDDEN+FA_SYSTEM+FA_RDONLY)// cerca ogni tipo di file

void cdecl scanDirectory(char *path,char *file)
{
    struct ffblk ff;

// Considera tutto quello che c'e' nella directory (*.*)

    strcat(path,"*.*");

// Qui inizia il loop di scansione della directory in cerca di eventuali subdir.
// In pratica ricerca tutti i tipi di file compresi quelli il cui attributo indica
// che si tratta di una directory.

    if(!findfirst(path,&ff,FA_DIREC+ALL_ATTR)) {                  // cerca le subdir
        do {

// Se il file trovato e' proprio una directory, e non e' "." o ".." (cioe' la
// directory stessa o la directory di cui essa e' subdir) allora viene concatenato
// il nome della dir trovata al pathname di lavoro...
```

```
            if((ff.ff_attrib & FA_DIREC) && (*ff.ff_name != '.')) {
                strcpy(strrchr(path,'\\')+1,ff.ff_name);
                strcat(path,"\\");

// ...e si effettua la ricorsione: scanDirectory() richiama se stessa passando alla
// nuova istanza il nuovo path in cui ricercare le directory. Cio' si spiega col
// fatto che per ogni directory l'elaborazione da effettuare e' sempre la stessa;
// l'unica differenza e' che ci si addentra di un livello nell'albero gerarchico del
// disco.

                scanDirectory(path,file);
            }
        } while(!findnext(&ff));            // procede finche' trova files o subdirs
    }

// Quando tutti gli elementi della directory sono stati scanditi ci troviamo nella
// subdir piu' "profonda" e si puo' cominciare a considerare i files: viene
// concatenato il template di file al path attuale e si inizia un secondo ciclo
// findfirst()/findnext().

    strcpy(strrchr(path,'\\')+1,file);
    if(!findfirst(path,&ff,ALL_ATTR)) {                              // cerca i files
        do {

// Per ogni file trovato e' visualizzato il pathname a partire da quello di origine.

            strcpy(strrchr(path,'\\')+1,ff.ff_name);
            printf("%s\n",path);
        } while(!findnext(&ff));                      // procede finche' trova files
    }

// Quando anche i files della directory sono stati analizati tutti, scanDirectory()
// elimina dalla stringa il nome dell'ultimo file trovato...

    *(strrchr(path,'\\')) = NULL;

// ...e quello dell'ultima directory scandita: si "risale" cosi' di un livello
// nell'albero.

    *(strrchr(path,'\\')+1) = NULL;

// A questo punto, se l'attuale istanza di scanDirectory() e' una ricorsione (siamo
// cioe' in una subdirectory) l'esecuzione del programma prosegue con la precedente
// istanza di scanDirectory(): sono cercate altre subdirectory, e, in assenza di
// queste, i files; se invece la presente istanza di scanDirectory() e' la prima
// invocata (non c'e' stata ricorsione o sono gia' state analizzate tutte le subdir
// nonche' la directory di partenza), allora il controllo e' restituito alla
// funzione chiamante originaria: il lavoro di ricerca e' terminato.

}
```

La funzione `scanDirectory()` riceve due parametri: il primo è una stringa che rappresenta il "punto di partenza", cioè la directory all'interno della quale ricercare il file; la ricerca è estesa a tutte le subdirectory in essa presenti. Il secondo parametro è una stringa esprimente il nome (e la estensione) del file da individuare e può contenere le wildcard "`*`" e "`?`", che sono risolte dal servizio DOS sottostante alle funzioni di libreria `findfirst()` e `findnext()`. Quando, nel "territorio di caccia", è individuato un file il cui nome soddisfa il template fornito dal secondo parametro, ne viene visualizzato il pathname completo (a partire dalla directory di origine). La `scanDirectory()` può essere sperimentata con l'aiuto di una semplice `main()` che riceva dalla riga di comando (vedere pag. 105) il path di partenza ed il template di file:

```
#include <stdio.h>
#include <string.h>
#include <dir.h>

void main(int argc,char ** argv);
void scanDirectory(char *path,char *file);

void main(int argc,char **argv)
{
    char path[MAXPATH];

    if(argc != 3)
        printf("Specificare un path di partenza e un template di file\n");
    else {
        strcpy(path,argv[1]);             // copia il path di partenza nel buffer...
        strupr(path);                         // ...e lo converte tutto in maiuscole
        if(path[strlen(path)-1] != '\\')    // se non c'e' una backslash in fondo...
            strcat(path,"\\");                                    // ...la aggiunge
        scanDirectory(path,argv[2]);
    }
}
```

Per un ulteriore esempio di utilizzo della funzione di libreria `findfirst()` si veda pag. 468.

Il punto debole dell'approccio ricorsivo alla definizione di un algoritmo consiste in un utilizzo dello stack più pesante rispetto alla soluzione iterativa: ogni variabile locale definita in una funzione ricorsiva è duplicata nello stack per ogni istanza attiva. E' perciò necessario, onde evitare disastrosi problemi in fase di esecuzione[88], contenere il numero delle variabili locali (soprattutto se "ingombranti"), o richiedere al compilatore la generazione di uno stack di maggiori dimensioni[89]. E' decisamente sconsigliabile definire array nelle funzioni ricorsive: ad essi può essere sostituito un puntatore, assai più parco in termini di stack, gestito mediante l'allocazione dinamica della memoria (vedere pag. 109). Anche l'allocazione dinamica può influenzare lo spazio disponibile nello stack: uno sguardo all'organizzazione di stack e heap nei diversi modelli di memoria (pag. 143) servirà a chiarire le idee.

Inoltre, per ragioni di efficienza, è a volte opportuno dichiarare esplicitamente `near` le funzioni ricorsive, infatti esse possono essere eseguite più e più volte (come tutte le funzioni all'interno di un ciclo, ma nel caso della ricorsione è la funzione che chiama se stessa): una chiamata `near` è più veloce e impegna meno stack di una chiamata `far`. Dichiarare esplicitamente `near` le funzioni ricorsive assicura che sia generata una chiamata `near` anche quando il modello di memoria utilizzato in compilazione preveda per default chiamate `far`. Lo svantaggio di questo approccio è che una funzione dichiarata `near` può essere chiamata esclusivamente da quelle funzioni il cui codice eseguibile sia compreso nel

[88] L'allocazione di variabili locali e il passaggio dei parametri avvengono modificando i valori di alcuni regsitri della CPU dedicati proprio alla gestione dello stack. Se il programma utilizza più spazio di quanto il compilatore ha assegnato allo stack, è molto probabile che vengano sovrascritti i dati memorizzati ai suoi "confini".

[89] Il metodo per richiedere più stack del default varia da compilatore a compilatore. Alcune implementazioni del C definiscono una variabile globale intera senza segno, `_stklen`, il cui valore può essere impostato al numero di byte richiesti come area di stack per il programma. Il valore di default è spesso 4 kb. Si noti che la funzione `fattoriale()` non pone problemi di stack: ogni istanza richiede unicamente 4 byte (per il `long` passato come parametro) oltre ai 2 o 4 byte necessari per l'indirizzo di ritorno della chiamata `near` o, rispettivamente, `far`. E' anche possibile richiedere al compilatore di inserire automaticamente una routine di controllo, in entrata ad ogni funzione, il cui scopo è verificare se nello stack vi sia spazio sufficiente per tutte le variabili locali ed interrompere il programma in caso contrario. Tale opzione risulta utile in fase di sviluppo; è tuttavia opportuno non utilizzarla nel compilare la versione di programma per il rilascio finale, in quanto essa ne diminuisce leggermente l'efficienza.

medesimo segmento[90]: è un aspetto da valutare attentamente in fase di scrittura del codice, dal momento che se non si è sicuri di poter soddisfare tale condizione occorre rinunciare alla dichiarazione `near`.

main(): PARAMETRI E VALORE RESTITUITO

La funzione `main()` è presente in tutti i programmi C ed è sempre eseguita per prima, tuttavia non è necessario chiamarla dall'interno del programma[91]. La chiamata a `main()` è contenuta in un object file, fornito con il compilatore, che il linker collega automaticamente in testa al modulo oggetto prodotto dalla compilazione del sorgente. Si tratta dello *startup module* (o *startup code*)[92]: è questa, in realtà, la parte di codice eseguita per prima; lo startup module effettua alcune operazioni preliminari ed al termine di queste chiama `main()` dopo avere copiato sullo stack tre parametri, che essa può, opzionalmente, referenziare.

La tabella che segue elenca e descrive detti parametri, indicandone anche il nome convenzionalmente loro attribuito[93]:

PARAMETRI DI `main()`

NOME	TIPO	DESCRIZIONE
`argc`	`int`	Numero degli argomenti della riga di comando, compreso il nome del programma.
`argv`	`char **`	Indirizzo dell'array di stringhe rappresentanti ciascuna un parametro della riga di comando. La prima stringa è il nome del programma completo di pathname se l'esecuzione avviene in una versione di DOS uguale o successiva alla 3.0, altrimenti contiene la stringa `"C"`. L'ultimo elemento dell'array è un puntatore nullo.
`envp`	`char **`	Indirizzo dell'array di stringhe copiate dall'environment (variabili d'ambiente) che il DOS ha reso disponibile al programma. L'ultimo elemento dell'array è un puntatore nullo.

[90] Lo sono sicuramente le funzioni definite nel medesimo sorgente, dal momento che ogni modulo `.OBJ` non può superare i 64Kb.

[91] Del resto se `main()` è eseguita per prima, da quale punto del programma la si potrebbe chiamare?

[92] Il modulo di startup altro non è che un file `.OBJ`, collegato dal linker in testa al (o ai) file `.OBJ` generati a partire dal sorgente (o sorgenti) del programma, avente lo scopo di svolgere le operazioni preliminari all'invocazione di `main()`, tra le quali vi è il controllo della versione di DOS, la chiamata a `_setargv__()` e `_setenvp__()` (vedere pag. 476), etc.; lo startup module relativo al modello di memoria tiny (pag. 143 e seguenti) include la direttiva assembler `ORG 100`, che permette di ottenere un file `.COM` dall'operazione di linking. Il sorgente dello startup module, scritto solitamente in assembler, è generalmente fornito insieme al compilatore, per consentirne personalizzazioni. Un esempio di startup module (adatto però ai device driver) è presentato a pag. 386.

[93] L'ordine in cui sono elencati è quello in cui `main()` li referenzia. E' ovvio che essi sono copiati sullo stack in ordine inverso, come normalmente avviene nelle chiamate a funzione. I nomi ad essi attribuiti in tabella sono quelli convenzionalmente utilizzati dai programmatori C. Nulla vieta, se non il buon senso, si utilizzare nomi differenti nei propri programmi.

La funzione `main()` può referenziare tutti i tre argomenti o solo alcuni di essi; tuttavia deve referenziare tutti i parametri che precedono l'ultimo nell'ordine in cui sono elencati nella tabella. Vediamo:

```
void main(void);
```

Quello appena presentato è il prototipo di una `main()` che non referenzia alcuno dei tre parametri. Perché `main()` li possa referenziare tutti, il prototipo deve essere:

```
void main(int argc,char **argv,char **envp);
```

Se, ad esempio, nel programma è necessario accedere solo alle stringhe dell'environment attraverso l'array `envp`, devono essere comunque dichiarati nel prototipo anche `argc` e `argv`.

La forma di `main()` più comunemente utilizzata è quella che referenzia `argv` al fine di accedere ai parametri della riga di comando[94]. Perché possa essere utilizzato `argv` deve essere referenziato anche `argc` (il quale, da solo, in genere non è di grande utilità):

```
void main(int argc,char **argv);
```

Ecco una semplice applicazione pratica:

```
#include <stdio.h>

void main(int argc,char **argv);

void main(int argc,char **argv)
{
    register i;

    printf("%s ha ricevuto %d argomenti:\n",argv[0],argc-1);
    for(i = 1; argv[i]; i++)
        printf("%d) %s\n",i,argv[i]);
}
```

Se il programma eseguibile si chiama `PRINTARG.EXE`, si trova nella directory `C:\PROVE\EXEC` e viene lanciato al prompt del DOS con la seguente riga di comando:

```
printarg Pippo Pluto & Paperino "Nonna Papera" 33 21
```

l'output prodotto è:

```
C:\PROVE\EXEC\PRINTARG.EXE ha ricevuto 7 argomenti:
Pippo
Pluto
&
Paperino
Nonna Papera
33
21
```

E' facile notare che viene isolata come parametro ogni sequenza di caratteri compresa tra spazi; le due parole `Nonna Papera` sono considerate un unico parametro in quanto racchiuse tra virgolette.

[94] A proposito della riga di comando, a pag. 475 si dicono alcune cosette piuttosto utili.

Anche i numeri 33 e 21 sono referenziati come stringhe: per poterli utilizzare come interi è necessario convertire le stringhe in numeri, mediante le apposite funzioni di libreria[95].

Come ogni altra funzione, inoltre, `main()` può restituire un valore tramite l'istruzione `return` (vedere pag. 88); in deroga, però, alla regola generale, per la quale è possibile la restituzione di un valore di qualsiasi tipo, `main()` può restituire unicamente un valore di tipo `int`.

Vediamo, con riferimento all'esempio precedente, quali sono i cambiamenti necessari perché `main()` possa restituire il numero di argomenti ricevuti dal programma:

```
#include <stdio.h>

int main(int argc,char **argv);

int main(int argc,char **argv)
{
    register i;

    printf("%s ha ricevuto %d argomenti:\n",argv[0],argc-1);
    for(i = 1; argv[i]; i++)
        printf("%d) %s\n",i,argv[i]);
    return(argc-1);
}
```

E' stato sufficiente modificare la definizione ed il prototipo di `main()`, sostituendo il dichiaratore di tipo `void` con `int` ed inserire un'istruzione `return`, seguita dall'espressione che produce il valore da restituire.

E' cosa arcinota, ormai, che l'esecuzione di un programma C ha inizio con la prima istruzione di `main()`; è, del resto, facilmente intuibile che l'esecuzione del programma, dopo avere eseguito l'ultima istruzione di `main()`, ha termine[96]. Ma allora, quale significato ha la restituzione di un valore da parte di `main()`, dal momento che nessuna altra funzione del programma lo può conoscere? In quale modo lo si può utilizzare? La risposta è semplice: il valore viene restituito direttamente al DOS, che lo rende disponibile attraverso il registro `ERRORLEVEL`. L'utilizzo più comune è rappresentato dall'effettuazione di opportuni tests all'interno di programmi batch che sono così in grado di condizionare il flusso esecutivo in dipendenza dal valore restituito proprio da `main()`. Di seguito è presentato un esempio di programma batch utilizzante il valore restituito dalla seconda versione di `PRINTARG`:

```
@echo off
printarg %1 %2 %3 %4 %5 %6 %7 %8 %9
if errorlevel 2 goto Molti
if errorlevel 1 goto Uno
echo PRINTARG lanciato senza argomenti (ERRORLEVEL = 0)
goto Fine
:Molti
echo PRINTARG lanciato con 2 o piu' argomenti (ERRORLEVEL >= 2)
goto Fine
:Uno
```

---

[95] La funzione `atoi()`, ad esempio, richiede quale parametro una stringa e restituisce come `int` il valore numerico che essa esprime.

[96] Attenzione: sostenere che dopo avere eseguito l'ultima istruzione di `main()` il programma termina è diverso dal dire che il programma termina dopo avere eseguito l'ultima istruzione di `main()`, perché un programma può terminare anche in altri modi, e al di fuori di `main()`. Molto utilizzata è, allo scopo, la funzione di libreria `exit()`, che richiede quale parametro un intero e lo utilizza per "restituirlo" come se venisse eseguita una normale return in `main()`, e ciò anche se questa è definita `void`.

```
echo PRINTARG lanciato con un solo argomento (ERRORLEVEL = 1)
:Fine
```

Occorre prestare attenzione ad un particolare: il valore restituito da `main()` è un `int` (16 bit) e poiché, per contro, il registro `ERRORLEVEL` dispone di soli 8 bit (equivale ad un `unsigned char`) ed il valore in esso contenuto può variare da `0` a `255`, gli 8 bit più significativi del valore restituito da `main()` sono ignorati. Ciò significa, in altre parole, che l'istruzione

```
return(256);
```

in `main()` restituisce, in realtà, `0` (la rappresentazione binaria di `256` è, infatti, `0000000100000000`), mentre

```
return(257);
```

restituisce `1` (`257` in binario è `0000000100000001`).

Va ancora precisato che le regole del C standard richiedono che `main()` sia sempre dichiarata `int` e precisano che una `main()` dichiarata `void` determina un *undefined behaviour*: non è cioè possibile a priori prevedere quale sarà il comportamento del programma. Del resto, numerosi esperimenti condotti non solo in ambiente DOS consentono di affermare che dichiarare `main()` con return type `void` non comporta alcun problema (ecco perché, per semplicità, detto tipo di dichiarazione ricorre più volte nel testo): ovviamente non è possibile utilizzare il valore restituito dal programma, perché questo è sicuramente indefinito (non si può cioè prevedere a priori quale valore contiene `ERRORLEVEL` in uscita dal programma). Qualora si intenda utilizzare il sorgente in ambienti o con compilatori diversi, può essere prudente dichiarare `main()` secondo le regole canoniche o verificare che il return type `void` non sia causa di problemi.

Se utilizzata con accortezza, la descritta tecnica di utilizzo dei parametri della riga di comando e di restituzione di valori al DOS consente di realizzare, con piccolo sforzo, procedure in grado di lavorare senza alcuna interazione con l'utilizzatore, cioè in modo completamente automatizzato.

Vale infine la pena di ricordare che dichiarare il parametro `envp` di `main()` non è l'unico modo per accedere alle stringhe dell'environment: allo scopo possono essere utilizzate le funzione di libreria `getenv()` e `putenv()`: la prima legge dall'environment il valore di una variabile, mentre la seconda lo modifica.

Chi ama le cose complicate può accedere all'environment leggendone la parte segmento dell'indirizzo nel PSP del programma, e costruendo un puntatore `far` con l'aiuto della macro `MK_FP()`, definita in `DOS.H` (pag. 24). Circa il PSP vedere pag. 324.

ALLOCAZIONE DINAMICA DELLA MEMORIA

Quando è dichiarata una variabile, il compilatore riserva la quantità di memoria ad essa necessaria e le associa, ad uso dei riferimenti futuri, il nome scelto dal programmatore. Il compilatore desume dal tipo della variabile, già al momento della dichiarazione, quanti byte devono essere allocati. La stessa cosa avviene quando si dichiara un puntatore, in quanto anch'esso è una variabile, sebbene dal significato particolare: il compilatore alloca 16 o 32 bit a seconda che si tratti di un puntatore `near`, oppure `far` (pag. 21). Anche per quanto riguarda gli array (pag. 29 e seguenti) il discorso non cambia: tipo di dato e numero degli elementi dicono al compilatore quanta memoria è necessaria, durante l'esecuzione del programma, per gestire correttamente l'array; l'obbligo di indicare con una espressione costante il numero di elementi o, alternativamente, di inizializzarli contestualmente alla dichiarazione, conferma che in tutti i casi descritti la memoria è allocata in modo statico.

L'aggettivo "statico" rappresenta qui la traduzione della parola riservata `static`, introdotta a pag. 37 (il cui significato è connesso alla visibilità e durata delle variabili), ma indica semplicemente che quanto dichiarato in fase di programmazione non è modificabile durante l'esecuzione del programma stesso; in altre parole essa è un evento *compile-time* e non *run-time.*

E' facile però individuare molte situazioni in cui tale metodo di gestione della memoria si rivela inefficiente: se, per esempio, i dati da memorizzare in un array sono acquisiti durante l'esecuzione del programma e il loro numero non è noto a priori, si è costretti a dichiarare un array di dimensioni sovrabbondanti "per sicurezza": le conseguenze sono un probabile spreco di memoria e il perdurare del rischio che il numero di elementi dichiarato possa, in qualche particolare situazione, rivelarsi comunque insufficiente.

Le descritte difficoltà possono essere superate mediante l'allocazione dinamica della memoria, tecnica consistente nel riservare durante l'esecuzione del programma la quantità di memoria necessaria a contenere i dati elaborati, incrementandola o decrementandola quando necessario, e rilasciandola poi al termine delle elaborazioni in corso al fine di renderla nuovamente disponibile per usi futuri.

Gli strumenti mediante i quali il C implementa la gestione dinamica della memoria sono, in primo luogo, i puntatori (ancora loro!), unitamente ad un gruppo di funzioni di libreria dedicate, tra le quali risultano di fondamentale importanza `malloc()`, `realloc()` e `free()`, dichiarate in `ALLOC.H` (o `MALLOC.H`, a seconda del compilatore).

La funzione `malloc()` consente di *allocare*, cioè riservare ad uno specifico uso, una certa quantità di memoria: essa si incarica di rintracciare (nell'insieme della RAM che il programma è in grado di gestire) un'area sufficientemente ampia e ne restituisce l'indirizzo, cioè il puntatore al primo byte. L'area così riservata non è più disponibile per successive allocazioni (successive chiamate a `malloc()` la considereranno, da quel momento in poi, "occupata") fino al termine dell'esecuzione del programma o fino a quando essa sia esplicitamente restituita all'insieme della memoria libera mediante la funzione `free()`. La funzione `realloc()` consente di modificare le dimensioni di un'area precedentemente allocata da `malloc()`: nel caso di una richiesta di ampliamento, essa provvede a copiarne altrove in RAM il contenuto, qualora non sia possibile modificarne semplicemente il limite superiore. In altre parole, se non può spostarne il confine, `realloc()` muove tutto il contenuto dell'area là dove trova spazio sufficiente, riserva la nuova area e libera quella precedentemente occupata.

La logica del marchingegno apparirà più chiara dall'esame di un esempio. Supponiamo di volere calcolare la somma di un certo numero di interi, introdotti a run-time dall'utilizzatore del programma: dopo avere digitato l'intero si preme il tasto `RETURN` per memorizzarlo; al termine della sequenza di interi è sufficiente premere `CTRL-Z` (in luogo di un intero) e `RETURN` ancora una volta perché tutti gli interi immessi e la loro somma siano visualizzati. Di seguito è presentato il codice della funzione `sommainteri()`:

```
#include <alloc.h>                    // prototipi di malloc(), realloc() e free()
#include <stdio.h>                                // prototipi di gets() e printf()
#include <stdlib.h>                                            // prototipo di atoi()

int sommainteri(void)
{
    register i, j;
    int retcode = 0;
    long sum = 0L;
    char inBuf[10];
    int *iPtr, *iPtrBackup;

    if(!(iPtr = (int *)malloc(sizeof(int))))
        return(-1);
    for(i = 0; gets(inBuf); i++) {
        iPtr[i] = atoi(inBuf);
        sum += iPtr[i];
        iPtrBackup = iPtr;
        if(!(iPtr = (int *)realloc(iPtr,sizeof(int)*(i+2)))) {
            retcode = -1;
            iPtr = iPtrBackup;
            break;
        }
    }
    for(j = 0; j < i; j++)
        printf("%d\n",iPtr[j]);
    printf("La somma è: %ld\n",sum);
    free(iPtr);
    return(retcode);
}
```

La funzione `sommainteri()` restituisce `-1` in caso di errore, `0` se tutte le operazioni sono state compiute regolarmente.

La chiamata a `malloc()` in ingresso alla funzione alloca lo spazio necessario a contenere un intero[97] e ne assegna l'indirizzo al puntatore `iPtr`. Se il valore assegnato è nullo (uno `0` binario), la `malloc()` ha fallito il proprio obiettivo, ad esempio perché non vi è più sufficiente memoria libera: in tal caso `sommainteri()` restituisce `-1`. Si noti che `iPtr` non è dihiarato come array, proprio perché non è possibile sapere a priori quanti interi dovranno essere memorizzati: esso è un normale puntatore, che contiene l'indirizzo del primo byte dell'area di memoria individuata da `malloc()` come idonea a contenere il numero di interi desiderati (per ora uno soltanto). Dunque, la `malloc()` riceve come parametro un `unsigned int`, che esprime in byte la dimensione desiderata dell'area di memoria, e restituisce un puntatore al pimo byte dell'area allocata: detto puntatore è di tipo `void` (pag. 34), pertanto l'operatore di cast (pag. 65) evita ambiguità e messaggi di warning. Se non è possibile allocare un'area della dimensione richiesta, `malloc()` restituisce `NULL`.

Si entra poi nel primo ciclo `for`, il quale è iterato sino a che `gets()` continua a restituire un valore non nullo. La `gets()` è una funzione di libreria che memorizza in un buffer i tasti digitati e, alla pressione del tasto `RETURN`, elimina il `RETURN` stesso sostituendolo con un `NULL` (e generando così una vera e propria stringa C). La `gets()`, quando riceve una sequenza `CTRL-Z` (il carattere che per il DOS significa `EOF`, End Of File) restituisce `NULL`, perciò, digitando `CTRL-Z` in luogo di un intero, l'utilizzatore segnala alla funzione che l'input dei dati è terminato e che può esserne visualizzata la somma.

La stringa memorizzata da `gets()` in `inBuf` è convertita in intero dalla `atoi()`: questo viene, a sua volta, memorizzato nell'`i`-esimo elemento dell'array indirizzato da `iPtr`. Poco importa se,

[97] Nell'assunzione che un intero occupi 16 bit, l'uso dell'operatore `sizeof()` (pag. 68) appare ridondante, ma consente una migliore portabilità del sorgente su macchine che gestiscano gli interi, ad esempio, in 32 bit.

come si è detto, iPtr non è dichiarato come array: l'espressione iPtr[i] indica semplicemente l'intero (perché iPtr è un puntatore ad intero) che ha un offset pari a i interi[98] dall'indirizzo contenuto in iPtr. Questo è il "trucco" che ci permette di lavorare con iPtr come se fosse un array. Alla prima iterazione i vale 0, pertanto l'intero è memorizzato nel primo elemento dell'array; alla seconda iterazione i vale 1 e l'intero è memorizzato nel secondo elemento, e via di seguito.

L'intero memorizzato nell'array è sommato alla variabile sum: dal momento che questa è stata inizializzata a 0 contestualmente alla dichiarazione, al termine della fase di input essa contiene la somma di tutti gli interi introdotti.

A questo punto occorre predisporsi ad un'ulteriore iterazione: bisogna "allungare" l'array, per riservare spazio al prossimo intero in arrivo. A ciò provvede la realloc(), che richiede due parametri: il primo è l'indirizzo dell'area da estendere (o contrarre), iPtr nel nostro caso, mentre il secondo è la nuova dimensione, in byte, desiderata per l'area. Qui il risultato restituito dall'operatore sizeof() è molitplicato per i+2: l'operazione è necessaria perché ad ogni iterazione l'array è già costituito di un numero di elementi pari a i+1. Nella prima iterazione, infatti, i vale 0 e l'array contiene già l'elemento allocato da malloc(). Alla realloc() bisogna quindi richiedere un'area di memoria ampia quanto basta a contenere 2 elementi; alla terza iterazione i vale 1, e gli elementi desiderati sono 3, e via di seguito. Come prevedibile, realloc() restituisce il nuovo indirizzo dell'area di memoria[99], ma se non è stato possibile ampliare quella precedentemente allocata neppure "spostandola" altrove, essa restituisce NULL, proprio come malloc(). Ciò spiega perché, prima di chiamare realloc(), il valore di iPtr è assegnato ad un altro puntatore (iPtrBackup): in tal modo, se realloc() fallisce è ancora possibile visualizzare la somma di tutti gli interi immessi sino a quel momento, riassegnando a iPtr il valore precedente alla chiamata. In questo caso, inoltre, sommainteri() deve comunque restituire -1 (tale valore è infatti assegnato a retcode) ed occorre forzare (break) l'uscita dal ciclo for.

In uscita dal ciclo, la variabile i contiene il numero di interi immessi ed è perciò pari all'indice, aumentato di uno, dell'elemento di iPtr contenente l'ultimo di essi. Nel secondo ciclo for, pertanto, i può essere utilizzata come estremo superiore (escluso) dei valori del contatore.

Dopo avere visualizzato tutti gli elementi dell'array e la loro somma, ma prima di terminare e restituire alla funzione chiamante il valore opportuno, sommainteri() deve liberare la memoria indirizzata da iPtr mediante una chiamata a free(), che riceve come parametro proprio l'indirizzo dell'area da rilasciare (e non restituisce alcun valore). Se non venisse effettuata questa operazione, la RAM indirizzata da iPtr non potrebbe più essere utilizzata per altre elaborazioni: i meccanismi C di gestione dinamica della memoria utilizzano infatti una tabella, non visibile al programmatore, che tiene traccia delle aree occupate tramite il loro indirizzo e la loro dimensione. Detta tabella è unica e globale per l'intero programma: ciò significa che un'area allocata dinamicamente in una funzione resta allocata anche dopo l'uscita da quella funzione; tuttavia essa può essere utilizzata da altre funzioni solo se queste ne conoscono l'indirizzo. E' ancora il caso di sottolineare che l'area rimane allocata anche quando il programma non ne conservi l'indirizzo: è proprio questo il caso di sommainteri(), perché iPtr è una variabile automatica, e come tale cessa di esistere non appena la funzione restituisce il controllo alla chiamante.

Se ne trae un'indicazione tecnica di estrema importanza: la memoria allocata dinamicamente non fa parte dello stack, ma di una porzione di RAM sottoposta a regole di utilizzo differenti, detta *heap*[100]. L'allocazione dinamica rappresenta perciò un'eccellente soluzione ai problemi di consumo dello

---

[98] E' importante sottolineare che l'offset è calcolato in termini di interi e non di byte. Chi avesse dubbi in proposito farà bene a rivedere la chiacchierata sull'aritmetica dei puntatori (pag. 33).

[99] E' lo stesso indirizzo passato come parametro se oltre il limite superiore dell'area attualmente allocata vi è sufficiente memoria libera per ampliarla semplicemente spostandone il confine.

[100] Se si alloca memoria per puntatori far, come si accennerà tra breve, si parla di *far heap*.

stack che possono presentarsi nell'implementazione di algoritmi ricorsivi (pag. 100); si tenga tuttavia presente che in alcuni modelli di memoria stack e heap condividono lo stesso spazio fisico, e quindi utilizzare heap equivale a sottrarre spazio allo stack. La condivisione degli indirizzi è possibile perché lo stack li utilizza dal maggiore verso il minore, mentre nello heap la memoria è sempre allocata a partire dagli indirizzi liberi inferiori. Un'occhiata agli schemi di pagina 143 e seguenti può chiarire questi aspetti, di carattere meramente tecnico.

Quando un'area di memoria allocata dinamicamente deve essere utilizzata al di fuori della funzione che effettua l'allocazione, questa può restituirne l'indirizzo oppure può memorizzarlo in un puntatore appartenente alla classe external (pag. 39)[101]. Ecco una nuova versione di `sommainteri()`, che restituisce il puntatore all'area di memoria se l'allocazione è avvenuta correttamente e `NULL` se `malloc()` o `realloc()` hanno determinato un errore:

```
#include <alloc.h>                         // prototipi di malloc(), realloc() e free()
#include <stdio.h>                                    // prototipi di gets() e printf()
#include <stdlib.h>                                               // prototipo di atoi()

int *sommainteri2(void)
{
    register i, j;
    long sum = 0L;
    char inBuf[10];
    int *iPtr, *iPtrBackup, *retPtr;

    if(!(iPtr = (int *)malloc(sizeof(int))))
        return(NULL);
    for(i = 0; gets(inBuf); i++) {
        iPtr[i] = atoi(inBuf);
        sum += iPtr[i];
        iPtrBackup = retPtr = iPtr;
        if(!(iPtr = (int *)realloc(iPtr,sizeof(int)*(i+2)))) {
            retPtr = NULL;
            iPtr = iPtrBackup;
            break;
        }
    }
    for(j = 0; j < i; j++)
        printf("%d\n",iPtr[j]);
    printf("La somma è: %ld\n",sum);
    return(retPtr);
}
```

Le modifiche di rilievo sono tre: innanzitutto, la funzione è dichiarata di tipo `int *`, in quanto restituisce un puntatore ad interi e non più un intero; in secondo luogo, per lo stesso motivo, la variabile intera `retcode` è stata sostituita con un terzo puntatore ad intero, `retPtr`.

Le terza e più rilevante modifica consiste nell'eliminazione della chiamata a `free()`: è evidente, del resto, che non avrebbe senso liberare l'area allocata prima ancora di restituirne l'indirizzo. La chiamata a `free()` può essere effettuata da qualunque altra funzione, purché conosca l'indirizzo restituito da `sommainteri2()`.

Quando sia necessario allocare dinamicamente memoria ed assegnarne l'indirizzo ad un puntatore `far` o `huge`, occorre utilizzare `farmalloc()`, `farrealloc()` e `farfree()`, del tutto

---

[101] Chi ama le cose difficili può dichiarare il puntatore nella funzione che chiama quella in cui avviene l'allocazione, e passarne a questa, come parametro, l'indirizzo. Il valore restituito da `malloc()` deve allora essere memorizzato nell'indirezione del puntatore ricevuto come parametro.

analoghe a malloc(), realloc() e free() ma adatte a lavorare su puntatori a 32 bit[102]. Ancora una volta bisogna accennare alla logica dei modelli di memoria: quando un programma è compilato in modo tale che tutti i puntatori non esplicitamente dichiarati near siano considerati puntatori a 32 bit, non è necessario utilizzare le funzioni del gruppo di farmalloc(), in quanto malloc(), realloc() e free() lavorano esse stesse su puntatori far (vedere anche pag. 21).

---

[102] Le funzioni farmalloc() e farrealloc() restituiscono comunque puntatori far: per trasformare l'indirizzo restituito in un puntatore huge è necessaria un'operazione di cast:

```
double huge *hPtr;
....
hPtr = (double huge *)farmalloc(100*sizeof(double));
```

Il numero di byte occorrenti è calcolato moltiplicando il numero di double (100 nell'esempio) per la dimensione del dato double (l'operatore sizeof() è descritto a pag. 68).

L'I/O E LA GESTIONE DEI FILE

Per Input/Output (I/O) si intende l'insieme delle operazioni di ingresso ed uscita, cioè di scambio di informazioni tra il programma e le unità periferiche del calcolatore (video, tastiera, dischi, etc.). Dal punto di vista del supporto dato dal linguaggio di programmazione alla gestione dello I/O, va sottolineato che il C non comprende alcuna istruzione rivolta alla lettura dalle periferiche né alla scrittura su di esse. In C l'I/O è interamente implementato mediante funzioni di libreria, in coerenza con la filosofia che sta alla base del C stesso, cioè di un linguaggio il più possibile svincolato dall'ambiente in cui il programma deve operare, e pertanto portabile.

Ciononostante, la gestione delle operazioni di I/O, in C, è piuttosto standardizzata, in quanto sono state sviluppate funzioni dedicate che, nel tempo, sono entrate a far parte della dotazione standard di libreria che accompagna quasi tutti i compilatori[103].

Le prime versioni di dette librerie sono state sviluppate, inizialmente, in ambiente Unix, sistema operativo in cui le periferiche sono trattate, a livello software, come file. Il linguaggio C consente di sfruttare tale impostazione, mediante il concetto di *stream*, cioè di flusso di byte da o verso una periferica. Alla luce delle considerazioni espresse, sembra di poter azzardare che leggere dati da un file non sia diverso che leggerli dalla tastiera e scrivere in un file sia del tutto analogo a scrivere sul video. Ebbene, in effetti è proprio così: associando ad ogni periferica uno stream, esse possono essere gestite, ad alto livello, nello stesso modo.

GLI STREAM

Dal punto di vista tecnico, uno stream è una implementazione software in grado di gestire le informazioni relative all'interazione a basso livello con la periferica associata, in modo che il programma possa trascurarne del tutto la natura. Lo stream rappresenta, per il programmatore, una interfaccia per la lettura e l'invio di dati tra il software e la periferica: non riveste alcuna importanza come il collegamento tra dati e periferiche sia realizzato; l'isolamento tra l'algoritmo e la "ferraglia" è forse il vantaggio più interessante offerto dagli streams.

Stream standard

Il DOS rende disponibili ad ogni programma 5 stream che possono essere utilizzati per leggere e scrivere dalla o sulla periferica associata. Essi sono i cosiddetti streams standard: la tabella che segue li elenca e li descrive.

---

[103] Ciò non significa, purtroppo, che le implementazioni del C siano identiche in tutti gli ambienti. Le caratteristiche di alcuni tipi di macchina, nonché quelle di certi sistemi operativi, possono rendere piuutosto difficile realizzare ciò che in altri appare invece naturale. Ne consegue che, in ambienti diversi, non sempre la libreria standard contiene le stesse funzioni; inoltre, una stessa funzione può avere in diverse librerie caratteristiche lievemente differenti (vedere anche pag. 461 e seguenti).

STREAM STANDARD

NOME DOS	NOME C	PERIFERICA ASSOCIATA PER DEFAULT	FLUSSO	DESCRIZIONE
`CON:`	`stdin`	tastiera	In	*Standard Input*. Consente di ricevere input dalla tastiera. Può essere rediretto su altre periferiche.
`CON:`	`stdout`	video	Out	*Standard Output*. Consente di scrivere sul video della macchina. Può essere rediretto su altre periferiche.
	`stderr`	video	Out	*Standard Error*. Consente di scrivere sul video della macchina. Non può essere rediretto. E' generalmente utilizzato per i messaggi d'errore.
`COM1:`	`stdaux`	prima porta seriale	In/Out	*Standard Auxiliary*. Consente di inviare o ricevere dati attraverso la porta di comunicazione asincrona. Può essere rediretto.
`LPT1:`	`stdprn`	prima porta parallela	Out	*Standard Printer*. Consente di inviare dati attraverso la porta parallela. Può essere rediretto. E' generalmente utilizzato per gestire una stampante.

Per maggiori approfondimenti si rimanda alla documentazione che accompagna il sistema operativo DOS. Qui preme sottolineare che ogni programma C ha la possibilità di sfruttare il supporto offerto dal sistema attraverso l'implementazione degli streams offerta dal linguaggio; va tuttavia tenuto presente che i nomi ad essi associati differiscono da quelli validi in DOS, come evidenziato dalla tabella.

Gli stream in C

Un programma C può servirsi degli stream standard senza alcuna operazione preliminare: è sufficiente che nel sorgente compaia la direttiva

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

Di fatto, molte funzioni standard di libreria che gestiscono l'I/O li utilizzano in modo del tutto trasparente: ad esempio `printf()` non scrive a video, ma sullo stream `stdout`. L'output di

printf() compare perciò a video solo in assenza di operazioni di redirezione DOS[104] dello stesso su altre periferiche, o meglio su altri streams. La funzione fgetchar(), invece, legge un carattere dallo standard input, cioè da stdin: con un'operazione di redirezione DOS è possibile forzarla a leggere il carattere da un altro stream.

Esistono, in C, funzioni di libreria che richiedono di specificare esplicitamente qual è lo stream su cui devono operare[105]. Si può citare, ad esempio, la fprintf(), che è del tutto analoga alla printf() ma richiede un parametro aggiuntivo: prima del puntatore alla stringa di formato deve essere indicato lo stream su cui effettuare l'output. Analogamente, la fgetc() può essere considerata analoga alla getchar(), ma richiede che le sia passato come parametro lo stream dal quale effettuare la lettura del carattere.

Vediamole al lavoro:

```
    ....
    int var;
    char *string;

    printf("Stringa: %s\nIntero: %d\n",string,var);
    fprintf(stdout,"Stringa: %s\nIntero: %d\n",string,var);
    fprintf(stderr,"Stringa: %s\nIntero: %d\n",string,var);
    ....
```

Nell'esempio, la chiamata a printf() e la prima delle due chiamate a fprintf() sono assolutamente equivalenti e producono outputs perfettamente identici sotto tutti i punti di vista[106]. La seconda chiamata a fprintf(), invece, scrive ancora la medesima stringa, ma la scrive su stderr, cioè sullo standard error: a prima vista può risultare un po' difficile cogliere la differenza, perché il DOS associa al video sia stdout che stderr, perciò, per default, tutte le tre stringhe (identiche) sono scritte a video. La diversità di comportamento degli stream appare però evidente quando sulla riga di comando si effettui una redirezione dell'output su un altro stream, ad esempio un file: in esso è scritto l'ouptut prodotto da printf() e dalla prima chiamata a fprintf(), mentre la stringa scritta dalla seconda chiamata a fprintf() continua a comparire a video, in quanto, come si è detto poco fa, il DOS consente la redirezione dello standard output, ma non quella dello standard error.

E' immediato trarre un'indicazione utile nella realizzazione di programmi che producono un output destinato a successive post-elaborazioni: se esso è scritto su stdout, ad eccezione dei messaggi di copyright, di errore o di controllo, inviati allo standard error, con una semplice redirezione sulla riga di comando è possibile memorizzare in un file tutto e solo l'output destinato ad elaborazioni successive. Il programma, inoltre, potrebbe leggere l'input da stdin per poterlo ricevere, anche in questo caso con un'operazione di redirezione, da un file o da altre periferiche.

Il C non limita l'uso degli stream solamente in corrispondenza con quelli standard resi disponibili dal DOS; al contrario, via stream può essere gestito qualunque file. Vediamo come:

```
#include <stdio.h>

    ....
    FILE *outStream;
    char *string;
```

---

[104] Con *redirezione DOS* si intende un'operazione effettuata al di fuori del codice del programma, direttamente sulla riga di comando che lo lancia, con i simboli ">", ">>" o "<".

[105] Si tenga presente che i prototipi di tutte le funzioni operanti su stream si trovano in STDIO.H.

[106] Non è escluso che nelle librerie di qualche compilatore C la printf() sia implementata semplicemente come un "guscio" che chiama fprintf() passandole tutti i parametri ricevuti, ma preceduti da stdout.

```
int var;

....
if(!(outStream = fopen("C:\PROVE\PIPPO","wt")))
    fprintf(stderr,"Errore nell'apertura del file.\n");
else {
    if(fprintf(outStream,"Stringa: %s\nIntero: %d\n",string,var) == EOF)
        fprintf(stderr,"Errore di scrittura nel file.\n");
    fclose(outStream);
}
....
```

Quelle appena riportate sono righe di codice ricche di novità. In primo luogo, la dichiarazione

```
FILE *outStream;
```

appare piuttosto particolare. L'asterisco indica che `outStream` è un puntatore, ma a quale tipo di dato? Il tipo `FILE` non esiste tra i tipi intrinseci... Ebbene, `FILE` è un tipo di dato generato mediante lo specificatore `typedef`, che consente di creare sinonimi per i tipi di dato. Non pare il caso di approfondirne sintassi e modalità di utilizzo; in questa sede basta sottolineare che quella presentata è una semplice dichiarazione di stream. Infatti, il dichiaratore `FILE` cela una `struct` ed evita una dichiarazione più complessa, del tipo `struct...`[107]; `outStream` è quindi, in realtà, un puntatore alla struttura utilizzata per implementare il meccanismo dello stream, perciò possiamo riferirci direttamente ad esso proprio come ad uno stream. Ora è tutto più chiaro (insomma...): la prima operazione da effettuare per poter utilizzare uno stream è dichiararlo, con la sintassi che abbiamo appena descritto.

L'associazione dello stream al file avviene mediante la funzione di libreria `fopen()`, che riceve quali parametri due stringhe, contenenti, rispettivamente, il nome del file (eventualmente completo di path) e l'indicazione della modalità di apertura del medesimo. Aprire un file significa rendere disponibile un "canale" di accesso al medesimo, attraverso il quale leggere e scrivere i dati; il nome del file deve essere valido secondo le regole del sistema operativo (il DOS, nel nostro caso), mentre le modalità possibili di apertura sono le seguenti:

[107] Ecco la definizione del tipo `FILE` data da `typedef`:

```
typedef struct {
        int             level;
        unsigned        flags;
        char            fd;
        unsigned char   hold;
        int             bsize;
        unsigned char *buffer;
        unsigned char *curp;
        unsigned        istemp;
        short           token;
} FILE;
```

MODALITÀ DI APERTURA DEL FILE CON `fopen()`

MODO	SIGNIFICATO
`"r"`	sul file sono possibili solo operazioni di lettura; il file deve esistere.
`"w"`	sul file sono possibili solo operazioni di scrittura; il file, se non esistente, viene creato; se esiste la sua lunghezza è troncata a 0 byte.
`"a"`	sul file sono possibili solo operazioni di scrittura, ma a partire dalla fine del file (*append mode*); in pratica il file può essere solo "allungato", ma non sovrascritto. Il file, se non esistente, viene creato.
`"r+"`	sul file sono possibili operazioni di lettura e di scrittura. Il file deve esistere.
`"w+"`	sul file sono possibili operazioni di lettura e di scrittura. Il file, se non esistente, viene creato; se esiste la sua lunghezza è troncata a 0 byte.
`"a+"`	sul file sono possibili operazioni di lettura e di scrittura, queste ultime a partire dalla fine del file (*append mode*); in pratica il file può essere solo "allungato", ma non sovrascritto. Il file, se non esistente, viene creato.

La `fopen()` restituisce un valore che deve essere assegnato allo stream[108], perché questo possa essere in seguito utilizzato per le desiderate operazioni sul file; in caso di errore viene restituito `NULL`. Abbiamo ora le conoscenze che servono per interpretare correttamente le righe di codice

```
if(!(outStream = fopen("C:\PROVE\PIPPO","wt")))
    fprintf(stderr,"Errore nell'apertura del file.\n");
```

Con la chiamata a `fopen()` viene aperto il file `PIPPO` (se non esiste viene creato), per operazioni di sola scrittura, con traslazione automatica `CR`/`CR-LF`. Il file aperto è associato allo stream `outStream`; in caso di errore (`fopen()` restituisce `NULL`) viene visualizzato un opportuno messaggio (scritto sullo standard error).

La scrittura nel file è effettuata da `fprintf()`, in modo del tutto analogo a quello già sperimentato con `stdout` e `stderr`; la sola differenza è che questa volta lo stream si chiama `outStream`. La `fprintf()` restituisce il numero di caratteri scritti; la restituzione del valore associato alla costante manifesta `EOF` (definita in `STDIO.H`) significa che si è verificato un errore. In tal caso viene ancora una volta usata `fprintf()` per scrivere un messaggio di avvertimento su standard error.

[108] E' evidente che quel valore è l'indirizzo della struttura di tipo `FILE`: non ha molta importanza; per noi rappresenta lo stream. Circa la `fopen()` va ancora precisato che ciascuna delle stringhe di indicazione della modalità di apertura può essere modificata aggiungendo il carattere "b" o, in alternativa, "`t`". La "`t`" indica che ogni '`\n`', prima di essere scritto in un file, deve essere trasformato in una coppia "`\r\n`" e che in lettura deve essere effettuata la trasformazione opposta: tale impostazione si rivela comoda per molte operazioni su file ASCII. La "`b`" invece indica che il file deve essere considerato "binario", e non deve essere effettuata alcuna trasformazione. La modalità di default è, di norma, "`t`"; nel C Borland è definita la variabile external `_fmode`, che consente di specificare il default desiderato per il programma.

Al termine delle operazioni sul file è opportuno "chiuderlo", cioè rilasciare le risorse di sistema che il DOS dedica alla sua gestione. La funzione `fclose()`, inoltre, rilascia anche lo stream precedentemente allocato[109] da `fopen()`, che non può più essere utilizzato, salvo, naturalmente, il caso in cui gli sia assegnato un nuovo valore restituito da un'altra chiamata alla `fopen()`.

Vi sono altre funzioni di libreria operanti su stream: ecco un esempio.

```
#include <stdio.h>

    ....
    int iArray[100], iBuffer[50];
    FILE *stream;

    ....
    if(!(fstream = fopen("C:\PROVE\PIPPO","w+b")))
        fprintf(stderr,"Errore nell'apertura del file.\n");
    ....
    if(fwrite(iArray,sizeof(int),50,fstream) < 50*sizeof(int))
        fprintf(stderr,"Errore di scrittura nel file.\n");
    ....
    if(fseek(fstream,-(long)(10*sizeof(int)),SEEK_END)
        fprintf(stderr,"Errore di posizionamento nel file.\n");
    ....
    if(fread(iBuffer,sizeof(int),10,fstream) < 10)
        fprintf(stderr,"Errore di lettura dal file.\n");
    ....
    if(fseek(fstream,0L,SEEK_CUR)
        fprintf(stderr,"Errore di posizionamento nel file.\n");
    ....
    if(fwrite(iArray+50,sizeof(int),50,fstream) < 50)
        fprintf(stderr,"Errore di scrittura sul file.\n");
    ....
    fclose(fstream);
    ....
```

Con la `fopen()` il file viene aperto (per lettura/scrittura in modalità binaria) ed associato allo stream `fstream`: sin qui nulla di nuovo. Successivamente la `fwrite()` scrive su `fstream` 50 interi "prelevandoli" da `iArray`: dal momento che la modalità di apertura `"w"` implica la distruzione del contenuto del file se questo esiste, o la creazione di un nuovo file (se non esiste), i 100 byte costituiscono, dopo l'operazione, l'intero contenuto del file. La `fwrite()`, in contrapposizione alla `fprintf()`, che scrive sullo stream secondo le specifiche di una stringa di formato, è una funzione dedicata alla scrittura di output non formattato e proprio per questa caratteristica essa è particolarmente utile alla gestione di dati binari (come gli interi del nostro esempio). La sintassi è facilmente deducibile, ma vale la pena di dare un'occhiata al prototipo della funzione:

```
int fwrite(void *buffer,int size,int count,FILE *stream);
```

Il primo parametro è il puntatore al buffer contenente i dati (o meglio, puntatore al primo dei dati da scrivere). E' un puntatore `void`, in quanto in sede di definizione della funzione non ha senso indicare a priori quale tipo di dato deve essere gestito: di fatto, in tal modo tutti i tipi sono ammissibili. Il secondo parametro esprime la dimensione di ogni singolo dato, e si rende necessario per le medesime ragioni poc'anzi espresse; infatti la `fwrite()` consente di scrivere direttamente ogni tipo di "oggetto", anche strutture o unioni; è sufficiente specificarne la dimensione, magari con l'aiuto dell'operatore

[109] La `fclose()` richiede che le sia passato come parametro lo stream da chiudere e non restituisce alcun valore. Per chiudere tutti gli stream aperti dal programma è disponibile la `fcloseall()`, che non richiede alcun parametro.

`sizeof()`, come nell'esempio. Il terzo parametro è il numero di dati da scrivere: `fwrite()` calcola il numero di byte che deve essere scritto con il prodotto di `count` per `size`. L'ultimo parametro, evidentemente, è lo stream. La `fwrite()` restituisce il numero di oggetti (gruppi di byte di dimensione pari al valore del secondo parametro) realmente scritti: tale valore risulta inferiore al terzo parametro solo in caso di errore (disco pieno, etc.): ciò chiarisce il significato della `if` in cui sono collocate le chiamate alla funzione.

La seconda novità dell'esempio è la `fseek()`, che consente di riposizionare il puntatore al file, cioè di muoversi avanti e indietro lungo il medesimo per stabilire il nuovo punto di partenza delle successive operazioni di lettura o scrittura.

Il primo parametro della `fseek()` è lo stream, mentre il secondo è un `long` che esprime il numero di byte dello spostamento desiderato; il valore è negativo se lo spostamento procede dal punto di partenza verso l'inizio del file, positivo se avviene in direzione opposta. Il punto di partenza è rappresentato dal terzo parametro (un intero), per il quale è comodo utilizzare le tre costanti manifeste appositamente definite in `STDIO.H`:

MODALITÀ OPERATIVE DI `fseek()`

COSTANTE	**SIGNIFICATO**
`SEEK_SET`	lo spostamento avviene a partire dall'inizio del file.
`SEEK_CUR`	lo spostamento avviene a partire dall'attuale posizione.
`SEEK_END`	lo spostamento avviene a partire dalla fine del file.

La `fseek()` restituisce `0` se l'operazione riesce; in caso di errore è restituito un valore diverso da `0`.

Il codice dell'esempio, pertanto, con la prima delle due chiamate ad `fseek()` sposta indietro di 20 byte, a partire dalla fine del file, il puntatore allo stream, preparando il terreno alla `fread()`, che legge gli ultimi 10 interi del file.

La `fread()` è evidentemente complementare alla `fwrite()`: legge da uno stream dati non formattati. Anche i parametri omologhi delle due funzioni corrispondono nel tipo e nel significato, con la sola differenza che `buffer` esprime l'indirizzo al quale i dati letti dal file vengono memorizzati. Il valore restituito da `fread()`, ancora una volta di tipo `int`, esprime il numero di oggetti effettivamente letti, minore del terzo parametro qualora si verifichi un errore.

L'esempio necessita ancora un chiarimento, cioè il ruolo della seconda chiamata a `fseek()`: il secondo parametro, che come si è detto esprime "l'entità", cioè il numero di byte, dello spostamento, è nullo. La conseguenza immediata è che, in questo caso, la `fseek()` non effettua alcun riposizionamento; tuttavia la chiamata è indispensabile, in quanto, per caratteristiche strutturali del sistema DOS, tra una operazione di lettura ed una di scrittura (o viceversa) su stream ne deve essere effettuata una di *seek*, anche fittizia[110].

Il frammento di codice riportato si chiude con una seconda chiamata a `fwrite()`, che scrive altri 50 interi "allungando" il file (ogni operazione di lettura o di scrittura avviene, in assenza di chiamate ad `fseek()`, a partire dalla posizione in cui è terminata l'operazione precedente).

Infine, il file è chiuso dalla `fclose()`.

---

[110] Ancora una particolarità su `fseek()`: uno spostamento di `0` byte è il solo affidabile su stream associati a file aperti in modo text ("`t`"). Le funzioni di libreria dedicate agli stream non gestiscono correttamente il computo dei caratteri scritti o letti quando alcuni di essi siano eliminati dal vero flusso di dati (o aggiunti ad esso).

Vale ancora la pena di soffermarsi su un'altra funzione, che può essere considerata complementare della `fprintf()`: si tratta della `fscanf()`, dedicata alla lettura da stream di input formattato.

Come `fprintf()`, anche `fscanf()` richiede che i primi due parametri siano, rispettivamente, lo stream e una stringa di formato ed accetta un numero variabile di parametri. Tuttavia vi è tra le due una sostanziale differenza: i parametri di `fscanf()` che seguono la stringa di formato sono puntatori alle variabili che dovranno contenere i dati letti dallo stream. L'uso dei puntatori è indispensabile, perché `fscanf()` deve restituire alla funzione chiamante un certo numero di valori, cioè modificare il contenuto di un certo numero di variabili: dal momento che in C le funzioni possono restituire un solo valore e, comunque, il passaggio dei parametri avviene mediante una copia del dato originario (pag. 87), l'unico metodo possibile per modificare effettivamente quelle variabili è utilizzare puntatori che le indirizzino.

E' ovvio che lo stream passato a `fscanf()` è quello da cui leggere i dati, e la stringa di formato descrive l'aspetto di ciò che la funzione legge da quello stream. In particolare, per ogni carattere diverso da spazio, tabulazione, a capo ("\n") e percentuale `fscanf()` si aspetta in arrivo dallo stream proprio quel carattere; in corrispondenza di uno spazio, tabulazione o ritorno a capo la funzione continua a leggere dallo stream in attesa del primo carattere diverso da uno dei tre e trascura tutti gli spazi, tabulazioni e ritorni a capo; il carattere "`%`", invece, introduce una specifica di formato che indica a `fscanf()` come convertire i dati provenienti dallo stream. E' evidente che deve esserci una corrispondenza tra le direttive di formato e i puntatori passati alla funzione: ad esempio, ad una direttiva "`%d`", che indica un intero, deve corrispondere un puntatore ad intero. Il carattere "`*`" posto tra il carattere "`%`" e quello che indica il tipo di conversione indica a `fscanf()` di ignorare quel campo.

La `fscanf()` restituisce il numero di campi ai quali ha effettivamente assegnato un valore.

Ed ecco alcuni esempi:

```
#include <stdio.h>

    ....
    FILE *fstream;
    int iVar;
    char cVar, string[80];
    float fVar;

    ....
    fscanf(fstream,"%c %d %s %f",&cVar,&iVar,string,&fVar);
    printf("%c %d %s %f\n",cVar,iVar,string,fVar);
    ....
```

La stringa di formato passata a `fscanf()` ne determina il seguente comportamento: il primo carattere letto dallo stream è memorizzato nella variabile `cVar`; quindi sono ignorati tutti i caratteri spazio, tabulazione, etc. (blank) incontrati, sino al primo carattere (cifra decimale) facente parte di un intero, che viene memorizzato in `iVar`. Tutti gli spazi incontrati dopo l'intero sono trascurati e il primo non-blank segna l'inizio della stringa da memorizzare in `string`, la quale è chiusa dal primo blank incontrato successivamente. Ancora una volta, tutti i blank sono scartati fino al primo carattere (cifra decimale) del numero in virgola mobile, memorizzato in `fVar`. Il primo blank successivamente letto determina il ritorno di `fscanf()` alla funzione chiamante. E' importante notare che a `fscanf()` sono passati gli indirizzi delle variabili mediante l'operatore "`&`" (pag. 17); esso non è necessario per la sola `string`, in quanto il nome di un array ne rappresenta l'indirizzo (pag. 29 e seguenti).

E' importante sottolineare che per `fscanf()` ciò che proviene dallo stream è una sequenza continua di caratteri, che viene interrotta solo dal terminatore (blank) che chiude l'ultimo campo specificato nella stringa di formato. Se, ad esempio, il file contiene

```
x 123 ciao 456.789 1
```

`fscanf()` assegna `x` a `cVar`, `123` a `iVar`, `"ciao"` a `string` e `456.789` a `fVar`, come del resto ci si aspetta, e la cifra `1` non viene letta: può esserlo in una successiva operazione di input dal medesimo stream. Ma se il contenuto del file è

```
x123ciao456.789 1
```

`'x'` è correttamente assegnato a `cVar`, poiché lo specificatore `%c` implica comunque la lettura di un solo carattere. Anche il numero `123` è assegnato correttamente a `iVar`, perché `fscanf()` "capisce" che il carattere `'c'` non può far parte di un intero. Ma la mancanza di un blank tra `ciao` e `456.789` fa sì che `fscanf()` assegni a `string` la sequenza `"ciao456.789"` e il numero `1.00000` a `fVar`. Inoltre, lo specificatore `%c` considera i non-blanks equivalenti a qualsiasi altro carattere: se il file contiene la sequenza

```
\n123 ciao 456.789 1
```

alla variabile `cVar` è comunque assegnato il primo carattere letto, cioè il `'\n'` (a capo). La `fscanf()`, inoltre, riconosce comunque i blanks come separatori di campo, a meno che non le sia indicato esplicitamente di leggerli: se la stringa di formato è `"%c%d%s%f"`, il comportamento della funzione con i dati degli esempi appena visti risulta invariato.

Vediamo ora un esempio relativo all'utilizzo del carattere di soppressione `'*'`, che forza `fscanf()` ad ignorare un campo: nella chiamata

```
    fscanf(fstream,"%d %*d %f",&iVar,&fVar);
```

è immediato notare che i puntatori passati alla funzione sono due, benché la stringa di formato contenga tre specificatori. Il carattere `'*'` inserito tra il `'%'` e la `'d'` del secondo campo forza `fscanf()` ad ignorare (e quindi a non assegnare alla variabile indirizzata da alcun puntatore) i dati corrispondenti a quel campo. Perciò, se il file contiene

```
123 45 67.89
```

l'intero `123` è assegnato a `iVar`, l'intero `45` è ignorato e il numero in virgola mobile `67.89` è assegnato a `fVar`.

Con gli specificatori di formato è possibile indicare l'ampiezza di campo, quando questa è costante[111]. Consideriamo la seguente chiamata:

```
    fscanf(fstream,"%3d%*2d%5f",&iVar,&fVar);
```

Se i dati letti sono

```
1234567.89
```

il risultato è assolutamente identico a quello dell'esempio precedente: le costanti inserite nelle specifiche di formato indicano a `fscanf()` di quanti caratteri si compone ogni campo e quindi essa è in grado di operare correttamente anche in assenza di blank.

Vediamo ancora un esempio: supponendo di effettuare due volte la chiamata

```
    fscanf(fstream,"%c%3d%*2d%5f",&cVar,&iVar,&fVar);
```

---

[111] A dire il vero, ciò vale anche per `printf()` e `fprintf()`. L'analogia tra queste funzioni è tale da fa pensare che esista anche una `scanf()`... ed è proprio così. La `scanf()` equivale ad una `fscanf()` chiamata passandole `stdin` come parametro stream.

senza operazioni di seek sullo stream e nell'ipotesi che i dati presenti nel file siano

```
01234567.89\n01234567.89
```

ci si aspetta, presumibilmente, di memorizzare in entrambi i casi, in `cVar`, `iVar` e `fVar`, rispettivamente, `'0'`, `123` e `67.89`. Invece accade qualcosa di leggermente diverso: con la prima chiamata il risultato è effettivamente quello atteso, mentre con la seconda i valori assunti da `cVar`, `iVar` e `fVar` sono, nell'ordine, `'\n'`, `12` e `567.8`. Il motivo di tale comportamento, anomalo solo in apparenza, è che, come accennato, lo stream è per `fscanf()` semplicemente una sequenza di caratteri in ingresso, pertanto nessuno di essi, neppure il ritorno a capo, può essere scartato se ciò non è esplicitamente richiesto dal programmatore. Per leggere correttamente il file è necessaria una stringa di formato che scarti il carattere incontrato dopo il `float`, oppure indichi la presenza, dopo il medesimo, di un blank: `"%c%3d%*2d%5f%*c"` e `"%c%3d%2*d%5f "` raggiungono entrambe l'obiettivo.

Le considerazioni espresse sin qui non esauriscono la gestione degli streams in C: le librerie standard dispongono di altre funzioni dedicate; tuttavia quelle presentate sono di utilizzo comune. Tutti i dettagli sintattici possono essere approfonditi sulla manualistica del compilatore utilizzato; inoltre, un'occhiatina a `STDIO.H` è sicuramente fonte di notizie e particolari interessanti.

IL CACHING

La gestione dei file implica la necessità di effettuare accessi, in lettura e scrittura, ai dischi, che, come qualsiasi periferica hardware, hanno tempi di risposta più lenti della capacità elaborativa del microprocessore[112]. L'efficienza delle operazioni di I/O su file può essere incrementata mediante l'utilizzo di uno o più buffer, gestiti mediante algoritmi di lettura ridondante e scrittura ritardata, in modo da limitare il numero di accessi fisici al disco[113]. La libreria C comprende alcune funzioni atte all'implementazione di capacità di caching nei programmi: nonostante la massima efficienza sia raggiungibile solo con algoritmi sofisicati[114], vale la pena di citare la

```
int setvbuf(FILE *stream,char *buf,int mode,int size);
```

che consente di associare un buffer di caching ad uno stream. La gestione del buffer è automatica e trasparente al programmatore, che deve unicamente preoccuparsi di chiamare `setvbuf()` dopo avere aperto lo stream con la solita `fopen()`: del resto il primo parametro richiesto da `setvbuf()` è proprio lo stream sul quale operare il caching. Il secondo parametro è il puntatore al buffer: è possibile passare alla funzione l'indirizzo di un'area di memoria precedentemente allocata, la cui dimensione è indicata dal quarto parametro, `size` (il cui massimo valore è limitato a 32767); tuttavia, se il secondo parametro attuale è la costante manifesta `NULL`, `setvbuf()` provvede essa stessa ad allocare un buffer di

[112] Si può affermare, anzi, che i dischi sono tra le periferiche più lente.

[113] Proviamo a chiarire: per lettura ridondante (*read forwarding*) si intende la lettura dal file di una quantità di byte superiore al richiesto e la loro memorizzazione in un buffer, nella "speranza" che eventuali successive operazioni di lettura possano essere limitate alla ricerca dei dati nel buffer medesimo, senza effettuare ulteriori accessi fisici al disco. La scrittura ritardata (*write staging*) consiste nel copiare in un buffer i dati da scrivere di volta in volta, per poi trasferirne sul disco la massima quantità possibile col minimo numero di accessi fisici.

[114] Esistono, del resto, programmi dedicati al caching dei dischi, generalmente operanti a livello di sistema operativo (ad esempio `SMARTDRV.EXE`, fornito come parte integrante del DOS). Inoltre non è raro il caso di dischi incorporanti veri e propri sistemi di caching a livello hardware.

dimensione `size`. Il terzo parametro indica la modalità di gestione del buffer: allo scopo sono definite (in `STDIO.H`) alcune costanti manifeste:

MODALITÀ DI CACHING CON `setvbuf()`

COSTANTE	SIGNIFICATO
`_IOFBF`	Attiva il caching completo del file (*full buffering*): in input, qualora il buffer sia vuoto la successiva operazione di lettura lo riempie (se il file ha dimensione sufficiente); nelle operazioni di output i dati sono scritti nel file solo quando il buffer è pieno.
`_IOLBF`	Attiva il caching a livello di riga (*line buffering*): le operazioni di input sono gestite come nel caso precedente, mentre in output i dati sono scritti nel file (con conseguente svuotamento del buffer) ogni volta che nello stream transita un carattere di fine riga.
`_IONBF`	Disattiva il caching (*no buffering*).

La funzione `setvbuf()` restituisce `0` in assenza di errori, altrimenti è restituito un valore diverso da `0`.

Attenzione al listato che segue:

```
#include <stdio.h>

FILE *fopenWithCache(char *name,char *mode)
{
    FILE *stream;
    char cbuf[1024];

    if(!(stream = fopen(name,mode)))
        return(NULL);
    if(setvbuf(stream,cbuf,_IOFBF,1024)) {
        fclose(stream);
        return(NULL);
    }
    return(stream);
}
```

Dove si nasconde l'errore? L'array `cbuf` è allocato come variabile automatica e, pertanto, cessa di esistere in uscita dalla funzione (vedere pag. 34); tuttavia `fopenWithCache()`, se non si è verificato alcun errore, restituisce il puntatore allo stream aperto, dopo avervi associato proprio cbuf come buffer di caching. E' evidente che tale comportamento è errato, perché forza tutte le operazioni di buffering a svolgersi in un'area di memoria riutilizzata, assai probabilmente, per altri scopi. In casi analoghi a quello descritto, è opportuno utilizzare `malloc()`; meglio ancora è, comunque, lasciare fare a `setvbuf()` (passandole `NULL` quale puntatore al buffer): ciò comporta, tra l'altro, il vantaggio della sua deallocazione automatica al momento della chiusura dello stream.

Per un esempio pratico di utilizzo di `setvbuf()` vedere pag. 587.

ALTRI STRUMENTI DI GESTIONE DEI FILE

Gli stream costituiscono un'implementazione software di alto livello, piuttosto distante dalla reale tecnica di gestione dei file a livello di sistema operativo DOS, il quale, per identificare i file aperti, si serve di proprie strutture interne di dati e, per quanto riguarda l'interfacciamento con i programmi, di descrittori numerici detti *handle*. Questi altro non sono che numeri, ciascuno associato ad un file aperto, che il programma utilizza per effettuare le operazioni di scrittura, lettura e posizionamento. Poiché il DOS non possiede routine di manipolazione diretta degli stream, questi, internamente, sono a loro volta basati sugli handle[115], ma ne nascondono l'esistenza e mettono a disposizione funzionalità aggiuntive, quali la possibilità di gestire input e output formattati nonché oggetti diversi dalla semplice sequenza di byte.

Le librerie standard del C includono funzioni di gestione dei file basate sugli handle, i cui prototipi sono dichiarati in `IO.H`, tra le quali vale la pena di citare:

```
int open(char *path,int operation,unsigned mode);
int _open(char *path,int oflags);
int write(int handle,void *buffer,unsigned len);
int read(int handle,void *buffer,unsigned len);
int lseek(int handle,long offset,int origin);
int close(int handle);
```

L'analogia con `fopen()`, `fwrite()`, `fread()`, `fseek()` e `fclose()` è immediato: in effetti le prime possono essere considerate le omolghe di queste. Non esistono, però, funzioni omologhe di altre molto utili, quali la `fprintf()` e la `fscanf()`.

Non sembra necessario dilungarsi sulla sintassi delle funzioni basate su handle: d'altra parte qualsiasi file può sempre essere manipolato via stream (ed è questa, tra l'altro, l'implementazione grandemente curata e sviluppata dal C++); è forse il caso di commentare brevemente la funzione `open()`.

Il parametro `path` equivale al primo parametro della `fopen()` ed indica il file che si desidera aprire.

Il secondo parametro (`operation`) è un intero che specifica la modalità di apertura del file (ed ha significato analogo al secondo parametro di `fopen()`), il cui valore risulta da un'operazione di or su bit (vedere pag. 72) tra le costanti manifeste elencate di seguito, definite in `FCNTL.H`.

MODALITÀ DI APERTURA DEL FILE CON `open()`: PARTE 1

COSTANTE	SIGNIFICATO
`O_RDONLY`	Apre il file in sola lettura.
`O_WRONLY`	Apre il file in sola scrittura.
`O_RDWR`	Apre il file il lettura e scrittura.

Le tre costanti sopra elencate sono reciprocamente esclusive. Per specificare tutte le caratteristiche desiderate per la modalità di apertura del file, la costante prescelta tra esse può essere posta in or su bit con una o più delle seguenti:

---

[115] Lo dimostra il fatto che uno dei campi della struttura di tipo `FILE` è un intero contenente proprio lo handle associato al file aperto. Detto campo (che nell'implementazione C della Borland ha nome `fd`) può essere utilizzato come handle del file con tutte le funzioni che richiedono proprio lo handle quale parametro in luogo dello stream.

MODALITÀ DI APERTURA DEL FILE CON `open()`: PARTE 2

COSTANTE	SIGNIFICATO
`O_APPEND`	Le operazioni dn scrittura sul file possono esclusivamente aggiungere byte al medesimo (modo append).
`O_CREAT`	Se il file non esiste viene creato e i permessi di accesso al medesimo sono impostati in base al terzo parametro di `open()`, `mode`. Se il file non esiste, `O_CREAT` è ignorata.
`O_TRUNC`	Se il file esiste, la sua lunghezza è troncata a 0.
`O_EXCL`	E' utilizzato solo con `O_CREAT`: se il file esiste, `open()` fallisce e restituisce un errore.
`O_BINARY`	Richiede l'apertura del file in modo binario (è alternativa a `O_TEXT`).
`O_TEXT`	Richiede l'apertura del file in modo testo (è alternativa a `O_BINARY`).

Se né `O_BINARY` né `O_TEXT` sono specificate, il file è aperto nella modalità impostata dalla variabile globale `_fmode`, come del resto avviene con `fopen()` in assenza degli specificatori "`t`" e "`b`".

Il terzo parametro di `open()`, `mode`, è un intero senza segno che può assumere uno dei valori seguenti (le costanti manifeste utilizzate sono definite in `SYS\STAT.H`):

PERMESSI DI ACCESSO AL FILE CON `open()`

COSTANTE	SIGNIFICATO
`S_IWRITE`	Permesso di accesso al file in scrittura.
`S_IREAD`	Permesso di accesso al file in sola lettura.
`S_IREAD|S_IWRITE`	Permesso di accesso al file in lettura e scrittura.

La libreria Borland comprende una variante di `open()` particolarmente adatta alla gestione della condivisione di file nel *networking*[116]: si tratta della

[116] Quando più personal computer sono collegati in rete (*net*), alcuni di essi mettono le proprie risorse (dischi, stampanti, etc.) a disposizione degli altri. Ciò consente di utilizzare in comune, cioè in modo condiviso, periferiche, dati e programmi (che in assenza della rete dovrebbero essere duplicati su ogni macchina interessata). In questi casi è possibile, per non dire probabile, che siano effettuati accessi concorrenti, da più parti, ai file memorizzati sui dischi condivisi tra computer: è evidente che si tratta di situazioni delicate, che possono facilmente causare, se non gestite in modo opportuno, problemi di una certa gravità. Si pensi, tanto per fare un semplice esempio, ad un database condiviso: un primo programma accede in lettura ad un record del medesimo; mentre i dati sono visualizzati, un secondo programma accede al medesimo record per modificarne il contenuto determinando così una palese incongruenza tra il risultato dell'interrogazione della base dati e la reale consistenza della medesima. E' evidente che occorre stabilire regole di accesso alle risorse condivise tali da evitare il rischio di conflitti analoghi a quello descritto.

```
int _open(char *path,int oflags);
```

Il secondo parametro, `oflags`, determina la modalità di apertura ed accesso condiviso al file, secondo il valore risultante da un'operazione di or su bit di alcune costanti manifeste. In particolare, deve essere utilizzata una sola tra le costanti `O_RDONLY`, `O_WRONLY`, `O_RDWR` (proprio come in `open()`); possono poi essere usate, a partire dalla versione 3.0 del DOS, le seguenti:

MODALITÀ DI CONDIVISIONE DEL FILE CON `_open()`

COSTANTE	SIGNIFICATO	DEFINITA IN
`O_NOINHERIT`	Il file non è accessibile ai *child process*.	`FCNTL.H`
`SH_COMPAT`	Il file può essere aperto in condivisione da altre applicazioni solo se anche queste specificano `SH_COMPAT` nella modalità di apertura.	`SHARE.H`
`SH_DENYRW`	Il file non può essere aperto in condivisione da altre applicazioni.	`SHARE.H`
`SH_DENYWR`	Il file può essere aperto in condivisione da altre applicazioni, ma solo per operazioni di lettura.	`SHARE.H`
`SH_DENYRD`	Il file può essere aperto in condivisione da altre applicazioni, ma solo per operazioni di scrittura.	`SHARE.H`
`SH_DENYNO`	Il file può essere aperto in condivisione da altre applicazioni per lettura e scrittura, purché esse non specifichino la modalità `SH_COMPAT`.	`SHARE.H`

Sia `open()` che `_open()` restituiscono un intero positivo che rappresenta lo handle del file (da utilizzare con `write()`, `read()`, `close()`, etc.); in caso di errore è restituito `-1`.

In particolare, la `_open()` sfrutta a fondo le funzionalità offerte dal servizio 3Dh dell'int 21h, descritto a pag. 320.

LANCIARE PROGRAMMI

Si tratta, ovviamente, di lanciare programmi dall'interno di altri programmi. E' una possibilità la cui utilità dipende largamente non solo dagli scopi del programma stesso, ma anche e soprattutto dalle caratteristiche del sistema operativo. E' facile intuire che un sistema in grado di dare supporto all'elaborazione multitasking (il riferimento a Unix, ancora una volta, è voluto e non casuale) offre interessanti possibilità al riguardo (si pensi, ad esempio, ad un gruppo di programmi elaborati contemporaneamente, tutti attivati e controllati da un unico programma gestore); tuttavia, anche in ambienti meno evoluti si può fare ricorso a tale tecnica (per un esempio, anche se non esaustivo, vedere pag. 507).

Per brevità, adottando la terminologia Unix, il programma chiamante si indica d'ora in poi con il termine *parent*, mentre *child* è il programma chiamato.

LA LIBRERIA C

La libreria C fornisce supporto al lancio di programmi esterni mediante un certo numero di funzioni, che possono essere considerate standard entro certi limiti (più precisamente, lo sono in ambiente DOS); per approfondimenti circa la portabilità dei sorgenti che ne fanno uso vedere pag. 136.

Come al solito, per i dettagli relativi alla sintassi, si rimanda alla manualistica specifica del compilatore utilizzato; qui si intende semplicemente mettere in luce alcune interessanti caratteristiche di dette funzioni e i loro possibili ambiti di utilizzo.

system()

La funzione `system()` costituisce forse il mezzo più semplice per raggiungere lo scopo: essa deve infatti essere invocata passandole come unico parametro una stringa contenente, né più né meno, il comando che si intende eseguire. Ad esempio, il codice:

```
#include <stdlib.h>
#include <stdio.h>                                                  // per fprintf()
#include <errno.h>                                                     // per errno
...
    if(system("comando param1 param2") == -1)
        fprintf(stderr,"errore %d in system()\n", errno);
```

lancia il programma `comando` passandogli i parametri `param1` e `param2`. Dal momento che, nell'esempio, per `comando` non è specificato un path, il sistema utilizza la variabile di environment `PATH`[117] (non è necessario specificare l'estensione). Dall'esempio si intuisce che `system()`

[117] Se viene fornito un path, tutte le backslash presenti nella stringa devono essere raddoppiate, onde evitare che il compilatore le interpreti come parte di sequenze di escape. Ad esempio, la stringa

```
"c:\dos\tree"
```

non viene gestita correttamente, mentre

```
"c:\\dos\\tree"
```

funziona come desiderato.

restituisce `-1` in caso di errore; tuttavia va sottolineato che la restituzione di `0` non implica che `comando` sia stato effettivamente eseguito secondo le intenzioni del programmatore. Infatti `system()` esegue il comando ricevuto come parametro attraverso l'interprete dei comandi: in altre parole, essa non fa altro che lanciare una copia dell'interprete stesso e scaricargli il barile, proprio come se fosse stata digitata la riga

```
command -c "comando param1 param2"
```

Ne segue che `system()` si limita a restituire `0` nel caso in cui sia riuscita a lanciare correttamente l'interprete, e non si preoccupa di come questo se la cavi poi con il comando specificato: pertanto, non solo non è possibile conoscere il valore restituito al sistema dal child, ma non è neppure possibile sapere se questo sia stato effettivamente eseguito.

Se, da una parte, ciò appare come un pesante limite, dall'altra la `system()` consente di gestire anche comandi interni DOS, proprio perché in realtà è l'interprete a farsene carico. Ad esempio è possibile richiedere

```
    system("dir /p");
```

e `system()` restituisce `-1` solo se non è stato possibile lanciare l'interprete dei comandi. Inoltre, è possibile eseguire i file batch. Ancora,

```
    system("command");
```

esegue un'istanza dell'interprete, mettendo il prompt del DOS a disposizione dell'utilizzatore: digitando

```
exit
```

al prompt la shell viene chiusa e l'elaborazione del programma parent riprende[118].

Infine, `system()` può essere utilizzata semplicemente per verificare se l'interprete dei comandi è disponibile:

```
    system(NULL);
```

restituisce un valore diverso da `0` se è possibile lanciare l'interprete dei comandi.

E' superfluo (speriamo!) chiarire che l'argomento di `system()` non deve necessariamente essere una costante stringa, come si è assunto per comodità negli esempi precedenti, ma è sufficiente che esso sia di tipo `char *`: ciò consente la costruzione dinamica della riga di comando, ad esempio mediante l'utilizzo di funzioni atte ad operare sulle stringhe[119](`strcpy()`, `strcat()`, `sprintf()`, etc.).

---

[118] A dire il vero anche le altre funzioni (`spawn...()`, `exec...()` possono eseguire un'istanza dell'interprete dei comandi. Bisogna però fare attenzione a non cacciarsi nei guai: una chiamata come

```
    system("command /p");
```

è lecita e viene tranquillamente eseguita, ma l'effetto dell'opzione `/p` su `command.com` è di renderne permanente in memoria la nuova istanza: in tal caso il comando `exit` non ha alcun effetto, e non è più possibile riprendere l'esecuzione del parent. Per ragioni analoghe, l'esecuzione di un programma TSR (vedere pag. 275) attraverso la `system()` potrebbe avere conseguenze distruttive.

[119] Vedere pag. 13 e seguenti.

spawn...()

Come la system(), anche le funzioni della famiglia spawn...() consentono di lanciare programmi esterni come se fossero subroutine del parent; tuttavia esse non fanno ricorso all'interprete dei comandi, in quanto si basano sul servizio 4Bh dell'int 21h[120]: di conseguenza, non è possibile utilizzarle per invocare comandi interni DOS né file batch, tuttavia si ha un controllo più ravvicinato sull'esito dell'operazione. Esse infatti restituiscono -1 se l'esecuzione del child non è riuscita; in caso contrario restituiscono il valore che il programma child ha restituito a sua volta.

Tutte le funzioni spawn...() richiedono come primo parametro un intero, di solito dichiarato nei prototipi con il nome mode, che indica la modalità di esecuzione del programma child: in PROCESS.H sono definite le costanti manifeste P_WAIT (il child è eseguito come una subroutine) e P_OVERLAY (il child è eseguito sostituendo in memoria il parent, proprio come se fosse chiamata la corrispondente funzione della famiglia exec...()). Come osservato riguardo system() (vedere pag. 130), anche le funzioni spawn...() non possono essere utilizzate per lanciare shell permanenti o programmi TSR (vedere pag. 275); tuttavia l'utilizzo del valore P_OVERLAY per il parametro mode consente un'eccezione, in quanto il parent scompare senza lasciare traccia di sé e, in uscita dal child, la sua esecuzione non può mai riprendere.

Il secondo parametro, di tipo char *, è invece il nome del programma da eseguire: esso, diversamente da quanto visto circa la system(), deve essere completo di estensione; inoltre, se non è specificato il path, solo le funzioni spawnlp(), spawnlpe(), spawnvp() e spawnvpe() utilizzano la variabile di environment PATH (la lettera "p" presente nel suffisso finale dei nomi delle funzioni indica proprio detta caratteristica).

Funzioni del gruppo "l"

Le funzioni del gruppo "l" si distinguono grazie alla presenza, nel suffisso finale del loro nome, della lettera "l", la quale indica che gli argomenti della riga di comando del child sono accettati dalla funzione spawnl...() come una lista di parametri, di tipo char *, conclusa da un puntatore nullo.

Ad esempio, per eseguire il comando

```
myutil -a -b 5 arg1 arg2
```

si può utilizzare la funzione spawnl():

```
#include <process.h>
...
    spawnl(P_WAIT,"myutil.exe","myutil","-a","-b","5","arg1","arg2",NULL);
```

Si noti che il nome del programma è passato due volte a spawnl(): la prima stringa indica il programma da eseguire, mentre la seconda rappresenta il primo parametro ricevuto dal programma child: essa deve essere comunque passata alla funzone spawnl...() e, per convenzione, è uguale al nome del programma stesso (il valore di argv[0], se questo è stato a sua volta scritto in linguaggio C: vedere pag. 105 e seguenti). Il programma myutil è ricercato solo nella directory corrente; la funzione spawnlp(), la cui sintassi è identica a quella di spawnl(), effettua la ricerca in tutte le directory specificate dalla variabile di environment PATH.

---

[120] Detto servizio utilizza il valore presente nel registro macchina AL per stabilire il tipo di azione da intraprendere: in particolare, AL = 0 richiede il caricamento e l'esecuzione del programma (funzioni spawn...() se il primo parametro è P_WAIT), mentre AL = 3 richiede il caricamento e l'esecuzione del child nella memoria riservata al parent (overlay), il quale viene a tutti gli effetti sostituito dal nuovo programma (funzioni spawn...() se il primo parametro è P_OVERLAY, e funzioni exec...()).

Il processo child eredita l'ambiente del parent: in altre parole, le variabili di environment del child sono una copia di quelle del programma chiamante. I due environment sono pertanto identici, tuttavia il child non può accedere a quello del parent, né tantomeno modificarlo. Se il parent ha la necessità di passare al child un environment diverso dal proprio, può farlo mediante le funzioni `spawnle()` e `spawnlpe()`, che, pur essendo analoghe alle precedenti, accettano un ulteriore parametro dopo il puntatore nullo che chiude la lista degli argomenti:

```
    static char *newenv[] = {"USER=Pippo","PATH=C:\\DOS",NULL);
    ...
    spawnle(P_WAIT,"myutil","myutil","-a","-b","5","arg1","arg2",NULL,newenv);
```

lancia `myutil` in un environment che comprende le sole[121] variabili `USER` e `PATH`, valorizzate come evidente nella dichiarazione dell'array di stringhe (o meglio, di puntatori a stringa, o, meglio ancora, di puntatori a puntatori a carattere) `newenv`. Il processo parent, qualora abbia necessità di passare al child una copia modificata del proprio environment, deve arrangiarsi a costruirla utilizzando le funzioni di libreria `getenv()` e `putenv()` e la variabile globale `environ`[122], dichiarate in `DOS.H`.

Funzioni del gruppo "v"

Le funzioni del gruppo "`v`" si distinguono grazie alla presenza, nel suffisso finale del loro nome, della lettera "`v`" (in luogo della lettera "`l`"), la quale indica che gli argomenti della riga di comando del child sono accettati dalla funzione `spawnv...()` come un puntatore ad un array di stringhe, il cui ultimo elemento deve essere un puntatore nullo.

Riprendendo l'esempio precedente, il comando

```
myutil -a -b 5 arg1 arg2
```

viene gestito mediante la funzione `spawnv()` come segue:

```
#include <process.h>
...
    char *childArgv[] = {"myutil","-a","-b","5","arg1","arg2",NULL};
    ...
    spawnv(P_WAIT,"myutil.exe",childArgv);
```

Si intuisce facilmente che la `spawnvp()` cerca il comando da eseguire in tutte le directory definite nella variabile di ambiente `PATH` (qualora il suo path non sia specificato esplicitamente), mentre `spawnv()` lo ricerca solo nella directory corrente.

---

[121] E' immediato verificarlo con il codice seguente:

```
    static char *newenv[] = {"USER=Pippo","PATH=C:\\DOS",NULL);
    ...
    spawnlpe(P_WAIT,"command.com","command.com",NULL,newenv);
```

che esegue una istanza dell'interprete dei comandi; digitando il comando `SET` al prompt viene visualizzato l'environment corrente. Il comando `EXIT` consente di chiudere la shell e restituire il controllo al parent.

[122] La variabile `environ` contiene l'indirizzo dell'array di stringhe (ciascuna avente formato `NOME=VAL`, dove `NOME` rappresenta il nome della variabile e `VAL` il suo valore) rappresentanti l'environemnt del programma. Se si utilizza `putenv()` per modificare il valore di una variabile o per inserirne una nuova, il valore di `environ` viene automaticamente aggiornato qualora sia necessario rilocare l'array. Dichiarando `main()` con tre parametri (vedere pag. 105) il terzo rappresenta il puntatore all'array delle stringhe di environment ed inizialmente ha lo stesso valore di `environ`, ma non viene modificato da `putenv()`.

Si noti che il primo elemento dell'array `childArgv[]` punta, per convenzione, al nome del child medesimo (del resto il nome scelto per l'array nell'esempio dovrebbe suggerire che esso viene ricevuto dal child come parametro `argv` di `main()`: vedere pag. 105).

Infine, le funzioni `spawnve()` e `spawnvpe()`, analogamente a `spawnle()` e `spawnlpe()`, accettano come ultimo parametro un puntatore ad un array di stringhe, che costituiranno l'environment del child.

exec...()

Le funzioni della famiglia `exec...()`, a differenza delle `spawn...()`, non trattano il child come una subroutine del parent: esso, al contrario, viene caricato in memoria ed eseguito in luogo del parent, sostituendovisi a tutti gli effetti.

I nomi e la sintassi delle funzioni `exec...()` sono strettamente analoghi a quelli delle `spawn...()`: esistono otto funzioni `exec...()`, ciascuna delle quali può essere posta in corrispondenza biunivoca con una `spawn...()`: a seconda della presenza delle lettere "`l`", "`v`", "`p`" ed "`e`" il comportamento di ciascuna `exec...()` è assolutamente identico a quello della corrispondente `spawn...()` chiamata con il parametro `mode` uguale a `P_OVERLAY` (le funzioni `exec...()` non accettano il parametro `mode`; il loro primo parametro è sempre il nome del programma da eseguire).

Se si desidera che il solito comando degli esempi precedenti sostituisca in memoria il parent e sia eseguito in luogo di questo, è del tutto equivalente utilizzare

```
spawnv(P_OVERLAY,"myutil.exe",childArgv);
```

oppure

```
execv("myutil.exe",childArgv);
```

ad eccezione di quanto specificato in tema di portabilità (pag. 136).

Tabella sinottica

Di seguito si presenta una tabella sinottica delle funzioni `spawn...()` ed `exec...()`.

SINTASSI E CARATTERISTICHE DELLE FUNZIONI `spawn...()` E `exec...()`

	MODO	NOME DEL CHILD	ARGOMENTI DEL CHILD	ENVIRONMENT DEL CHILD

`spawnl()`	`int:`
	`P_WAIT,`
	`P_OVERLAY`
		`char *`	lista di `char *`
			il primo è = child
			l'ultimo è `NULL`	

`spawnlp()`	`int:`
	`P_WAIT,`
	`P_OVERLAY`
		`char *`
		(utilizza `PATH`)
			lista di `char *`
			il primo è = child
			l'ultimo è `NULL`	

`spawnle()`	`int:` `P_WAIT,`	`char *`	lista di `char *` il primo è = child	`char **Env`

	P_OVERLAY		l'ultimo è NULL	
spawnlpe()	int: P_WAIT, P_OVERLAY	char * (utilizza PATH)	lista di char * il primo è = child l'ultimo è NULL	char **Env
spawnv()	int: P_WAIT, P_OVERLAY	char *	char **Argv Argv[0] = child Argv[ultimo] = NULL	
spawnvp()	int: P_WAIT, P_OVERLAY	char * (utilizza PATH)	char **Argv Argv[0] = child Argv[ultimo] = NULL	
spawnve()	int: P_WAIT, P_OVERLAY	char *	char **Argv Argv[0] = child Argv[ultimo] = NULL	char **Env
spawnvpe()	int: P_WAIT, P_OVERLAY	char * (utilizza PATH)	char **Argv Argv[0] = child Argv[ultimo] = NULL	char **Env
execl()		char *	lista di char * il primo è = child l'ultimo è NULL	
execlp()		char * (utilizza PATH)	lista di char * il primo è = child l'ultimo è NULL	
execle()		char *	lista di char * il primo è = child l'ultimo è NULL	char **Env
execlpe()		char * (utilizza PATH)	lista di char * il primo è = child l'ultimo è NULL	char **Env
execv()		char *	char **Argv Argv[0] = child Argv[ultimo] = NULL	
execp()		char * (utilizza PATH)	char **Argv Argv[0] = child Argv[ultimo] = NULL	
execve()		char *	char **Argv Argv[0] = child Argv[ultimo] = NULL	char **Env

execvpe()		char * (utilizza PATH)	char **Argv Argv[0] = child Argv[ultimo] = NULL	char **Env

Condivisione dei file

I processi child lanciati con `spawn...()` e `exec...()` condividono i file aperti dal parent. In altre parole, entrambi i processi possono accedere ai file aperti dal parent, per ciascuno dei quali il sistema operativo mantiene un unico puntatore: ciò significa che le operazioni effettuate da uno dei processi (spostamento lungo il file, lettura, scrittura) influenzano l'altro processo; tuttavia se il child chiude il file, questo rimane aperto per il parent. Vediamo un esempio:

Il seguente frammento di codice, che si ipotizza appartenere al parent, apre il file `C:\AUTOEXEC.BAT`, effettua un'operazione di lettura e lancia il child, passandogli il puntatore allo stream (vedere pag. 116).

```
...
#define   MAX    128
...
    char sPrtStr[10], line[MAX];
    FILE *inP;
    ...
    inP = fopen("C:\\AUTOEXEC.BAT","r");
    printf(fgets(line,MAX,inP));
    sprintf(sPtrStr,"%p",inP);
    spawnl(P_WAIT,"child","child",sPtrStr,NULL);
    printf(fgets(line,MAX,inP));
    ...
```

Se si eccettua la mancanza del pur necessario codice per la gestione degli eventuali errori, tralasciato per brevità, il listato appare piuttosto banale: l'unica particolarità è rappresentata dalla chiamata alla funzione `sprintf()`, con la quale si converte in stringa il valore contenuto nella variabile `inP` (l'indirizzo della struttura che descrive lo stream aperto dalla `fopen()`). Come si può vedere, il parent passa al child proprio detta stringa (è noto che i parametri ricevuti da un programma sulla riga di comando sono necessariamente stringhe), alla quale esso può accedere attraverso il proprio `argv[1]`. Ecco un frammento del child:

```
...
#define   MAX    128
...
int main(int argc,char **argv)
{
    ...
    FILE *inC;
    ...
    sscanf(argv[1],"%p",&inC);
    printf(fgets(line,MAX,inC));
    fclose(inC);
    ....
}
```

Il child memorizza in `inC` l'indirizzo della struttura che descrive lo stream aperto dal parent ricavandolo da `argv[1]` mediante la `sscanf()`, effettua un'operazione di lettura e chiude lo stream; tuttavia, il parent è ancora in grado di effettuare operazioni di lettura dopo il rientro dalla `spawnl()`:

l'effetto congiunto dei due programmi consiste nel visualizzare le prime tre righe del file `C:\AUTOEXEC.BAT`.

Va sottolineato che è necessario compilare entrambi i programmi per un modello di memoria che gestisca i dati con puntatori a 32 bit (medium, large, huge: vedere pag. 143 e seguenti): è infatti molto verosimile (per non dire scontato) che il child non condivida il segmento dati del parent, nel quale è allocata la struttura associata allo stream: l'utilizzo di indirizzi a 16 bit, che esprimono esclusivamente offset rispetto all'indirizzo del segmento dati stesso, condurrebbe inevitabilmente il child a utilizzare quel medesimo offset rispetto al proprio data segment, accedendo così ad una locazione di memoria ben diversa da quella desiderata.

PORTABILITÀ

Date le differenti caratteristiche del supporto fornito dai diversi sistemi operativi (DOS e Unix in particolare), sono necessarie alcune precisazioni relative alla portabilità del codice tra i due ambienti.

La funzione `system()` può essere considerata portabile: essa è infatti implementata nelle librerie standard dei compilatori in entrambi i sistemi.

Analoghe considerazioni valgono per le funzioni `exec...()`, ma con prudenza: in ambiente Unix, solitamente, non sono implementate le funzioni `execlpe()` e `execvpe()`. Inoltre, le funzioni `execlp()` e `execvp()` in versione Unix sono in grado di eseguire anche shell script (analoghi ai file batch del DOS). Tutte le funzioni `exec...()` in Unix, infine, accettano come nome del child il nome di un file ASCII che a sua volta, con una particolare sintassi, specifica qual è il programma da eseguire (ed eseguono quest'ultimo).

Le funzioni `spawn...()` non sono implementate in ambiente Unix. La modalità di gestione dei child, in questo caso, si differenzia profondamente proprio perché Unix è in grado di eseguire più processi contemporaneamente: pertanto un child non è necessariamente una subroutine del parent; i due programmi possono essere eseguiti in parallelo. Un modo per emulare le `spawn...()` consiste nell'uso congiunto delle funzioni `fork()` (assente nelle librerie C in DOS) ed `exec...()`: la `fork()` crea una seconda istanza del parent; di conseguenza, essa fa sì che coesistano in memoria due processi identici, l'esecuzione di entrambi i quali riprende in uscita dalla `fork()` stessa.. Dall'esame del valore restituito dalla `fork()` è possibile distinguere l'istanza parent dall'istanza child, in quanto `fork()` restituisce `0` al child, mentre al parent restituisce il PID[123] del child stesso. L'istanza child può, a questo punto, utilizzare una delle `exec...()` per eseguire il programma desiderato, mentre il parent, tramite la funzione `waitpid()` (anch'essa non implementata nel C in DOS) può attendere la terminazione del child e esaminarne il valore restituito mediante la macro `WEXITSTATUS()`. A puro titolo di esempio si riporta di seguito un programma, compilabile in ambiente Unix, che utilizza la tecnica descritta.

```
#include <stdio.h>                          /* printf(), puts(), fprintf(), stderr */
#include <unistd.h>                                     /* fork(), execlp(), pid_t */
#include <errno.h>                                                      /* errno */
#include <sys/wait.h>                                /* waitpid(), WEXITSTATUS() */

int main(void);
void child(void);
void parent(pid_t pid);

int main(void)
{
    pid_t pid;
```

[123] Process IDentifier. E' un intero positivo che identifica univocamente un processo tra tutti quelli in esecuzione in un dato momento. Si tratta di un concetto sconosciuto in DOS.

```
    puts("Il child elenchera' i files presenti nella directory /etc.");
    switch(pid = fork()) {
        case 0:
            child();
        case -1:
            fprintf(stderr,"Errore %d in fork().\n",errno);
            exit(errno);
        default:
            parent(pid);
    }
    return(0);
}

void child(void)
{
    if(execlp("ls","ls","-la","/etc",NULL) == -1) {
        fprintf(stderr,"Errore %d in execlp().\n",errno);
        exit(errno);
    }
}

void parent(int pid)
{
    int status;

    if(waitpid(pid,&status,0) <= 0) {
        printf("Errore %d in waitpid().\n");
        exit(errno);
    }
    printf("Il child ha restituito %d.\n",WEXITSTATUS(status));
}
```

In uscita dalla `fork()` entrambe le istanze del programma effettuano il test sul valore da questa restituito, e solo in base al risultato del test medesimo esse si differenziano, eseguendo `parent()` oppure `child()`. E' ovvio che l'istanza child non deve necessariamente eseguire una `exec...()` e annullarsi: essa può eseguire qualunque tipo di operazione (comprese ulteriori chiamate a `fork()`), come del resto l'istanza parent non ha l'obbligo di attendere la terminazione del child, ma, al contrario, può eseguire altre elaborazioni in parallelo a quello e verificarne lo stato solo in un secondo tempo.

La libreria C in ambiente Unix implementa altre funzioni (assenti sotto DOS) per il controllo dei processi child: ad esempio la `popen()`, che, con una sintassi del tutto analoga alla `fopen()` (vedere pag. 116 e seguenti), consente di lanciare un programma e al tempo stesso rende disponibile uno stream di comunicazione, detto *pipe*, mediante il quale il parent può leggere dallo standard output o scrivere sullo standard input del child. Ancora, la `pipe()` apre una pipe (questa volta non collegata a standard input e standard output) che può essere utilizzata come un file virtuale in condivisione tra processi parent e child.

Come strumento di comunicazione inter-process, in DOS si può ricorrere alla condivisione dei file, come descritto a pag. 135. Trattandosi di file reali, il metodo è certo meno efficiente della pipe, ma ha il vantaggio di risultare portabile tra i due sistemi. Per utilizzare in DOS aree di memoria in condivisione (tecnica in qualche modo paragonabile alla shared memory supportata da Unix) si può ricorrere, rinunciando alla portabilità, allo stratagemma illustrato a pag. 550.

Per approfondimenti circa le problematiche di portabilità dipendenti dai sistemi operativi si veda pag. 465.

GLI INTERRUPT: UTILIZZO

Gli interrupt sono routine, normalmente operanti a livello di ROM-BIOS o DOS, in grado di svolgere compiti a "basso livello", cioè a stretto contatto con lo hardware. Esse evitano al programmatore la fatica di riscrivere per ogni programma il codice (necessariamente in assembler) per accedere ai dischi o al video, per inviare caratteri alla stampante, e così via. Le routine di interrupt, inoltre, rendono i programmi indipendenti (almeno in larga parte) dallo hardware e dal sistema operativo; si può pensare ad esse come ad una libreria alla quale il programma accede per svolgere alcune particolari attività. Tutto ciò nei linguaggi di alto livello avviene in modo trasparente: è infatti il compilatore che si occupa di generare le opportune chiamate ad interrupt in corrispondenza delle istruzioni peculiari di quel linguaggio. Nei linguaggi di basso livello (assembler in particolare) esistono istruzioni specifiche per invocare gli interrupt: è proprio in questi casi che il programmatore ne può sfruttare al massimo le potenzialità e utilizzarli in modo consapevole proprio come una libreria di routine. Il C mette a disposizione diverse funzioni che consentono l'accesso diretto[124] agli interrupt: cerchiamo di approfondire un poco[125].

ROM-BIOS E DOS, HARDWARE E SOFTWARE

Le routine di interrupt sono dette ROM-BIOS quando il loro codice fa parte, appunto, del BIOS della macchina; sono dette, invece, DOS, se implementate nel sistema operativo. Gli interrupt BIOS possono poi essere suddivisi, a loro volta, in due gruppi: hardware, se progettati per essere invocati da un evento hardware[126], esterno al programma; software, se progettati per essere esplicitamente chiamati da programma[127], mediante un'apposita istruzione (`INT` per l'assembler). Gli interrupt DOS sono tutti software, e rappresentano spesso una controparte, di livello superiore[128], delle routine BIOS, parte delle quali costituisce il gruppo degli interrupt hardware. Si comprende facilmente che si tratta di caratteristiche specifiche dell'ambiente DOS su personal computer con processore Intel: un concetto di interrupt analogo a quello DOS è sconosciuto, ad esempio, in Unix.

Le funzioni della libreria C consentono l'accesso esclusivamente agli interrupt software: del resto, in base alla definizione appena data di interrupt hardware, non sarebbe pensabile attivare questi ultimi come subroutine di un programma

---

[124] Per accesso diretto si intende la possibilità di effettuare una chiamata ad interrupt. In tal senso vi è differenza con le funzioni di libreria che usano internamente gli interrupt per svolgere il loro lavoro.

[125] Per indicazioni circa i metodi e gli artifici utilizzabili per scrivere funzioni in grado di sostituirsi esse stesse agli interrupt (*interrupt handler*), vedere pag. 251 e seguenti.

[126] Ad esempio: la pressione ed il rilascio di un tasto generano una chiamata all'int 09h.

[127] Ad esempio: l'int 13h, che gestisce i servizi di basso livello dedicati ai dischi (formattazione, lettura o scrittura di settori, etc.).

[128] Ad esempio: gli int 25h e 26h leggono e, rispettivamente, scrivono settori dei dischi, ma con criteri meno legati alle caratteristiche hardware della macchina rispetto all'int 13h. Per essere espliciti: il BIOS individua un settore mediante numero di testina (lato), numero di cilindro (traccia) e posizione del settore nella traccia; il DOS invece numera progressivamente i settori del disco a partire dal boot sector (settore 0), ma non è in grado, contrariamente al BIOS, di accedere alla traccia (presente solo negli hard disk) che precede il boot sector e contiene la tavola delle partizioni.

LA LIBRERIA C

Gli interrupt si interfacciano al sistema mediante i registri della CPU. Il concetto è leggermente diverso da quello dei parametri di funzione, perché i registri possono essere considerati variabili globali a tutti i software attivi sulla macchina (in effetti, anche per tale motivo, le routine di interrupt non sono rientranti[129]: vedere pag. 295 e dintorni). Scopo delle funzioni è facilitare il passaggio dei dati mediante i registri della CPU e il recupero dei valori in essi restituiti (un interrupt può restituire più valori semplicemente modificando il contenuto dei registri stessi).

Vi è un gruppo di funzioni di libreria che consente l'utilizzo di qualsiasi interrupt: di esso fanno parte, ad esempio, la `int86()` e la `int86x()`. Vediamo subito un esempio di utilizzo della seconda: la lettura di un settore di disco via int 13h (BIOS).

INT 13H, SERV. 02H: LEGGE SETTORI IN UN BUFFER

Input	AH	02h
	AL	numero di settori da leggere
	CH	numero della traccia di partenza (10 bit[130])
	CL	numero del settore di partenza
	DH	numero della testina (cioè del lato)
	DL	numero del drive (0 = A:)
	ES:BX	indirizzo (seg:off) del buffer in cui vengono memorizzati i settori letti

```
#include <dos.h>                        // prototipo di int86x() e variabile _doserrno
#include <stdio.h>                                                  // prototipo printf()

    ....
    struct SREGS segRegs;
    union REGS inRegs, outRegs;
    char buffer[512];
    int interruptAX;

    segread(&segRegs);
    segRegs.es = segRegs.ss;                                        // segmento di buffer
    inRegs.x.bx = (unsigned)buffer;                                   // offset di buffer
    inRegs.h.ah = 2;                                              // BIOS function number
    inRegs.h.al = 1;                                              // # of sectors to read
    inRegs.h.ch = 0;                                            // track # of boot sector
    inRegs.h.cl = 1;                                           // sector # of boot sector
    inRegs.h.dh = 0;                                                  // disk side number
    inRegs.h.dl = 0;                                                 // drive number = A:
    interruptAX = int86x(0x13, &inRegs, &outRegs, &segRegs);
```

[129] Non possono essere usate in modo ricorsivo.

[130] La traccia di partenza è indicata mediante un numero esadecimale a 10 bit. Dal momento che CH è un registro a 8 bit, i bit 7 e 8 di CL ne rappresentano i 2 bit più significativi. In questo caso essi sono entrambi zero, pertanto il numero della traccia di partenza è deducibile dal solo valore in CH.

```
    if(outRegs.x.cflag)
        printf("Errore n. %d\n",_doserrno);
    ....
```

Procediamo con calma. La `int86x()` richiede 4 parametri: un `int` che esprime il numero dell'interrupt da chiamare, due puntatori a `union` tipo `REGS` e un puntatore a `struct` di tipo `SREGS`. La `union REGS` rende disponibili campi che vengono utilizzati dalla `int86x()` per caricare i registri della CPU o memorizzare i valori in essi contenuti. In pratica essa consente di accedere a due strutture, indicate con `x` e con `h`: i campi della prima sono interi che corrispondono ai registri macchina a 16 bit, mentre quelli della seconda sono tutti di tipo `unsigned char` e corrispondono alla parte alta e bassa di ogni registro[131]. Tramite la `x` sono disponibili i campi `ax`, `bx`, `cx`, `dx`, `si`, `di`, `cflag`, `flags` (i campi `cflags` e `flags` corrispondono, rispettivamente, al Carry Flag e al registro dei Flag); tramite la `h` sono disponibili i campi `al`, `ah`, `bl`, `bh`, `cl`, `ch`, `dl`, `dh`. Caricare valori nei campi di una `union REGS` non significa assolutamente caricarli direttamente nei registri: a ciò provvede la `int86x()`, prelevandoli dalla `union` il cui indirizzo le è passato come secondo parametro, prima di chiamare l'interrupt.

L'esempio chiama l'int 13h per leggere un settore del disco: il numero del servizio dell'interrupt (`2` = lettura di settori) deve essere caricato in `AH`: perciò

```
inRegs.h.ah = 2;
```

Con tecnica analoga si provvede al caricamento di tutti i campi come necessario. Dopo la chiamata all'interrupt, la `int86x()` provvede a copiare nei campi dell'apposita `union REGS` (il cui puntatore è il terzo parametro della funzione) i valori che quello restituisce nei registri. Nell'esempio sono dichiarate due union, perché sia possibile conservare sia i valori in ingresso che quelli in uscita; è ovvio che alla `int86x()` può essere passato il puntatore ad una medesima `union` sia come secondo che come terzo parametro, ma va tenuto presente che in questo caso i valori dei registri di ritorno dall'interrupt sono sovrascritti, negli omologhi campi della struttura, a quelli in entrata, che vengono persi.

E veniamo al resto... Il servizio 2 dell'int 13h memorizza i settori letti dal disco in un buffer il cui indirizzo deve essere caricato nella coppia `ES:BX`, ma la `union REGS` non dispone di campi corrispondenti ai registri di segmento `ES`, `CS`, `SS` e `DS`. Occorre perciò servirsi di una `struct SREGS`, che contiene, appunto, i campi `es`, `cs`, `ss` e `ds` (`unsigned int`). La funzione `segread()` copia nei campi della `struct SREGS` il cui indirizzo riceve come parametro i valori presenti nei registri di segmento al momento della chiamata.

Tornando al nostro esempio, se ipotizziamo di compilarlo per lo small memory model (pag. 143 e seguenti), `buffer` è un puntatore `near`: occorre ricavare comunque la parte segmento per caricare correttamente l'indirizzo a 32 bit in `ES:BX`. Più semplice di quanto sembri: `buffer` è una variabile locale, e pertanto è allocata nello stack. La parte segmento del suo indirizzo a 32 bit è perciò, senz'altro, `SS`[132]; ciò spiega l'assegnazione

```
    segRegs.es = segRegs.ss;
```

---

[131] Ad esempio, il registro a 16 bit `AX` può essere considerato suddiviso in due metà di 8 bit ciascuna: `AH` (la parte alta, cioè gli 8 bit più significativi) e `AL` (la parte bassa). Così `BX` è accessibile come `BH` e `BL`, `CX` come `CH` e `CL`, `DX` come `DH` e `DL`. Gli altri registri sono accessibili solamente come word di 16 bit.

[132] Si noti che questa regola vale in tutti i modelli di memoria; tuttavia se `buffer` fosse un puntatore `far` (perché dichiarato tale o a causa del modello di memoria) sarebbe più semplice ricavarne la parte segmento e la parte offset con le macro `FP_SEG()` e `FP_OFF()`, definite in `DOS.H`. In effetti, dette macro possono essere utilizzate anche con puntatori a 16 bit, purché siano effettuate le necessarie operazioni di cast:

```
    segRegs.es = FP_SEG((void far *)buffer);
    segRegs.bx = FP_OFF((void far *)buffer);
```

Sappiamo che il nome di un array è puntatore all'array stesso e che un puntatore `near` esprime in realtà un offset, pertanto per caricare in `inRegs.x.bx` la parte offset dell'indirizzo di `buffer` è sufficiente la semplice assegnazione che compare nell'esempio: il cast ha lo scopo di evitare un messaggio di warning, perché il campo `bx` è dichiarato come intero e non come puntatore.

L'indirizzo della `struct SREGS` è il quarto parametro passato a `int86x()`: i campi di `segRegs` sono utilizzati, come prevedibile, per inizializzare correttamente i registri di segmento prima di chiamare l'interrupt.

La `int86x()` restituisce il valore assunto da `AX` al rientro dall'interrupt. Inoltre, se il campo `outRegs.x.cflag` è diverso da `0`, l'interrupt ha restituito una condizione di errore e la variabile globale `_doserrno` (vedere pag. 499) ne contiene il codice numerico.

Non tutti gli interrupt richiedono in ingresso valori particolari nei registri di segmento: in tali casi è possibile validamente utilizzare la `int86()`, analoga alla `int86x()`, ma priva del quarto parametro (l'indirizzo della `struct SREGS`), evitando chiamate a `segread()` e strane macchinazioni circa il significato dei puntatori.

Vi è poi la `intr()`, che accetta come parametri: un intero, esprimente il numero dell'interrupt da chiamare, e un puntatore a `struct REGPACK`; questa contiene 10 campi, tutti `unsigned int`, ciascuno dei quali rappresenta una registro a 16 bit: `r_ax`, `r_bx`, `r_cx`, `r_dx`, `r_bp`, `r_si`, `r_di`, `r_ds`, `r_es`, `r_flags`. I valori contenuti nei campi della `struct REGPACK` sono copiati nei registri corrispondenti prima della chiamata ad interrupt, mentre al ritorno è eseguita l'operazione inversa. La `intr()` non restituisce nulla (è dichiarata `void`): lo stato dell'operazione può essere conosciuto analizzando direttamente i valori contenuti nei campi della struttura (è evidente che i valori in ingresso sono persi). Per un esempio di utilizzo della `intr()` vedere pag. 202 e seguenti.

Il secondo gruppo include funzioni specifiche per l'interfacciamento con le routine dell'int 21h[133]: due di esse, `intdosx()` e `intdos()`, sono analoghe a `int86x()` e `int86()` rispettivamente, ma non richiedono il numero dell'interrupt come parametro, in quanto questo è sempre 21h. Alla `intdosx()` è quindi necessario passare due puntatori a `union REGS` e uno a `struct SREGS`, mentre la `intdos()` richiede solamente i due puntatori a `union REGS`.

Le rimanenti due funzioni che consentono di chiamare direttamente l'int 21h sono `bdos()` e `bdosptr()`. La prima richiede che le siano passati, nell'ordine: un intero esprimente il numero del servizio richiesto all'int 21h, un intero il cui valore viene caricato in `DX` prima della chiamata e un terzo intero i cui 8 bit meno significativi sono caricati in `AL` (in pratica come terzo parametro si può utilizzare un `unsigned char`).

Nella `bdosptr()` il secondo parametro è un puntatore (nel prototipo è dichiarato `void *`, perciò può puntare a qualsiasi tipo di dato). Va sottolineato che se il programma è compilato con modello di memoria tiny, small o medium detto puntatore è a 16 bit e il suo valore è caricato in `DX` prima della chiamata all'interrupt; con i modelli compact, large e huge, invece, esso è un puntatore a 32 bit e viene utilizzato per inizializzare la coppia `DS:DX`.

La scelta della funzione da utilizzare di volta in volta, tra tutte quelle presentate, dipende essenzialmente dalle caratteristiche dell'interrupt che si intende chiamare; va tuattavia osservato che la `int86x()` è l'unica funzione che consenta di chiamare qualsiasi interrupt DOS o BIOS, senza limitazioni di sorta[134].

---

[133] L'int 21h rende disponibile la quasi totalità dei servizi DOS: gestione files, I/O, etc..

[134] Infatti se l'interrupt non richiede il caricamento di particolari valori nei registri di segmento, è sufficiente inizializzare una `struct SREGS` mediante una chiamata a `segread()` e passarne l'indirizzo alla `int86x()` senza effettuare altre operazioni.

I MODELLI DI MEMORIA

L'architettura hardware dei processori Intel 80x86, ed in particolare i registri a 16 bit, implementati anche dai processori 80386 e superiori per compatibilità con quelli di categoria inferiore, impongono che la memoria sia gestita in modo segmentato, esprimendo, cioè, un indirizzo a 20 bit mediante 2 registri a 16 bit, detti registro di segmento e registro di offset (vedere, per i particolari, pag. 16). Secondo tale schema sono indirizzati il codice eseguibile (`CS:IP`), lo stack (`SS:SP` o `SS:BP`) e i dati (`DS` o `ES` per esprimere il segmento; l'offset può essere contenuto in diversi registri, tra cui `BX`, `DX`, `SI` e `DI`). L'inizializzazione dei registri della CPU al fine di una corretta esecuzione dei programmi è effettuata dal DOS quando il programma è caricato in memoria per essere eseguito, con regole differenti per i file `.EXE` e `.COM`.

Questi ultimi sono eseguiti sempre a partire dal primo byte del file e la loro dimensione non può superare i 64 Kb, all'interno dei quali, peraltro, devono trovare posto anche il PSP, i dati e lo stack[135]. Ne segue che un programma `.COM` occupa un solo segmento di memoria, e quindi tutti i registri di segmento assumono identico valore. Dette limitazioni[136] sono superate dal formato `.EXE`, di successiva introduzione, che, grazie ad una tabella posta all'inizio del file (la relocation table) sono in grado di dare istruzioni al DOS circa l'inizializzazione dei registri e quindi, in definitiva, sul modo di gestire gli indirizzamenti di codice, stack e dati[137].

La notevole flessibiltà strutturale consentita dalla tipologia `.EXE` può essere sfruttata al meglio dichiarando in modo opportuno puntatori e funzioni[138], in modo da lavorare con indirizzi a 16 o 32 bit, a seconda delle necessità. I compilatori C (o almeno la maggior parte di essi) sono in grado, se richiesto tramite apposite opzioni di compilazione, di generare programmi strutturati secondo differenti default di indirizzamento della memoria, "mescolando" secondo diverse modalità gli indirizzamenti a 32 e a 16 bit per codice, dati e stack: ciascuna modalità rappresenta un modello di memoria, cioè un modello standard di indirizzamento della RAM, che viene solitamente individuato dal programmatore in base alle caratteristiche desiderate per il programma.

Date le differenti modalità di gestione degli indrizzi di codice e dati implementate nei diversi modelli di memoria, a ciascuno di questi corrisponde una specifica versione di libreria di funzioni; in altre parole, ogni compilatore è accompagnato da una versione di libreria per ogni modello di memoria supportato. Fa eccezione soltanto il modello tiny, che utilizza la libreria del modello small: in entrambi i modelli, infatti, la gestione degli indirizzamenti è implementata mediante puntatori `near` tanto per il codice, quanto per i dati[139]. Ne segue che la realizzazione di una libreria di funzioni implica la

---

[135] Inoltre, il DOS copia nello stack dei `.COM` l'indirizzo di ritorno (li chiama "quasi come" se fossero un funzione C); perciò un programma `.COM` può terminare con una `RET`, mentre un `.EXE` deve sempre ricorrere all'int 21h.

[136] Il formato di programma descritto (`.COM`) è il primo nato sui personal computer e deriva le proprie caratteristiche da quelle degli eseguibili per macchine con 64 Kb di RAM totali.

[137] In realtà la relocation table contiene anche una serie di parametri tra cui quelli tramite i quali il DOS può modificare (cioè rilocare) gli indirizzi delle chiamate `far` a funzioni contenute dell'eseguibile, in relazione all'indirizzo al quale il file è caricato per poter essere eseguito. Tutto ciò non è però fondamentale ai fini della discussione dei modelli di memoria di un compilatore C.

[138] I puntatori, lo ripetiamo, possono essere `near`, `far` o `huge`; le funzioni `near` o `far`.

[139] La differenza tra modello tiny e modello small è implementata nei rispettivi moduli di startup (vedere pag. 105).

costruzione di più file `.LIB` e quindi la compilazione dei sorgenti e l'inserimento dei moduli oggetto nella libreria devono essere effettuate separatamente per ogni modello di memoria: ciò non è richiesto solo per i modelli tiny e small, che possono condividere un'unica libreria (vedere pag. 149 e seguenti).

Di seguito descriviamo brevemente i modelli di memoria generalmente supportati dai compilatori.

TINY MODEL

E' il modello che consente la creazione di file `.COM` (oltre ai `.EXE`). Tutti i registri di segmento (`CS`, `SS`, `DS` ed `ES`) contengono lo stesso indirizzo, quello del Program Segment Prefix (pag. 324) del programma. Quando il programma è un `.COM`, il registro `IP` è sempre inizializzato a 100h (256 decimale) e, dal momento che il PSP occupa proprio 256 byte, l'entry point del programma coincide col primo byte del file: i conti tornano.

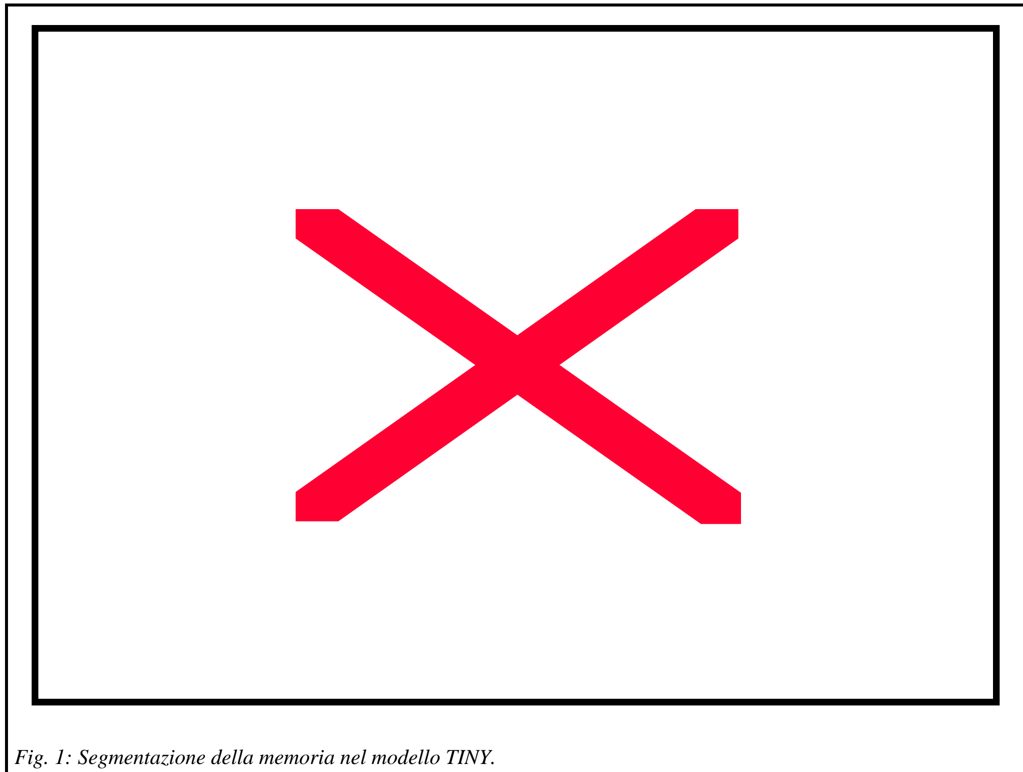

Fig. 1: Segmentazione della memoria nel modello TINY.

Tanto nei file `.COM` che nei file `.EXE`, codice, dati e stack non possono superare i 64 Kb e tutti i puntatori sono, per default, `near`. La memoria è dunque gestita secondo una "mappa" analoga a quella presentata nella figura 1.

Chi non ricordasse che cosa è lo heap e in che cosa si differenzia dallo stack rilegga pagina 111. Qui vale la pena di sottolineare che dati globali e statici, stack e heap condividono il medesimo segmento di memoria: un utilizzo "pesante" dell'allocazione dinamica della memoria riduce quindi lo spazio disponibile per le variabili locali e per i dati globali, e viceversa.

L'opzione del compilatore Borland che richiede la generazione del modello tiny è `-mt`; se sulla riga di comando del compilatore è presente anche l'opzione `-lt` viene prodotto un file `.COM`.

SMALL MODEL

Nel modello small il segmento del codice è separato da quello per i dati. I programmi generati con l'opzione `-ms` (del compilatore Borland, per il quale essa è il default) possono avere fino a 64 Kb di codice eseguibile, ed altri 64 Kb condivisi tra dati statici e globali, heap e stack.

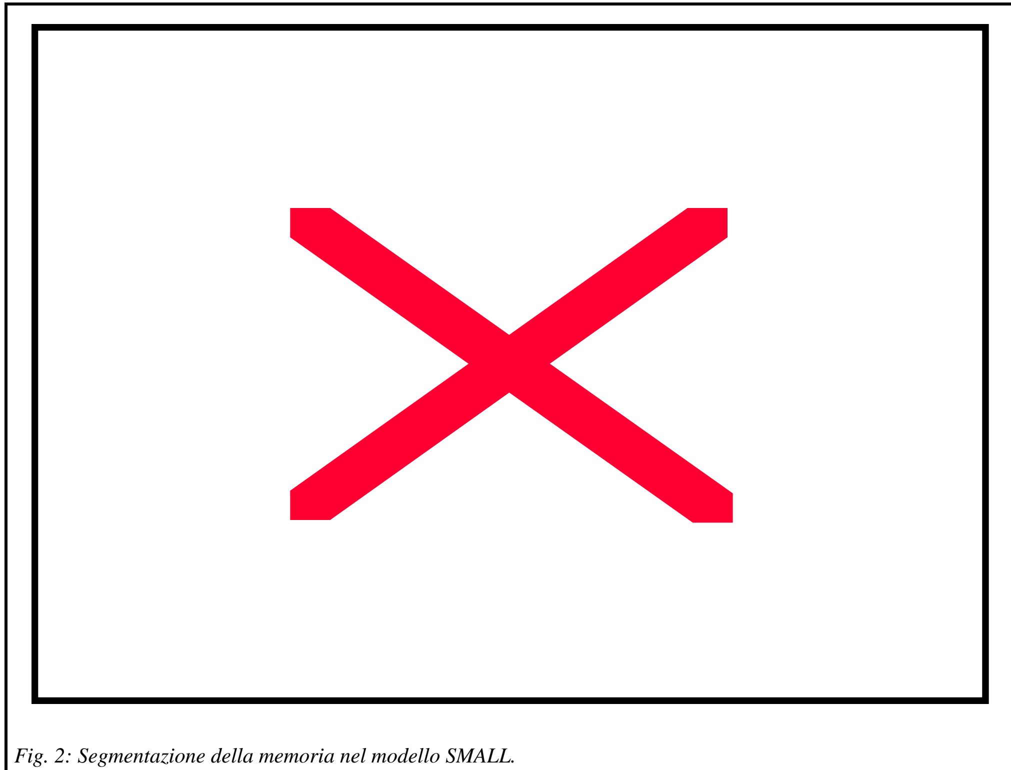

Fig. 2: Segmentazione della memoria nel modello SMALL.

Come si vede dalla figura 2, anche nei programmi compilati in modalità small lo spazio utilizzato dai dati globali riduce heap e stack, e viceversa, ma il valore iniziale di `DS` ed `SS` non coincide con quello di `CS`, in quanto viene stabilito in base ai parametri presenti nella relocation table, generata dal linker. E' inoltre disponibile il far heap, nel quale è possibile allocare memoria da gestire mediante puntatori `far`.

MEDIUM MODEL

Il modello medium è adatto ai programmi di grosse dimensioni che gestiscono piccole quantità di dati: infatti, i puntatori per il codice sono tutti a 32 bit (le chiamate a funzione sono tutte `far`), mentre i puntatori per i dati, per default, sono a 16 bit come nel modello small. Analogamente a quest'utlimo, perciò, il modello medium gestisce un segmento di 64 Kb per dati statici e globali, heap e stack separato dagli indirizzi del codice, che può invece raggiungere la dimensione (teorica) di 1 Mb.

Si noti che il codice eseguibile, qualora superi la dimensione di 64 Kb, deve essere "spezzato" in più moduli `.OBJ`, ognuno dei quali deve essere di dimensioni non superiori ai 64 Kb. La generazione di più moduli oggetto presuppone che il sorgente sia suddiviso in più file, ma è appena il caso di rimarcare che la dimensione di ogni singolo sorgente non ha alcuna importanza: i limiti accennati valgono per il codice già compilato. La figura 3 evidenzia che il registro `CS` è inizializzato per puntare ad uno dei moduli oggetto.

L'opzione del compilatore Borland che richiede la generazione del modello medium è `-mm`.

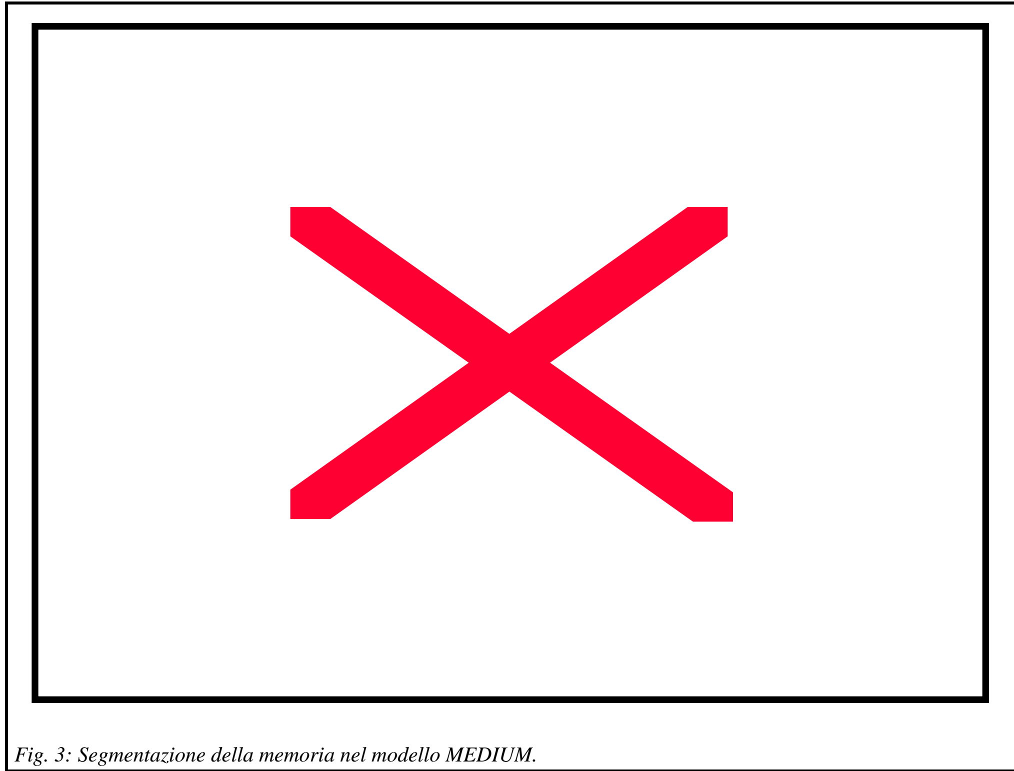

Fig. 3: Segmentazione della memoria nel modello MEDIUM.

Nel modello medium, le funzioni dichiarate esplicitamente `near` sono richiamabili solo dall'interno dello stesso modulo oggetto nel quale esse sono definite, in quanto una chiamata `near`, gestita con un indirizzo a soli 16 bit, non può gestire "salti" inter-segmento. L'effetto è analogo a quello che si ottiene dichiarando `static` una funzione, con la differenza che in questo caso la chiamata è ancora `far`, secondo il default del modello. Una dichiarazione `near` trae dunque motivazione da sottili considerazioni di efficienza, mentre una `static` può rispondere esclusivamente a scelte di limitazione logica di visibilità.

COMPACT MODEL

Il modello compact può essere considerato il complementare del modello medium, in quanto genera per default chiamate `near` per le funzioni e indirizzamenti `far` per i dati: in pratica esso si addice a programmi piccoli, che gestiscono grandi moli di dati. Il codice non può superare i 64 Kb, come nel modello small, mentre per i dati può essere utilizzato fino ad 1 Mb (tale limite è teorico, in quanto ogni programma, in ambiente DOS, si scontra con l'infame "barriera" dei 640 Kb).

L'opzione (compilatore Borland) che richiede la generazione del programma secondo il modello compact è `-mc`.

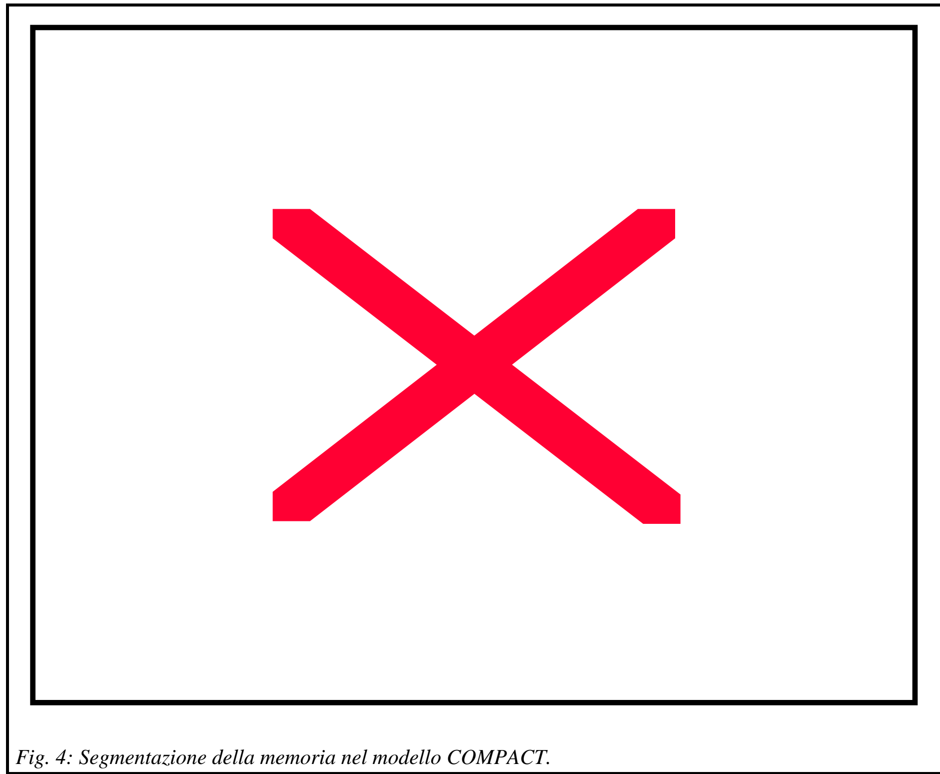

Fig. 4: Segmentazione della memoria nel modello COMPACT.

La figura 4 evidenzia che, a differenza di quanto avviene nei modelli tiny, small e medium, `DS` e `SS` sono inizializzati con valori differenti: il programma ha perciò un segmento di 64 Kb dedicato ai dati statici e globali, ed un altro, distinto, per la gestione dello stack. Lo heap (cioè l'area di RAM allocabile dinamicamente) occupa tutta la rimanente memoria disponibile ed è indirizzato per default con puntatori `far`. Proprio per questa caratteristica esso è definito *heap* e non *far heap*, come avviene invece nel modello small, nel quale è necessario dichiarare esplicitamente `far` i puntatori al far heap e si deve utilizzare `farmalloc()` per allocarvi memoria (vedere pag. 113).

LARGE MODEL

Il modello large genera per default indirizzamenti `far` sia al codice che ai dati e si rivela perciò adatto a programmi di notevoli dimensioni che gestiscono grandi quantità di dati. Esso è, in pratica, un ibrido tra i modelli medium (per quanto riguarda la gestione del codice) e compact (per l'indirizzamento dei dati); codice e dati hanno quindi entrambi a disposizione (in teoria) 1 Mb.

L'opzione (compilatore Borland) per la generazione del modello large è `-ml`.

Il modello large, per le sue caratteristiche di indirizzamento (figura 5), è probabilmente il più flessibile, anche se non il più efficiente. Le funzioni contenute in una libreria compilata per il modello large[140] possono essere utilizzate senza problemi anche da programmi compilati per altri modelli: è sufficiente ricordarsi che tutti i puntatori parametri delle funzioni sono `far` e che le funzioni devono essere prototipizzate anch'esse come `far`: se questi non sono i default del modello di memoria utilizzato occorre agire di conseguenza. Esempio: abbiamo un sorgente, `PIPPO.C`, da compilare con il modello small, nel quale deve essere inserita una chiamata a `funzStr()` (che accetta un puntatore a carattere

[140] Ogni modello richiede una libreria contenente moduli oggetto di funzioni compilate in quel particolare modello. Fa eccezione il modello tiny, che utilizza la medesima libreria del modello small (è però differente lo startup code).

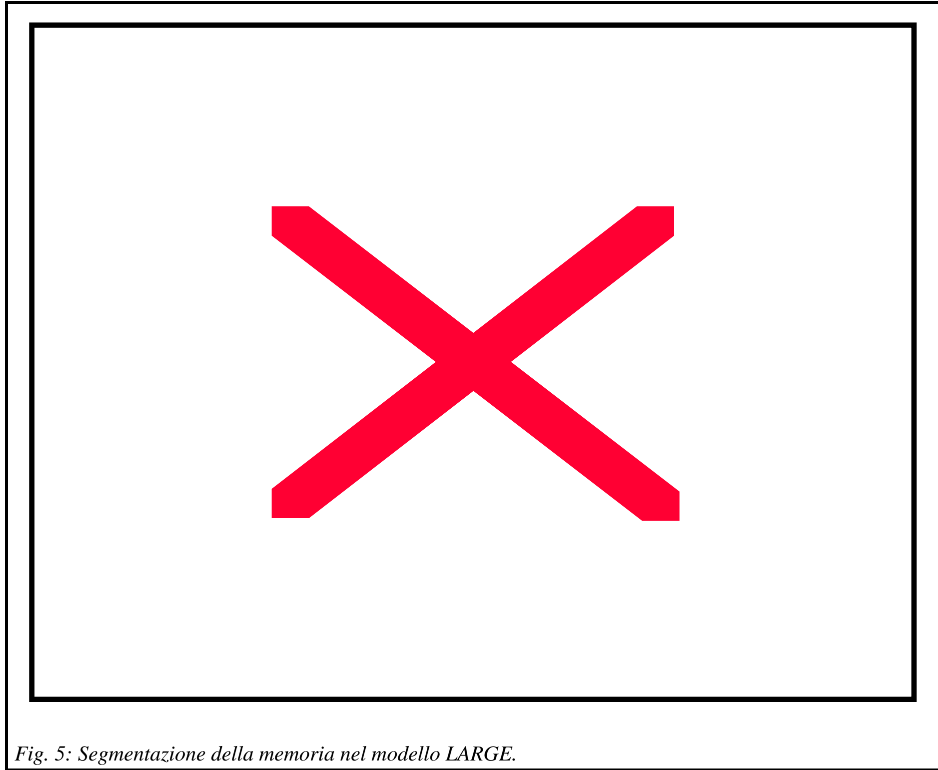

Fig. 5: Segmentazione della memoria nel modello LARGE.

quale parametro e restituisce un puntatore a carattere) disponibile nella libreria `LARGELIB.LIB`, predisposta per il modello large. Alla libreria è associato uno header file, `LARGELIB.H`, che contiene il seguente prototipo di `funzStr()`:

```
char *funz(char *string);
```

La funzione e i puntatori (il parametro e quello restituito) non sono dichiarati `far`, perché nel modello large tutti i puntatori e tutte le funzioni lo sono per default. Se non si provvede ad informare il compilatore che, pur essendo il modello di memoria small, `funzStr()`, i suoi parametri e il valore restituito sono `far`, si verificano alcuni drammatici problemi: in primo luogo, lo stack è gestito come se entrambi i puntatori fossero `near`. Ciò significa che a `funzStr()`, in luogo di un valore a 32 bit, ne viene passato uno a 16; il codice di `funzStr()`, però, lavora comunque su 32 bit, prelevando dallo stack 16 bit di "ignota provenienza" in luogo della vera parte segmento del puntatore. La `funzStr()`, inoltre, restituisce un valore a 32 bit utilizzando la coppia di registri `DX:AX`, ma la funzione chiamante, aspettandosi un puntatore a 16 bit, ne considera solo la parte in `AX`, cioè l'offset. Ma ancora non basta: la chiamata a `funzStr()` generata dal compilatore è `near`, secondo il default del modello small, perciò, a run-time, la `CALL` salva sullo stack solo il registro `IP` (e non la coppia `CS:IP`). Quando `funzStr()` termina, la `RETF` (far return) estrae dello stack 32 bit e ricarica con essi la coppia `CS:IP`; anche in questo caso, 16 di quei 32 bit sono di "ignota provenienza". Ce n'è quanto basta per bloccare la macchina alla prima chiamata. E' indispensabile correre ai ripari, modificando come segue il prototipo in `LARGELIB.H`

```
char far * far funzStr(char far *string);
```

e dichiarando esplicitamente far i puntatori coinvolti nella chiamata a `funzStr()`:

```
....
char far *parmPtr, far *retPtr;
....
retPtr = funzStr(parmPtr);
....
```

A dire il vero, si può evitare di dichiarare `parmPtr` esplicitamente `far`, perché il compilatore, dall'esame del prototipo, è in grado di stabilire quale tipo di puntatore occorre passare a `funzStr()` e provvede da sé copiando sullo stack un puntatore far costruito come `DS:parmPtr`; la dichiarazione `far`, comunque, non guasta, purché ci si ricordi di avere a che fare con un puntatore a 32 bit anche laddove ciò non è richiesto.

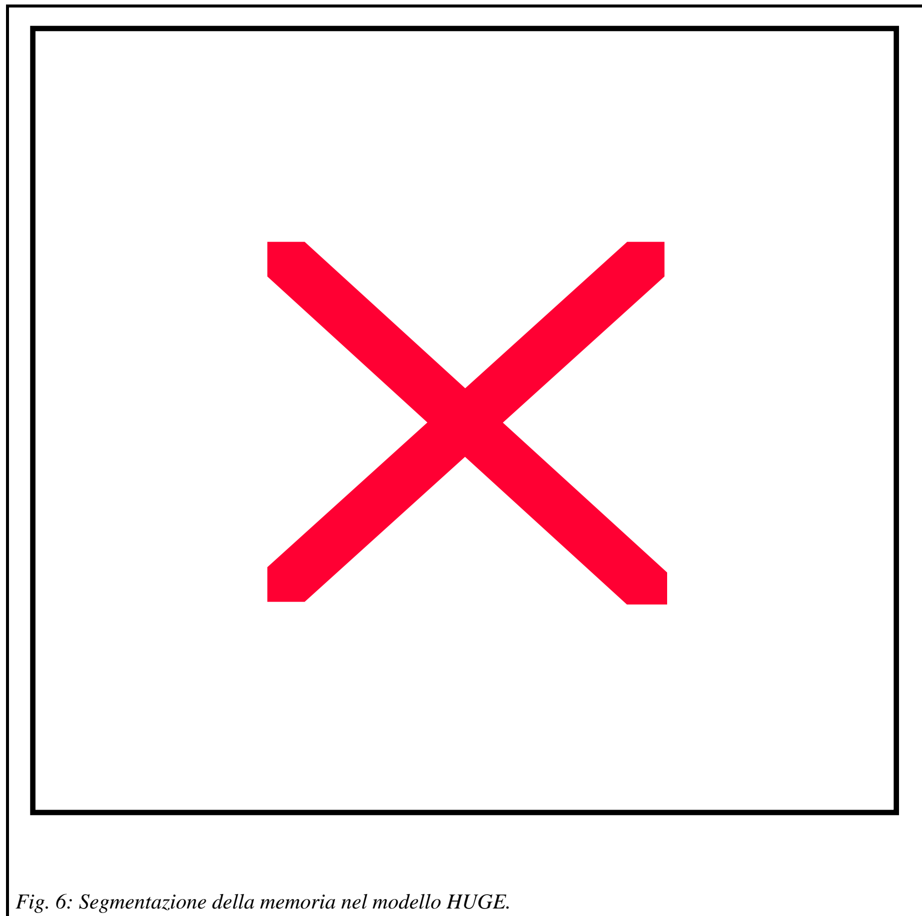

Fig. 6: Segmentazione della memoria nel modello HUGE.

Per facilitare l'uso dei puntatori `far` nei modelli di memoria tiny, small e medium sono state di recente aggiunte alle librerie standard nuove versioni (adatte a puntatori a 32 bit) di alcune funzioni molto usate: accanto a `strcpy()` troviamo perciò `_fstrcpy()`, e così via.

HUGE MODEL

Il modello huge consente di gestire (in teoria) sino ad 1 Mb di dati statici e globali, estendendo ad essi la modalità di indirizzamento implementata dai modelli large e medium per il codice. E' l'unico modello che estende ad 1 Mb il limite teorico sia per il codice che per tutti i tipi di dato.

L'opzione (Borland) per la compilazione secondo il modello huge è `-mh`.

Dal momento che la dimensione di ogni singolo modulo oggetto di dati statici e globali non può superare i 64 Kb, il superamento del limite dei 64 Kb è da intendersi per l'insieme dei dati stessi; non è possibile avere un singolo dato `static` (ad esempio un array) di dimensioni maggiori di 64 Kb. Il registro `DS` è inizializzato con l'indirizzo di uno dei moduli di dati statici e globali (figura 6).

SCRIVERE FUNZIONI DI LIBRERIA

La scrittura di un programma C implica sempre la necessità di scrivere funzioni, in quanto almeno `main()` deve essere definita. Spesso, però, le funzioni che fanno parte di uno specifico programma sono scritte avendo quali linee guida la struttura e gli obiettivi di quello. Leggermente diverso è il comportamento da tenere quando si scrivano funzioni destinate a far parte di una libreria e, come tali, utilizzabili almeno in teoria da qualsiasi programma: in questo caso è opportuno osservare alcune regole, parte delle quali derivano dal buon senso e dalla necessità di scrivere codice qualitativamente valido; parte, invece, dettate dalle esigenze tecniche del linguaggio e dei compilatori.

ACCORGIMENTI GENERALI

Nello scrivere funzioni di libreria va innanzitutto ricordato che il codice scritto può essere utilizzato nelle situazioni più disparate: è pertanto indispensabile evitare, per quanto possibile, qualsiasi assunzione circa le condizioni operative a runtime.

Supponiamo, ad esempio, di scrivere una funzione in grado di copiare in un buffer il contenuto della memoria video: se il codice deve far parte di un programma, magari preparato per una specifica macchina, è possibile che le modalità operative (tipo di monitor, pagina video attiva) siano note al momento della compilazione e non pongano dunque problemi di sorta. Ma se la funzione deve essere inserita in una libreria, non può ipotizzare nulla circa tali condizioni: è opportuno, allora, che esse siano richieste quali parametri. In alternativa la funzione stessa può incorporare alcune routine atte a conoscere tutti i parametri operativi necessari mediante le opportune chiamate al BIOS. Oppure, ancora, possono essere predisposte una o più funzioni "complementari", da chiamare prima di quella in questione, che memorizzino i dati necessari in variabili globali.

L'indipendenza del codice dalle condizioni operative del programma è anche detta parametricità, e rappresenta un requisito essenziale delle funzioni di libreria.

Un'altra importante osservazione riguarda la coerenza delle regole di interfacciamento funzioni/programma. Accade spesso di scrivere gruppi di funzioni le quali, nel loro insieme, permettono di gestire in modo più o meno completo determinate situazioni o caratteristiche del sistema in cui opera il programma che le utilizza. E' bene che le funzioni inserite in una libreria, ed in particolare quelle che implementano funzionalità tra loro correlate, siano simili quanto a parametri, valori restituiti e modalità di gestione degli errori. In altre parole, esse dovrebbero, per quanto possibile, somigliarsi reciprocamente. Con riferimento ad un gruppo di funzioni che utilizzino servizi DOS per realizzare particolari funzionalità, si può pensare ad una modalità standard di gestione degli errori, nella quale il valore restituito è sempre il codice di stato a sua volta restituito dal DOS. In alternativa ci si può uniformare alla modalità implementata da gran parte delle funzioni della libreria standard, che prevedono la restituzione del valore `-1` in caso di errore e la memorizzazione del codice di errore nella variabile globale `errno`: a pag. 499 sono forniti la descrizione di come tale algoritmo sia realizzato nella libreria C ed un esempio di utilizzo della funzione (non documentata) `__IOerror()`.

Ancora, i nomi di variabili e funzioni dovrebbero essere il più possibile autoesplicativi: dalla loro lettura dovrebbe cioè risultare evidente il significato della variabile o il compito della funzione. Al proposito sono state sviluppate specifiche formali[141] che descrivono un possibile metodo per uniformare i nomi C basato, tra l'altro, sulle modalità di allocazione delle variabili e su un insieme di suffissi standard per le funzioni. Se non si scrive codice a livello professionale, tali formalismi possono forse risultare eccessivi; è bene comunque ricordarsi che le funzioni di libreria sono spesso utilizzate da terzi, i quali è

---

[141] Si tratta della *Notazione Ungherese*, così detta dalla nazionalità del suo inventore C. Simonyi. E', tra l'altro, la convenzione utilizzata per i simboli in ambiente Microsoft Windows.

bene possano concentrarsi sul programma che stanno implementando piuttosto che essere costretti a sforzarsi di decifrare significati e modalità di utilizzo di una interfaccia software criptica, disordinata e disomogenea.

Analoghe considerazioni valgono per la documentazione delle funzioni. E' indispensabile che le librerie siano accompagnate da una chiara e dettagliata descrizione, per ciascuna funzione, del tipo e del significato di tutti i parametri richiesti e del valore eventualmente restituito. Del pari, è opportuno fornire esaustiva documentazione delle strutture ed unioni definite, delle variabili globali e delle costanti manifeste.

ESIGENZE TECNICHE

Alcune regole derivano invece dalle caratteristiche proprie del linguaggio C e dei compilatori. Si è detto (pag. 87 e seguenti) che, per verificare la correttezza sintattica della chiamata a funzione e gestirla nel modo opportuno, il compilatore deve conoscere le regole di interfacciamento tra la stessa funzione e quella chiamante (cioè la coerenza tra i parametri formali e quelli attuali). Dal momento che una funzione di libreria non è mai definita nel sorgente del programma che ne fa uso, è necessario fornirne il prototipo. Allo scopo si rivelano particolarmente adatti gli header file (`.H`): è bene, pertanto, che una libreria di funzioni sia sempre accompagnata da uno o più file `.H` contenenti tutti i prototipi delle funzioni, la dichiarazione (come variabili external, pag. 39) di tutte le variabili globali, i template delle strutture ed unioni, nonché le costanti manifeste eventualmente definite per comodità del programmatore.

Si ricordi, poi, che una libreria di funzioni non è che un file contenente uno o più object file (`.OBJ`), generati dalla compilazione dei rispettivi sorgenti. Detti moduli oggetto possono derivare da sorgenti scritti in linguaggi diversi dal C: è frequente, soprattutto per l'implementazione di rouine dei basso livello, il ricorso al linguaggio Assembler. E' evidente che negli include file devono essere dati anche i prototipi delle funzioni facenti parti di moduli assembler. Inoltre, dal momento che, per default, il compilatore genera un *underscore* (il carattere "_") in testa ai nomi delle funzioni C, mentre ciò non viene fatto dall'assemblatore, i nomi di tutte le funzioni definite in moduli assembler devono inziare con un underscore, che viene ignorato nelle chiamate nel sorgente C. Se, ad esempio, la libreria contiene il modulo oggetto relativo alla funzione assembler definita come segue:

```
....
_machineType proc near
    ....
_machineType endp
....
```

il prototipo fornito nello header file è:

```
int machineType(void);                              // senza undescore iniziale!!
```

e le chiamate nel sorgente C saranno analoghe alla seguente:

```
    ....
    int cpu;
    ....
    cpu = machineType();                         // niente underscore neppure qui!
    ....
```

Va ancora sottolineato che, essendo il C un linguaggio *case-sensitive*, anche la compilazione dei sorgenti assembler mediante l'assemblatore deve essere effettuata in modo che le maiuscole siano distinte dalle minuscole, attivando le opportune opzioni[142].

[142] Dal momento che l'assembler è un linguaggio case-insensitive, è improbabile che tali opzioni siano attive per default.

Se le funzioni sono scritte in C ma incorporano parti di codice in assembler, è opportuno prestare particolare attenzione alle istruzioni che referenziano i parametri formali (e soprattutto i puntatori): per un esempio vedere pag. 196. Maggiori dettagli sull'interazione tra C ed assembler si trovano a pagina 155 e seguenti.

Qualche raccomandazione in tema di variabili globali. Quando si scrive un gruppo di funzioni che per lo scambio reciproco di informazioni utilizzano anche variabili globali, è opportuno che queste siano dichiarate `static` se non devono essere referenziate dal programma che utilizza quelle funzioni: in tal modo si accentua la coerenza logica del codice, impedendo la visibilità delle variabili "ad uso riservato" all'esterno del modulo oggetto che contiene le funzioni. Esempio:

```
static int commonInfo;                    // visibile solo in f_a(), f_b() e f_c()
....
void f_a(int iParm)
{
    extern int commonInfo;
    ....
}

int f_b(char *sParm)
{
    ....                          // non referenzia commonInfo (ma potrebbe farlo)
}

int f_c(char *sParm,int iParm)
{
    extern int commonInfo;
    ....
}
```

E' però indispensabile che tutte le funzioni che referenziano dette variabili siano definite nel medesimo file sorgente in cui quelle sono dichiarate.

Considerazioni analoghe valgono anche per le funzioni: una funzione implementata unicamente come subroutine di servizio per un'altra può essere dichiarata `static` (e resa invisibile all'esterno del modulo oggetto) purché definita nel medesimo sorgente di questa.

Attenzione, però: se una variabile globale (o una funzione) deve essere referenziabile dal programma che utilizza la libreria, essa non deve assolutamente essere dichiarata `static`.

Qualche precauzione è richiesta anche nella gestione dei puntatori. Non va dimenticato che i puntatori non dichiarati esplicitamente `near`, `far` o `huge` (pag. 21) sono implementati dal compilatore con 16 o 32 bit a seconda del modello di memoria (pag. 143) utilizzato; analoga regola si applica inoltre alle funzioni (pag. 93). Ne segue che solo le funzioni dichiarate `far`, che accettano quali parametri e restituiscono puntatori esplicitamente `far` possono essere utilizzate senza problemi in ogni programma, indipendentemente dal modello di memoria con il quale esso è compilato.

Al proposito, è regola generale scrivere le funzioni senza tenere conto del modello di memoria[143] e generare diversi file di libreria, uno per ogni modello di memoria a cui si intende fornire supporto (è necessario, come si vedrà tra breve, compilare più volte i sorgenti). Si noti che i compilatori C sono accompagnati da una dotazione completa di librerie per ogni modello di memoria gestito.

Si è detto, poco fa, che una libreria è, dal punto di vista tecnico, un file contenente più moduli oggetto, ciascuno originato dalla compilazione di un file sorgente. Durante la fase di linking vengono individuati, all'interno della libreria, i moduli oggetto in cui si trovano le funzioni chiamate nel programma e nell'eseguibile in fase di creazione è importata una copia di ciascuno di essi. Ciò significa che se una funzione è chiamata più volte, il suo codice compilato compare una volta sola nel programma

[143] Eccetto i casi in cui sia necessario implementare il codice in maniera dipendente proprio dal modello di memoria, utilizzando la compilazione condizionale: ancora una volta si rimanda all'esempio di pagina 196, nonché a pag. 44.

eseguibile; tuttavia, se un modulo oggetto implementa più funzioni, queste sono importate in blocco nell'eseguibile anche qualora una sola di esse sia effettivamente utilizzata nel programma. Appare pertanto conveniente, a scopo di efficienza, definire in un unico sorgente più funzioni solo se, per le loro caratteristiche strutturali, è molto probabile (se non certo) che esse siano sempre utilizzate tutte insieme. Ad esempio, tutte le subroutine di servizio di una funzione dovrebbero essere definite nel medesimo sorgente di questa: ciò minimizza il tempo di linking senza nulla sottrarre all'efficienza del programma in termini di spazio occupato.

LA REALIZZAZIONE PRATICA

A complemento delle considerazioni teoriche sin qui esposte, vediamo quali sono le operazioni necessarie per la costruzione di una libreria di funzioni.

In primo luogo occorre scrivere il codice ed effettuare il necessario debugging, ad esempio aggiungendo al sorgente una `main()` che richiami le funzioni in modo da testare, nel modo più completo possibile, tutte le caratteristiche implementate. Al termine della fase di prova bisogna assolutamente ricordarsi di eliminare la `main()`, in quanto nessuna libreria C può includere una funzione con tale nome.

Nell'ipotesi di avere realizzato un gruppo di sorgenti chiamati, rispettivamente, `MYLIB_A.C`, `MYLIB_B.C`, `MYLIB_C.C` e `MYLIB_D.C`, accompagnati dallo header file `MYLIB.H`, si può procedere, a questo punto, alla generazione dei moduli oggetto e della libreria; le operazioni descritte di seguito dovranno essere ripetute per ogni modello di memoria (eccetto il modello tiny, che utilizza le medesime librerie del modello small). Negli esempi che seguono si propone la costruzione della libreria per il modello large.

Si parte sempre dalla compilazione dei sorgenti: dal momento che non si vuole generare un programma eseguibile, ma solamente i moduli oggetto, è necessaria l'opzione `-c` sulla riga di comando del compilatore[144]:

```
bcc -c -ml mylib_a.c mylib_b.c myliv_c.c mylib_d.c
```

L'opzione `-ml` richiede che la compilazione sia effettuata per il large memory model; l'operazione produce, in assenza di errori, i moduli oggetto `MYLIB_A.OBJ`, `MYLIB_B.OBJ`, `MYLIB_C.OBJ` e `MYLIB_D.OBJ`.

E' ora possibile generare il file di libreria mediante la utility `TLIB` (o `LIB`, a seconda del compilatore utilizzato):

```
tlib mylibl /C +mylib_a +mylib_b +mylib_c +mylib_d
```

L'opzione `/C` richiede che la generazione della libreria avvenga in modalità case-sensitive. Il nome file libreria è `MYLIBL.LIB`; se non esiste esso è creato e vi sono inseriti i quattro moduli oggetto preceduto dall'operatore "`+`". Si noti che il nome del file deve essere differenziato per ogni modello di memoria; è pratica comune indicare il modello supportato mediante una lettera aggiunta in coda al nome (`S` per small e tiny, `M` per medium, `C` per compact, `L` per large, `H` per huge). L'estensione è, per default, `.LIB`.

Il pacchetto di libreria completo è perciò costituito, in definitiva, dal file `MYLIB.H`, unico per tutti i modelli di memoria, e dai file `MYLIBS.LIB`, `MYLIBM.LIB`, `MYLIBC.LIB`, `MYLIBL.LIB` e `MYLIBH.LIB`. E' ovvio che la libreria può essere pienamente utilizzata da chi entri in possesso dei file appena elencati (e della documentazione!), senza necessità alcuna di disporre anche dei file sorgenti.

---

[144] La sintassi descritta negli esempi è quella del compilatore Borland. Per altri compilatori è opportuno consultare la documentazione con essi fornita.

Va infine sottolineato che la utility TLIB permette anche di effettuare operazioni di manutenzione: se, ad esempio, a seguito di modifiche si rendesse necessario sostituire all'interno della libreria il modulo mylib_a con una differente versione, il comando

```
tlib mylibl +- mylib_a
```

raggiunge lo scopo. Si noti che, nonostante l'operatore "+" sia specificato prima dell'operatore "-", l'operazione di eliminazione è eseguita sempre prima di quella di inserimento. L'operatore "*" consente di estrarre dalla libreria una copia di un modulo oggetto: il comando

```
tlib mylibl *mylib_a
```

genera il file MYLIB_A.OBJ, sovrascrivendo quello che eventualmente preesiste nella directory.

Per una descrizione completa della utility di manutenzione delle librerie si rimanda comunque alla documentazione fornita con il compilatore.

INTERAZIONE TRA C E ASSEMBLER

Il C, pur rientrando tra i linguaggi di alto livello, rende disponibili potenti funzionalità di gestione dello hardware: nelle librerie di tutti (o quasi) i compilatori oggi in commercio sono incluse funzioni progettate appositamente per controllare "da vicino" e pilotare il comportamento del BIOS e delle porte. A questo va aggiunta la notevole efficienza del codice compilato, una delle caratteristiche di maggior pregio dei programmi scritti in C.

Ciononostante, il controllo totale del sistema nel modo più efficiente possibile è ottenibile solo tramite la programmazione in linguaggio assembler, in quanto esso costituisce la traduzione letterale, in termini umani, del linguaggio macchina, cioè dell'insieme di istruzioni in codice binario che il processore installato sul computer è in grado di eseguire[145]. In altre parole, ogni istruzione assembler corrisponde (in prima approssimazione) ad una delle operazioni elementari eseguibili dalla CPU.

Si comprende perciò come il realizzare parte di un programma in C ricorrendo direttamente all'assembler possa accentuarne la potenza e l'efficienza complessive.

In generale, si può affermare che esistono due differenti approcci metodologici alla realizzazione di programmi parte in linguaggio C e parte in linguaggio assembler.

ASSEMBLER

Il primo metodo consiste nello scrivere una o più routine interamente in linguaggio assembler: è necessaria un'ottima padronanza del linguaggio, con particolare riferimento alla gestione dello stack e dei segmenti di codice in relazione ai differenti modelli di memoria. Assemblando il sorgente si ottiene un modulo oggetto (un file `.OBJ`) che deve essere collegato in fase di linking ai moduli oggetto generati dal compilatore C; le routine così implementate possono essere richiamate nel sorgente C come una funzione qualsiasi. Si tratta di un ottimo sistema per realizzare librerie di funzioni, ma, di solito, alla portata unicamente dei più esperti. Ecco un semplice esempio di sorgente assembler per una funzione che stampa sullo standard output il carattere passatole come parametro, facendolo seguire da un punto esclamativo, e restituisce `1`:

```
_TEXT segment byte public 'CODE'
_TEXT ends
DGROUP group _DATA,_BSS
    assume cs:_TEXT,ds:DGROUP
_DATA segment word public 'DATA'
_p_escl label   byte
    db '!'
_DATA ends
_BSS segment word public 'BSS'
_BSS ends
_TEXT segment byte public 'CODE'
    assume cs:_TEXT
_stampa proc near
    push bp
    mov bp,sp
```

[145] Tant'è che ogni microprocessore ha il proprio linguaggio assembler. I personal computer generalmente indicati come "IBM o compatibili" sono basati sulla famiglia di processori Intel 80x86: 8086, 80286, 80386, 80386SX, 80486 ed altri ancora. Esiste un vasto insieme di istruzioni comuni a tutti questi microprocessori: di qui la possibilità concreta di scrivere programmi in grado di girare su macchine di tipo differente. Il programmatore che intenda sfruttare le prestazioni più avanzate offerte da ciascuno di essi (in particolare dal tipo 80286 in poi) deve però rinunciare alla compatibilità del proprio programma con i processori che non dispongono di tali caratteristiche.

```
    mov ah,2
    mov dl,[bp+4]
    int 21h
    mov ah,2
    mov dl,DGROUP:_p_escl
    int 21h
    mov ax,1
    pop bp
    ret
_stampa endp
_TEXT ends
    public _p_escl
    public _stampa
    end
```

Un programma C può utilizzare la funzione del listato dopo averla dichiarata:

```
    ....
    int stampa(char c);
    ....
```

Va osservato che il nome assembler della funzione è `_stampa`, mentre in C l'underscore non compare. In effetti, per default, il compilatore modifica i nomi di tutte le funzioni aggiungendovi in testa un underscore, perciò quando si scrive una funzione direttamente in assembler bisogna ricordarsi che il nome deve iniziare con "_". Nelle chiamate alla funzione inserite nel sorgente C, invece, l'underscore deve essere omesso.

Come si vede, in questo caso la maggior parte[146] del listato assembler non è destinato a produrre codice eseguibile, ma ad informare l'assemblatore sulla struttura dei segmenti di programma, sulla presenza di simboli pubblici e sulla gestione dei registri di segmento. Per completezza presentiamo la funzione `stampa()` realizzata in C:

```
char p_escl = '!';
int bdos(int dosfn,int dosdx,int dosal);

int stampa(char c)
{
    (void)bdos(2,c,0);
    (void)bdos(2,p_escl,0);
    return(1);
}
```

La maggiore compattezza del listato C è evidente. Il lettore curioso che compili la versione C di `stampa()` richiedendo al compilatore di produrre il corrispondente sorgente assembler[147] ottiene un listato strutturato come quello discusso poco sopra. Il file eseguibile generato a partire dalla funzione C è però, verosimilmente, di dimensioni maggiori (a parità di altre condizioni), in quanto il linker collega ad esso anche il modulo oggetto della funzione di libreria `bdos()`.

Per sviluppare a fondo l'argomento sarebbe necessaria una approfondita digressione sul linguaggio assembler, per la quale preferiamo rimandare alla vasta manualistica disponibile: chi desidera ottimizzare, laddove necessario, i propri sorgenti C ricorrendo a un poco di assembler (e senza esserne un vero esperto) non perda le speranze...

---

[146] Data la banalità della funzione in sé medesima, si tratta, in fondo, di un caso limite.

[147] Il compilatore C Borland accetta, allo scopo, l'opzione `-S`:

```
bcc -S stampa.c
```

INLINE ASSEMBLY

Alcuni compilatori C[148] supportano un secondo metodo di programmazione mista, consentendo la compresenza di codice C e assembler nel medesimo sorgente: in altre parole essi permettono al programmatore di assumere a basso livello il controllo del flusso di programma, senza richiedere una conoscenza della sintassi relativa alla segmentazione del codice più approfondita di quella necessaria per la programmazione in C "puro". La funzione `stampa()` può allora essere realizzata così:

```
#pragma  inline                    // informa il compilatore: usato inline assembly

char p_escl = '!';

int stampa(char c)
{
   asm mov ah,2;
   asm mov dl,c;
   asm int 21h;
   asm mov ah,2;
   asm mov dl,p_escl;
   asm int 21h;
   return(1);
}
```

Il compilatore C si occupa di generare il sorgente assembler inserendovi, senza modificarle[149], anche le righe precedute dalla direttiva `asm`[150], invocando poi l'assemblatore (che effettua su di esse i controlli sintattici del caso) per generare il modulo oggetto. Segmentazione e modelli di memoria sono gestiti dal compilatore stesso come di norma, in modo del tutto trasparente al programmatore.

---

[148] Ad esempio le più recenti versioni dei compilatori Microsoft e Borland. Il contenuto del presente capitolo fa riferimento specifico alle possibilità offerte dal secondo.

[149] Per la verità, le istruzioni inline assembly vengono modificate laddove contengano riferimenti a simboli C (ad esempio nomi di variabili, come vedremo). Nella maggior parte dei casi ciò rappresenta un vantaggio, perché consente di referenziare le variabili definite direttamente in C, come nell'istruzione

```
asm mov dl,p_escl;
```

ma in alcuni casi è fonte di grattacapi non da poco. Esempietto chiarificatore, tra i diversi possibili: se una istruzione assembly contiene l'operatore `DUP`, che serve a inizializzare più byte ad un dato valore, e prima di tale istruzione è incluso il file `IO.H`, contenente il prototipo della funzione C `dup()`, che duplica lo handle di un file, il compilatore la scambia (orrore!) per una chiamata a detta funzione C, confondendo le idee all'assemblatore. La soluzione è una sola: includere il file `IO.H` dopo tutte le righe assembly che fanno uso di `DUP`. Nel vostro programma non è possibile? Peggio per voi: non vi rimane che copiare il file `IO.H` e modificare la copia eliminando la dichiarazione del prototipo di `dup()`; è ovvio che nel sorgente C deve essere inclusa la copia così modificata.

[150] In alternativa, la direttiva `asm` può introdurre un blocco di istruzioni racchiuso, come di consueto, tra parentesi graffe:

```
asm {
    mov ah,2;
    ....
    int 21h;
}
```

Calma, calma: non è questa la sede adatta a presentare le regole sintattiche relative all'uso dello inline assembly nei sorgenti C: esse sono più o meno esaurientemente trattate nella documentazione di corredo ai compilatori; inoltre il testo è ricco di esempi nei quali è frequente il ricorso alla programmazione mista. In questa sede intendiamo soffermarci, piuttosto, su alcune questioni in apparenza banali, ma sicura fonte di noiosi problemi per il programmatore che non le tenga nella dovuta considerazione.

Non bisogna dimenticare, infatti, che una singola istruzione[151] di un linguaggio di alto livello (quale è il C) può corrispondere a più istruzioni di linguaggio macchina, e dunque di assembler. A ciò si aggiunga che i compilatori, di norma, dispongono di opzioni di controllo delle ottimizzazioni[152]: non è sempre agevole, dunque, prevedere con precisione quali registri della CPU vengono di volta in volta impegnati e a quale scopo, o quale struttura assumono cicli e sequenze di salti condizionati.

Va ancora sottolineato che è buona norma inserire sempre la direttiva riservata

```
#pragma inline
```

in testa ai sorgenti in cui si faccia uso dello inline assembly, onde evitare strani messaggi di errore o di warning da parte del compilatore, anche laddove tutto è in regola.

Qualche ulteriore approfondimento potrà servire.

Lo stack

Si consideri la seguente funzione:

```
void prova(char var1,int var2)
{
    char result;

    asm mov ah,2;
    result = var1*var2;
    asm mov dl,result;
    asm int 21h;
    asm ret;
}
```

e la si confronti con il corrispondente codice assember generato dal compilatore[153]:

```
....
_prova proc near
    push bp                   ; gestione stack: salva l'ind. della base
    mov bp,sp                 ; crea lo stack privato della funzione
    dec sp                    ; riserva lo spazio per result (word)
    dec sp
    mov ah,2                  ; carica AH per INT 21h, servizio 2
    mov al,byte ptr [bp+4]    ; carica AL con var1
```

[151] Il termine va inteso in senso lato: in questo caso con "istruzione" si indicano anche le chiamate a funzione. Le funzioni, a loro volta, possono essere considerate sequenze di istruzioni delle quali la routine chiamante conosce esclusivamente le modalità di interfacciamento al programma (in altre parole la struttura dell'input e dell'output).

[152] Essi sono cioè in grado, su richiesta, di compilare il codice sorgente in modo che il codice oggetto prodotto sia il più compatto possibile, oppure il più veloce possibile, e così via.

[153] Compilato con l'opzione `-S` sulla riga di comando. Il codice riportato è solo una parte di quello prodotto dal compilatore: sono state eliminate tutte le parti relative alla segmentazione.

```
    cbw                        ; "promuove" AL (var1) ad integer
    imul word ptr [bp+6]       ; moltiplica var1 * var2 (integer * integer)
    mov byte ptr [bp-1],al     ; carica result (un char) con il risultato
    mov dl,[bp-1]              ; carica DL con result
    int 21h                    ; invoca servizio DOS
    ret                        ; ritorna alla routine chiamante
    mov sp,bp                  ; gestione stack
    pop bp
    ret                        ; ritorna alla routine chiamante
_prova endp
....
```

La funzione `prova()` dovrebbe caricare i registri `AH` e `DL` per invocare il servizio 2 dell'int 21h (scrittura del carattere in `DL` sullo standard output), ma, invocandola, si ottiene un crash di sistema.

Il motivo va ricercato nell'istruzione inline assembly `RET`: essa impedisce che siano eseguite le istruzioni generate dal compilatore per gestire correttamente lo stack in uscita dalla funzione; infatti la sequenza

```
MOV SP,BP
POP BP
```

rappresenta la controparte delle istruzioni

```
PUSH BP
MOV BP,SP
DEC SP
DEC SP
```

che si trovano in testa al codice. Se ne trae che è opportuno, salvo casi particolari, usare l'istruzione C `return` in uscita dalle funzioni (tra parentesi, pur eliminando la `RET` il risultato non è ancora quello voluto: chi sia curioso di consocerne il motivo, può vedere a pag. 161).

Lo stack è in sostanza utilizzato per il passaggio dei parametri alle funzioni e per l'allocazione delle loro variabili locali. Senza entrare nel dettaglio, esso è gestito mediante alcuni registri dedicati: `SS` (puntatore al segmento dello stack), `SP` (puntatore, nell'ambito del segmento individuato da `SS`, all'indirizzo che ospita l'ultima word salvata sullo stack) e `BP` (puntatore base, per la gestione dello stack locale delle funzioni).

Una chiamata a funzione richiede che siano copiati (`PUSH`) sullo stack tutti i parametri[154]: ad ogni istruzione `PUSH` eseguita, `SP` viene decrementato di due (lo stack è gestito a word, procedendo a ritroso dalla parte alta del suo segmento) per puntare all'indirizzo al quale memorizzare il parametro (o una word del parametro stesso); soltanto quando tutti i parametri sono stati copiati nello stack viene eseguita la `CALL` che trasferisce il controllo alla funzione (e modifica ancora una volta `SP`, salvando sullo stack l'indirizzo[155] al quale deve essere trasferita l'esecuzione in uscita alla funzione stessa). Ne segue che

---

[154] Non bisogna dimenticare che il C garantisce che i parametri siano passati alle funzioni per valore: in altre parole alla funzione è passata una copia di ogni parametro, ottenuta copiando il valore di questo nello stack, dal quale la funzione lo preleva. Ciò implica che una funzione non può modificare il valore delle variabili che le sono date quali parametri attuali. Vedere pag. 87 e seguenti.

[155] Se la chiamata è `near`, cioè se il codice della funzione è compreso nello stesso segmento in cui appare la `CALL`, viene salvato sullo stack solamente un offset (il valore che `IP` deve assumere perché sia eseguita la prima istruzione che segue la `CALL` nella routine chiamante) e quindi `SP` è decrementato di 2. Se, al contrario, la chiamata è di tipo `far`, vengono salvati sullo stack un segmento ed un offset, e pertanto `SP` è decrementato di 4.

ogni funzione può recuperare i propri parametri ad offset positivi rispetto a SP[156], ed allocare spazio nello stack per le proprie variabili locali ad offset negativi (sempre rispetto a SP): infatti BP viene salvato sullo stack (PUSH BP con ulteriore decremento di SP) e caricato con il valore di SP (MOV BP,SP); BP costituisce, a questo punto, la base di riferimento per detti offset. Se sono definite variabili locali SP è decrementato (DEC o SUB) per riservare loro lo spazio necessario[157] nello stack. In uscita, la funzione deve disallocare la parte di stack assegnata alle variabili locali (MOV SP,BP) ed eliminare BP dallo stack medesimo (POP BP): in tal modo la RET (o RETF) può estrarre, sempre dallo stack, il corretto indirizzo a cui trasferire l'esecuzione. La restituzione di un valore alla funzione chiamante non coinvolge lo stack: essa avviene, di norma, tramite il registro AX, o la coppia DX:AX se si tratta di un valore a 32 bit. Alla routine chiamante spetta il compito di liberare lo stack dai parametri passati alla funzione invocata: lo scopo è raggiunto incrementando opportunamente SP con una istruzione ADD (se i parametri sono pochi, il compilatore tende ad ottimizzare il codice generando invece una o più PUSH di un registro libero, di solito CX).

La gestione della struttura dello stack è invisibile al programmatore C, in quanto il codice necessario al mantenimento della cosiddetta standard stack frame è generato automaticamente dal compilatore, ma un uso poco accorto dello inline assembly può interferire (distruttivamente!) con tali delicate operazioni, come del resto gli esempi poco sopra riportati hanno evidenziato.

Consideriamo ora un caso particolare: le funzioni che non prendono parametri e non definiscono variabili locali non fanno uso dello stack, quindi il compilatore può evitare di generare le istruzioni necessarie alla standard stack frame. Chi utlizzi lo inline assembly deve documentarsi con molta attenzione sulle caratteristiche del compilatore utilizzato, in quanto alcuni prodotti ottimizzano il codice evitando di generare dette istruzioni, mentre altri le generano in ogni caso privilegiando la standardizzazione del comportamento delle funzioni[158].

Qualche cenno meritano infine le funzioni dichiarate interrupt[159], dal momento che il compilatore si occupa di gestire lo stack in un modo particolare: infatti una funzione interrupt, normalmente, è destinata a sostituirsi ad un gestore di interrupt di sistema (o a modificarne il comportamento) e dunque non viene invocata dal programma di cui è parte, ma dal sistema operativo o dallo hardware. Si rendono pertanto necessarie alcune precauzioni, quali preservare lo stato dei registri al momento della chiamata e ripristinare i flag in uscita. A questo pensa, automaticamente, il compilatore, ma deve tenerne conto il programmatore che intenda servirsi dello inline assembly nel codice della funzione (ed è un caso frequente): i registri sono, ovviamente, salvati sullo stack in testa alla funzione, e devono esserne estratti prima di restituire il controllo al processo chiamante. Dunque attenzione, a scanso

[156] Per essere più precisi: rispetto al valore che SP ha al momento dell'ingresso nella funzione, cioè dopo la CALL. Per questo entra in gioco il registro BP.

[157] In termini di word. Esempi: un int richiede una word; un long ne richiede due; tre char ne richiedono tre; un array di nove char ne richiede cinque; due array di nove char ne richiedono cinque ciascuno. E' ovvio che in assenza di variabili locali non vi sono istruzioni SUB SP,... o DEC SP in testa, né la MOV SP,BP in coda alla funzione.

[158] Tanto per fare un esempio: il compilatore Borland TURBO C 2.0 non mantiene la standard stack frame, e quindi genera le istruzioni per la gestione dello stack solo nelle funzioni in cui sono indispensabili. Il compilatore Borland C++, dalla versione 1.0 in poi, genera per default, in qualsiasi funzione, le istruzioni necessarie alla standard stack frame, se sulla command line non è specificata l'opzione -k-. Si tornerà sull'argomento a pag. , con riferimento alla gestione dei dati nel Code Segment.

[159] La possibilità di dichiarare interrupt una funzione è offerta da molti compilatori C in commercio, soprattutto nelle loro versioni recenti.

di disastri, ancora una volta alle IRET o RET selvagge[160], e attenzione a quanto detto a pagina e seguenti (forse è bene dare un'occhiata anche al capitolo sui TSR, pag.).

Utilizzo dei registri

Torniamo alla funzione prova() di pag. : si è detto che eliminare l'istruzione RET, causa di problemi nella gestione dello stack, non è sufficiente per rimuovere ogni malfunzionamento: infatti il registro AH, caricato con il numero del servizio richiesto all'int 21h, viene azzerato dall'istruzione CBW, necessaria per "promuovere" var1 da char ad int, secondo le convenzioni C in materia di operazioni algebriche tra dati di tipi diversi. L'esempio non solo evidenzia una delle molteplici situazioni in cui il codice assembler generato dal compilatore nel tradurre le istruzioni C risulta in conflitto con lo inline assembly, ma fornisce lo spunto per alcune precisazioni in materia di registri.

Il registro AX è spesso utilizzato come variabile di transito per risultati intermedi[161] ed è quindi prudente, ove possibile, caricarlo immediatamente prima dell'istruzione che ne utilizza il contenuto (vedere anche pag.). La funzione prova(), a scanso di problemi, può essere riscritta come segue:

```
char prova(char var1,int var2)
{
    char result;

    result = var1*var2;
    asm mov dl,result;
    asm mov ah,2;
    asm int 21h;
}
```

Il registro AX non è il solo a richiedere qualche cautela: anche BX e CX sono spesso utilizzati in particolari situazioni, mentre DX è forse, tra i registri della CPU, il più "tranquillo"[162].

Qualche considerazione a parte meritano SI e DI[163]. Il compilatore garantisce che al ritorno da una funzione C essi conservano il valore che avevano al momento della chiamata: per tale motivo detti registri, se utilizzati nella funzione, vengono salvati sullo stack (PUSH) in testa al codice della funzione ed estratti dallo stesso (POP) in uscita, anche qualora SI e DI siano referenziati esclusivamente nell'ambito di righe inline assembly. Ecco un esempio:

```
void copia(char *dst,char *src,int count)
{
    asm {
        mov si,src;
        mov di,dst;
        mov cx,count;
```

[160] Per la verità, le funzioni interrupt ritornano mediante l'istruzione IRET. La RET può essere utilizzata solo nella forma RET 2, per eliminare la copia dei flag spinta sullo stack dalla chiamata ad interrupt.

[161] D'altra parte la "A" di AX sta per "Accumulatore". Esso è anche usato in modo implicito da alcune istruzioni, quali LODSB, LODSW, STOSB, STOSW.

[162] Ad esempio, BX (Base) può essere usato nel calcolo degli offset per referenziare i membri delle strutture; CX (Counter) agisce come contatore con l'istruzione LOOP e con quelle precedute da REP; DX (Data) è destinato a generiche funzionalità di archivio dati.

[163] Source Index e Destination Index. Utilizzati, tra l'altro, come indici ad incremento o decremento automatico dalle istruzioni MOVSB, MOVSW, etc..

```
    }
REPEAT:
    asm {
        lodsb;
        cmp al,0FFh;
        jne DO_LOOP;
        pop bp;
        ret;
    }
DO_LOOP:
    asm {
        stosb;
        loop REPEAT;
    }
}
```

La funzione presentata copia dall'array `src` all'array `dst` un numero di byte pari al valore della variabile intera `count`; se è incontrato un ASCII `255` (`FFh`), esso non è copiato e il controllo è restituito alla routine chiamante. Un rapido esame del codice assembler prodotto dal compilatore consente di verificare la fondatezza di quanto affermato:

```
....
_copia proc near
    push bp
    mov bp,sp
    push si
    push di
    mov si,[bp+6]
    mov di,[bp+4]
    mov cx,[bp+8]
@1@98:
    lodsb
    cmp al,0FFh
    jne short @1@242
    pop bp
    ret
@1@242:
    stosb
    loop short @1@98
    pop di
    pop si
    pop bp
    ret
_copia endp_copia endp
....
```

E' facile immaginare i problemi[164] che si verificano quando il byte letto da `src` è un ASCII `255`: il valore di `BP` all'uscita dalla funzione è, in realtà, quello che il registro `DI` presenta in entrata[165], mentre dovrebbe essere quello dello stesso `BP` in entrata.

Per ottenere il salvataggio automatico di `SI` e `DI` in ingresso alla funzione (ed il loro ripristino in uscita) è sufficiente dichiarare due variabili `register`:

[164] Coloro che sono privi di immaginazione sappiano che si tratta di problemi legati alla gestione dello stack, del tutto analoghi a quelli discussi con riferimento alla funzione `prova()`. Nella `copia()` non compaiono le istruzioni `DEC SP` e `MOV SP,BP` in quanto essa non fa uso di variabili locali.

[165] Si ricordi che lo stack è sempre movimentato con un algoritmo del tipo LIFO (Last In First Out): l'ultima word entrata con una `PUSH` è la prima ad esserne estratta con una `POP`.

```
#pragma warn -use

void function(void)
{
    register dummy_SI, dummy_DI;

    ....
    ....
}
```

Il trucchetto è particolarmente utile quando il codice inline assembly può modificare il contenuto di `SI` e `DI` in modo implicito, cioè senza memorizzare direttamente in essi alcun valore (ad esempio con una chiamata ad un interrupt che utilzzi internamente detti registri).

Per quanto riguarda il registro di segmento `DS`[166], è opportuno evitare di modificarlo, se non quando si sappia molto bene ciò che si sta facendo, e dunque si conosca l'impatto che tale modifica ha sul comportamento del codice: il problema può essere eliminato salvando sullo stack `DS` prima di modificarlo e ripristinandolo prima che esso sia referenziato da altre istruzioni C. Un esempio:

```
    ....
    char far *source, far *dest;
    char stringa;
    ....
    asm {
        lds si,source;
        les di,dest;
        mov cx,05h;
        rep movsw;
    }
    printf(stringa);
    ....
```

Nel frammento di codice presentato, `source` e `dest` sono puntatori `far`, mentre `stringa` è un puntatore `near` (ipotizzando di compilare per uno dei modelli tiny, small o medium). Il compilatore lo gestisce pertanto come un offset rispetto a `DS`: se il segmento del puntatore `far source` non è pari a `DS` la `printf()` non stampa `stringa`, bensì ciò che si trova ad un offset pari a `stringa` nel segmento di `source`. Il codice listato di seguito lavora correttamente:

```
    char far *source, far *dest;
    char stringa;
    ....
    asm {
        push ds;
        lds si,source;
        les di,dest;
        mov cx,05h;
        rep movsw;
        pop ds;
    }
    printf(stringa);
    ....
```

Considerazioni analoghe valgono per il registro `ES`[167], con la differenza che non è necessario ripristinarne il valore in uscita alla funzione che lo ha modificato, in quanto il compilatore C, contrariamente a quanto avviene per `DS`, non lo associa ad alcun utilizzo particolare.

---

[166] Data Segment: usato normalmente come puntatore al segmento dati di default del C.

[167] Extra Segment: il suo utilizzo è libero.

Variabili e indirizzi C

Le variabili C possono essere referenziate da istruzioni inline assembly, che hanno così modo di accedere al loro contenuto: le istruzioni

```
int Cvar;
....
asm mov ax,Cvar;
```

caricano in `AX` il contenuto di `Cvar`. Analogamente:

```
char byteVar;
....
asm mov al,byteVar;
```

caricano in `AL` il contenuto di `byteVar`. Tutto fila liscio, in quanto il tipo di dato C (o meglio: le dimensioni in bit dalle variabili C) sono coerenti con le dimensioni dei registri utilizzati; nel caso in cui tale coerenza non vi sia occorre tenere in considerazione alcune cosette. Riprendendo gli esempi precedenti, l'istruzione:

```
asm mov al,Cvar;
```

è perfettamente lecita. L'assemblatore conosce la dimensione di `AL` (ovvio!) e si regola di conseguenza, caricandovi uno solo dei due byte di `Cvar`. Quale? Quello "basso". Il motivo è evidente: il nome di una variabile può essere inteso, in assembler, come una sorta di puntatore[168] all'indirizzo di memoria al quale si trova il dato contenuto nella variabile stessa. Il nome `Cvar`, dunque, "punta" al primo byte (quello meno significativo, appunto) di una zona di tanti byte quanti sono quelli richiesti dal tipo di dato definito in C (si tratta di un integer, pertanto sono due[169]); in altre parole, l'istruzione commentata ha l'effetto di caricare in un registro della CPU tanti byte quanti sono necessari per "riempire" il registro stesso, (in questo caso C, dunque un solo byte) a partire dall'indirizzo della variabile C. Sorge il dubbio che, allora, l'istruzione:

```
asm mov ax,byteVar;
```

carichi in `AX` due byte presi all'indirizzo di `byteVar`, anche se questa è definita `char` nel codice C. Ebbene, è proprio così. In tali scomode situazioni occorre venire in aiuto all'assemblatore, che non conosce le definizioni C:

```
asm mov al,byteVar;
asm cbw;
```

lavora correttamente, caricando il byte di `byteVar` in `AL` e azzerando `AH`[170].

Le variabili a 32 bit (tipico esempio: i long integer e i puntatori `far`) devono essere considerate una coppia di variabili a 16 bit[171]. Supponiamo, per esempio, che la coppia di registri `DS:SI` punti ad un intero di nostro interesse; ecco come memorizzare detto indirizzo in un puntatore `far`:

---

[168] In C un evidente esempio è rappresentato dai nomi degli array.

[169] A dire il vero, i byte potrebbero essere quattro se il codice venisse compilato in modalità 80386 a 32 bit, ma ciò sarebbe, in questo caso, ininfluente.

[170] L'istruzione `CBW` (Convert Byte to Word) funziona solo con `AX`: la parte alta degli altri registri deve essere azzerata esplicitamente (ad es.: `XOR BH,BH` o `MOV BH,0`).

```
char far *varptr;
....
asm mov varptr,si;
asm mov varptr+2,ds;
```

La prima istruzione MOV copia SI nei due byte meno significativi di varptr (è la parte offset dell'indirizzo); la seconda copia DS nei due byte alti (la parte segment): non va dimenticato che i processori 80x86 memorizzano le variabili numeriche con la tecnica backwords, in modo tale, cioè, che a indirizzo di memoria inferiore corrisponda, byte per byte, la parte meno significativa della cifra. E se volessimo copiare il dato (non il suo indirizzo) referenziato da DS:SI in una variabile? Nell'ipotesi che DS:SI punti ad un long, la sequenza di istruzioni è la seguente:

```
long var32bits;
....
asm mov ax,ds:[si];
asm mov dx,ds:[si+2];
asm mov var32bits,ax;
asm mov var32bits+2,dx;
```

Si possono fare due osservazioni: in primo luogo, non è indispensabile usare due registri[172] (qui sono usati AX e DX) come "tramite", ma non è possibile muovere dati direttamente da un'indirizzo di memoria ad un altro (in questo caso dall'indirizzo puntato da DS:SI all'indirizzo di var32bits); inoltre non è stato necessario tenere conto del metodo backwords perché il dato a 32 bit è già in formato backwords all'indirizzo DS:SI.

Abbiamo così accennato ai puntatori: il discorso merita qualche approfondimento. I puntatori C, indipendentemente dal tipo di dato a cui puntano, si suddividono in due categorie: quelli near, che esprimono un offset relativo ad un registro di segmento (un indirizzo a 16 bit), e quelli far, che esprimono un indirizzo completo di segmento e offset (32 bit). I puntatori near, però, sono il default solo nei modelli di memoria tiny, small e medium (che d'ora in avanti chiameremo "piccoli"): negli altri modelli (compact, large e huge, d'ora in poi "grandi")[173] tutti i puntatori a dati sono far, anche se non dichiarati tali esplicitamente. La gestione dei puntatori C con lo inline assembly dipende dunque in modo imprescindibile dal modello di memoria utilizzato in compilazione. Vediamo subito qualche esempio.

```
/*
    modello "PICCOLO"
*/
    ....
    int *s_pointer;
    int *d_pointer;
    ....
/* s_pointer e d_pointer sono inizializzati, ad es. con malloc() */
    ....
    asm mov si,s_pointer;
```

---

[171] Eccetto il caso in cui il codice sia in grado di sfruttare le caratteristiche dell'assembler 80386: in tal caso sono disponibili i registri estesi (EAX, EBX, etc.) a 32 bit.

[172] Per la precisione, se ne può usare uno solo:

```
asm mov ax,ds:[si];
asm mov var32bits,ax;
asm mov ax,ds:[si+2];
asm mov var32bits+2,ax;
```

[173] Per i dettagli sui modelli di memoria vedere pag. 143.

```
    asm mov di,d_pointer;
    asm push ds;
    asm pop es;
    asm mov cx,4;
    asm rep movsw;
    ....
```

Il frammento di codice riportato copia 8 byte da `s_pointer` a `d_pointer`: dal momento che si è ipotizzato un modello di memoria "piccolo", questi sono entrambi puntatori `near` relativi a `DS`. In pratica, il contenuto di `s_pointer` è l'offset dell'area di memoria da cui si vogliono copiare i byte, e il contenuto di `d_pointer` è l'offset dell'area nella quale essi devono essere copiati: è necessario caricare `ES` con il valore di `DS` perché la `MOVSW` lavori correttamente.

```
/*
    modello "GRANDE"
*/
    ....
    int *s_pointer;
    int *d_pointer;
    ....
/* s_pointer e d_pointer sono inizializzati, ad es. con malloc() */
    ....
    asm push ds;
    asm lds si,s_pointer;
    asm les di,d_pointer;
    asm mov cx,4;
    asm rep movsw;
    asm pop ds;
    ....
```

Nei modelli "grandi" `s_pointer` e `d_pointer` sono, per default, puntatori `far`, pertanto è possibile usare le istruzioni `LDS` e `LES` per caricare registri di segmento e di offset. Ciò vale anche per puntatori dichiarati `far` nei modelli "piccoli":

```
/*
    modello "PICCOLO"
*/
    ....
    int far *s_pointer;
    int far *d_pointer;
    ....
/* s_pointer e d_pointer sono inizializzati, ad es. con farmalloc() */
    ....
    asm push ds;
    asm lds si,s_pointer;
    asm les di,d_pointer;
    asm mov cx,4;
    asm rep movsw;
    asm pop ds;
    ....
```

ALTRI STRUMENTI DI PROGRAMMAZIONE MISTA

Molti compilatori offrono supporto a strumenti che consentono il controllo a basso livello delle risorse del sistema senza il ricorso diretto al linguaggio assembler.

P s e u d o r e g i s t r i

Gli pseudoregistri sono identificatori che consentono di manipolare direttamente da istruzioni C i registri della CPU (compreso il registro dei flag). Per quel che riguarda il compilatore C Borland, essi sono implementati intrinsecamente: pertanto non è possibile portare il codice che li utilizza ad altri compilatori che non li implementino in maniera analoga. Va precisato che gli pseudoregistri non sono variabili, ma, come accennato, semplicemente identificatori che consentono al compilatore di generare le opportune istruzioni assembler (essi non hanno dunque un indirizzo referenziabile da puntatori): ad esempio le istruzioni C

```
    _AX = 9;
    _BX = _AX;
```

producono le istruzioni assembler

```
    mov ax,9
    mov bx,ax
```

come, del resto, ci si può attendere.

Non sempre, però, un'istruzione C contenente un riferimento ad uno pseudoregistro genera una singola istruzione assembler: vediamo un esempio.

```
    ....
    if(!_AH)
        if(_AL == _BL)
            _AL = _CL;
    ....
```

Il compilatore, a partire dal frammento di codice riportato, genera il seguente listato assembler:

```
    ....
    mov al,ah
    mov ah,0
    or  ax,ax
    jne short @1@74
    cmp al,bl
    jne short @1@74
    mov al,cl
@1@74:
    ....
```

Come si vede, il valore di `AL` viene modificato per effettuare il primo test, con la conseguenza di invalidare i risultati del confronto successivo. Il ricorso allo inline assembly può evitare tali pasticci (e consentire la scrittura di codice più efficiente):

```
    ....
    asm {
        or ah,ah;
        jne NO_CHANGE;
        cmp al,bl;
        jne NO_CHANGE;
        mov al,cl;
    }
NO_CHANGE:
    ....
```

Quando vengono assegnati valori ai registri di segmento il compilatore deve tenere conto dei limiti alla libertà di azione imposti, in questi casi, dalle regole sintattiche del linguaggio assembler. Vediamo un esempio:

```
_ES = 0xB800;
```

è tradotta in

```
mov ax,0B800h
mov es,ax
```

e anche

```
_DS = _ES;
```

produce due istruzioni assembler:

```
mov ax,es
mov ds,ax
```

che hanno, tra l'altro, l'effetto di modificare il contenuto di `AX`; programmando direttamente in assembler (o con l'inline assembly) si può assegnare `ES` a `DS`, senza rischio alcuno di modificare `AX`, con le due istruzioni seguenti:

```
push es
pop ds
```

Analoghe osservazioni sono valide per operazioni coinvolgenti lo pseudoregistro dei flag. L'istruzione C

```
_FLAGS |= 1;
```

produce la sequenza di istruzioni assembler riportata di seguito:

```
pushf
pop ax
or ax,1
push ax
popf
```

Si noti che programmando direttamente in assembler (o inline assembly), sarebbe stato possibile ottenere il risultato desiderato (`CarryFlag = 1`) con una sola istruzione[174]:

```
stc
```

Vale infine la pena di richiamare l'attenzione sul fatto che l'utilizzo degli pseudoregistri può condurre ad un impiego "nascosto" di registri apparentemente non coinvolti nell'operazione: negli esempi appena visti il registro `AX` viene usato all'insaputa del programmatore; sull'argomento si veda pag. .

[174] Il compilatore ha generato una sequenza valida per ogni bit dei flag (non esiste una specifica istruzione assembler per ciascuno di essi).

geninterrupt()

La `geninterrupt()` è una macro basata sulla funzione intrinseca `__int__()` (la portabilità è perciò praticamente nulla) e definita in `DOS.H`. Essa invoca l'interrupt il cui numero le è passato come argomento; in pratica ha un effetto equivalente a quello di una istruzione `INT` inserita nel codice C tramite l'inline assembly. I registri devono essere gestiti tramite l'inline assembly medesimo o mediante gli pseudoregistri.

__emit__()

La `__emit__()` è una funzione intrinseca del compilatore C Borland: ciò implica l'impossibilità di portare ad altri compilatori che non la implementino i programmi che ne fanno uso. Nella forma più semplice di utilizzo i suoi argomenti sono byte, che vengono inseriti dal compilatore direttamente nel codice oggetto prodotto, senza che sia generato il codice relativo ad una reale chiamata a funzione. Come si comprende facilmente, `__emit__()` va oltre lo inline assembly, consentendo una vera e propria forma di programmazione in linguaggio macchina. Un esmpio:

```
#pragma option -k-

void boot(void)
{
    __emit__(0xEA,(unsigned)0x00,0xFFFF);
}
```

La funzione `boot()`, quando eseguita, provoca un bootstrap[175]. Il codice macchina prodotto è il seguente (byte esadecimali):

```
EA 00 00 FF FF C3
```

I primi quattro byte rappresentano l'istruzione `JMP FAR` seguita dall'indirizzo (standard; `FFFF:0000`) al quale, nel BIOS, si trova l'istruzione di salto all'effettivo indirizzo della routine BIOS dedicata al bootstrap. Il quinto byte è la `RET` (peraltro mai eseguita, in questo caso) che chiude la funzione. La `#pragma` evita la gestione, evidentemente inutile, della standard stack frame (pag.). Da sottolineare: l'assemblatore non accetta l'istruzione

```
JMP FAR 0FFFFh:0h
```

ed emette un messaggio di errore del tipo "indirizzamento diretto illecito". La `__emit__()` permette, in questo caso, di ottimizzare il codice evitando il ricorso ad un puntatore contenente l'indirizzo desiderato.

UNO STRATAGEMMA: DATI NEL CODE SEGMENT

Lo scrivere programmi in una sorta di linguaggio misto "C/Assembler/codice macchina" mette a disposizione possibilità realmente interessanti: vediamone una utile in diverse intricate situazioni.

Di norma, nei modelli di memoria "piccoli" (tiny, small, medium) lo spazio necessario alle variabili globali è allocato ad offset relativi a `DS`: ciò significa che l'accesso ad ogni variabile globale

---

[175] E' un cold bootstrap se prima di invocare la `boot()` il programma non scrive il numero `1234h` all'indirizzo RAM `0:0472`.

utilizza DS come punto di riferimento. Ciò avviene in maniera trasparente al programmatore, ma vi sono casi in cui non è possibile fare affidamento su detto registro.

Una tipica situazione è quella dei gestori di interrupt (di cui si parla e straparla a pag. e seguenti): questi entrano in azione in un contesto non conoscibile a priori, pertanto nulla garantisce (e infatti raramente accade) che DS punti proprio al segmento dati del programma in cui il gestore è definito; CS è l'unico registro che in entrata ad un interrupt assuma sempre il medesimo valore[176]. In casi come quello descritto è pratica prudente ed utile, a scanso di problemi, allocare le variabili esterne alla funzione ad indirizzi relativi a CS.

Un metodo per raggiungere lo scopo, benché un poco macchinoso (come al solito!), consiste nel dichiarare, collocandola opportunamente nel sorgente, una funzione fittizia, il cui codice non rappresenti istruzioni, bensì i dati (generalmente variabili globali) che dovranno essere referenziati mediante il registro CS[177].

Come fare? All'interno della funzione fittizia i dati non possono essere dichiarati come variabili esterne, perché mancherebbe comunque la dichiarazione globale, né come variabili statiche, perché subirebbero pressappoco la medesima sorte dei dati globali, né, tantomeno, come variabili automatiche, perché esisterebbero (nello stack) solamente durante l'esecuzione della funzione fittizia, la quale, proprio in quanto tale, non è mai eseguita (del resto, anche se venisse eseguita, non sarebbe comunque possibile risolvere i problemi legati alla visibilità delle variabili locali). E' necessario ricorrere allo inline assembly, che consente l'inserimento diretto di costanti numeriche nel codice oggetto in fase di compilazione; il nome della funzione fittizia, grazie ad opportune operazioni di cast, può essere utilizzato in qualunque parte del codice come puntatore ai dati in essa generati[178].

Supponiamo, ad esempio, di voler definire un puntatore a funzione interrupt e un intero senza segno:

```
void Jolly(void)
{
    asm db 5 dup (0);                            // genera 5 bytes inizializzati a 0
}

#define OldIntVec  ((void (interrupt *)())(*((long *)Jolly)))
#define UnsIntegr  (*(((unsigned int *)Jolly)+2))
```

I dati da noi definiti occupano, complessivamente, 6 byte: tale deve essere l'ingombro minimo, in termini di codice macchina, della funzione Jolly(). Con lo inline assembly è però sufficiente definire un byte in meno dello spazio richiesto dai dati, in quanto il byte mancante è, in realtà, rappresentato dall'opcode dell'istruzione RET (C3), generata dal compilatore in chiusura della funzione, il quale può essere sovrascritto senza alcun problema: la Jolly() non viene mai eseguita[179].

Le macro definite dalle direttive #define eliminano la necessità di effettuare complessi cast quando si utilizzano, nel codice, i dati contenuti in Jolly(): si può fare riferimento ad essi come a

[176] Infatti esso punta al code segment del gestore di interrupt; in altre parole esso è la parte segmento del vettore, cioè dell'indirizzo, della routine.

[177] Meglio non fidarsi dell'opportunità, offerta da alcuni compilatori, di definire nel sorgente variabili indirizzate relativamente a CS: a differenza del metodo descritto in queste pagine, l'opzione citata non consente di controllare *dove*, all'interno del code segment, tali variabili saranno allocate.

[178] In realtà le funzioni potrebbero essere più di una, e ciascuna potrebbe contenere uno o più dati. Il programmatore è naturalmente libero di scegliere l'approccio che gli è più congeniale.

[179] Come si è anticipato, quello descritto è un trucco per memorizzare dati in locazioni relative a CS e non a DS, contrariamente al default del compilatore.

variabili globali dichiarate nel modo tradizionale. Infatti `OldIntVec` rappresenta, a tutti gli effetti, un puntatore a funzione di tipo `interrupt`: il nome della `Jolly()`, che è implicitamente puntatore alla funzione medesima, viene forzato a puntatore a `long` (in pratica, puntatore a un dato a 32 bit), la cui indirezione (il valore contenuto all'indirizzo `CS:Jolly`) è a sua volta forzata a puntatore a funzione `interrupt`. La seconda macro impone al compilatore di considerare `Jolly` quale puntatore a intero senza segno; sommandovi 2 si ottiene l'indirizzo `CS:Jolly+4` (il compilatore somma in realtà quattro a `Jolly` proprio perché si tratta, in seguito al cast, di puntatore ad intero), la cui indirezione è un unsigned integer, rappresentato da `UnsIntegr`[180].

Una precisazione, a scanso di problemi: può accadere di dover fare riferimento con istruzioni inline assembly ai dati globali gestiti nella `Jolly()`. E' evidente che in tale caso le macro descritte non sono utilizzabili, in quanto, espanse dal preprocessore, diverrebbero parti in C di istruzioni assembler, con la conseguenza che l'assemblatore non sarebbe in grado di compilare l'istruzione così costruita. Ad esempio:

```
#define integer1    (*((int *)Jolly))
#define integer2    (*(((int *)Jolly)+1))
#define new_handler ((void (interrupt *)())(*(((long *)Jolly)+1)))

void Jolly(void)
{
    asm dw 0;
    asm dw 1;
    asm dd 0;
}

void interrupt new_handler(void)
{
    ....
    asm {
        pushf;
        call dword ptr new_handler;
    }
    ....
}
```

Il gestore di interrupt `new_handler()` utilizza il vettore del gestore originale per invocare quest'ultimo[181]; il sorgente assembler risultante contiene le righe seguenti:

```
....
PUSHF
CALL DWORD PTR ((void (interrupt *)())(*(((long *)Jolly)+1)))
....
```

Sicuramente la `CALL` non è assemblabile: la macro `new_handler` può essere utilizzata solo nell'ambito di istruzioni in linguaggio C. Con lo inline assembly è necessario fare riferimento al nome della funzione fittizia sommandovi l'offset, in byte, del dato che si intende referenziare. Come in precedenza, una macro può semplificare le operazioni:

```
#define integer1    (*((int *)Jolly))
#define integer2    (*(((int *)Jolly)+1))
```

[180] Sembra incredibile, ma funziona.

[181] Niente paura: simili follie sono analizzate in profondità con riferimento alla gestione degli interrupt (pag. 251) ed ai TSR (pag. 275). Gli esempi qui riportati vanno esaminati semplicemente dal punto di vista della sintassi.

```
#define new_handler ((void (interrupt *)())(*(((long *)Jolly)+1)))

#define ASM_handler Jolly+4

void Jolly(void)
{
    asm dw 0;
    asm dw 1;
    asm dd 0;
}

void interrupt new_handler(void)
{
    ....
    asm {
        pushf;
        call dword ptr ASM_handler;
    }
    ....
}
```

L'espansione della macro, questa volta, genera la riga seguente, che può essere validamente compilata dall'assemblatore:

```
....
CALL DWORD PTR Jolly+4
....
```

Per evitare un proliferare incontrollato di direttive `#define`, si può definire una funzione fittizia per ogni variabile da simulare:

```
void vectorPtr(void)
{
    asm db 3 dup(0);
}

void unsigednVar(void)
{
    asm db 0;
}
```

In tal modo le istruzioni assembly potranno contenere riferimenti diretti ai nomi delle funzioni fittizie:

```
....
    asm mov ax,word ptr unsignedVar;
    asm pushf;
    asm call dword ptr vectorPtr;
....
```

Sfortunatamente, le differenze esistenti tra versioni successive del compilatore introducono alcune complicazioni: gli esempi riportati sono validi sia per TURBO C 2.0 che per TURBO C++ 1.0 e successivi, ma occorrono alcune precisazioni.

Le versioni C++ 1.0 e successive del compilatore (a differenza di TURBO C 2.0) inseriscono per default i tre opcode corrispondenti alle istruzioni `PUSH BP` e `MOV BP,SP` in testa al codice di ogni funzione e, di conseguenza, quello dell'istruzione `POP BP` prima della `RET` finale, anche quando la funzione stessa sia dichiarata `void` e priva di parametri formali, come nel caso della `Jolly()`: questo significa che l'ingombro minimo di una funzione è 5 byte (`55h,8Bh,ECh` e `5Dh,C3h`). Nel caso

esaminato è allora sufficiente definire un solo byte aggiuntivo per riservare tutto lo spazio necessario ai dati utilizzati:

```
void Jolly(void)
{
    asm db 0;                               // genera 1 byte inizializzato a 0
}
```

Si noti che almeno un byte deve essere comunque riservato mediante lo inline assembly, anche qualora i 5 byte di ingombro minimo della funzione siano sufficienti a contenere tutti i dati necessari: in caso contrario il compilatore non interpreta come desiderato il codice e gestisce l'offset rispetto a CS della funzione fittizia come offset rispetto a DS, con la conseguenza di vanificare tutto l'artificio, ed il rischio di obliterare selvaggiamente il segmento dati. Inoltre, se si desidera inizializzare i dati direttamente con le istruzioni DB[182], non è possibile sfruttare i byte di codice generati dal compilatore, e può essere necessario inserire dei byte a "tappo" per semplificare la gestione dei cast. Per definire un long integer inizializzato al valore 0x12345678 occorre regolarsi come segue:

```
void Jolly(void)
{
    asm db 0;                                                    // tappo
    asm dd 0x12345678;
}

#define LongVar  (*(((long *)Jolly)+1))
```

Il byte tappo consente di sfruttare l'aritmetica dei puntatori: in sua assenza, la macro sarebbe risultata necessariamente più complessa:

```
void Jolly(void)
{
    asm dd 0x12345678;
}

#define LongVar  (*((long *)(((char *)Jolly)+3)))
```

Solo i puntatori a dati di tipo char, infatti, ammettono un offset di un numero dispari di byte rispetto all'indirizzo di base (si ricordi che TURBO C++ 1.0 e successivi generano tre opcode in testa alla funzione).

Va rilevato, infine, che il compilatore, quando si utilizza il modello di memoria huge, genera un'istruzione PUSH DS subito dopo la MOV BP,SP e, di conseguenza, una POP DS immediatamente prima della POP BP: di ciò bisogna ovviamente tenere conto.

Da quanto ora evidenziato derivano problemi di portabilità, che possono però essere risolti con poco sforzo. Il compilatore definisce automaticamente alcune macro, una delle quali può essere utilizzata per scrivere programmi che, pur utilizzando la tecnica di gestione dei dati globali sopra descritta, risultino portabili tra le diverse versioni del compilatore stesso e possano quindi essere compilati senza necessità di modifiche al sorgente.

La macro predefinita in questione è __TURBOC__: essa rappresenta una costante esadecimale intera senza segno che corrisponde alla versione di compilatore utilizzata. Ad esempio, il programma seguente visualizza numeri differenti se compilato con differenti versioni del compilatore:

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

---

[182] Oltre a DB (Define Byte) sono ammesse anche DW (Define Word) e DD (Define Doubleword), nonché l'estensione DUP, circa la quale è meglio vedere pag. 157, in nota: programmatore avvisato...

```
void main(void)
{
    int Version, Revision;

    Version = __TURBOC__ >> 8;
    Revision = __TURBOC__ & 0xFF;
    printf("Versione del Compilatore TURBO C: %02X.%02X\n",Version,Revision);
}
```

La tabella che segue è riportata per comodità.

VALORI DELLA MACRO `__TURBOC__`

VALORE MACRO	VERSIONE	REVISIONE	COMPILATORE
0x0001	00	01	TURBO C 1.00
0x0200	02	00	TURBO C 2.00
0x0295	02	95	TURBO C++ 1.00
0x0296	02	96	TURBO C++ 1.01
0x0297	02	97	BORLAND C++ 2.00
0x0410	04	10	BORLAND C++ 3.1

Infine, riprendiamo uno dei precedenti esempi applicandovi la tecnica di compilazione condizionale:

```
void Jolly(void)
{
#if __TURBOC__ >= 0x0295
    asm db 0;                                                      // tappo
#endif
    asm dd 0x12345678;
}

#if __TURBOC__ >= 0x0295
#define LongVar  (*(((long *)Jolly)+1))
#else
#define LongVar  (*((long *)Jolly))
#endif
```

Una seconda via, ancora più semplice, per aggirare l'ostacolo consiste nello specificare, quando si compili con TURBO C++ 1.0 o successivi, l'opzione `-k-` sulla command line (o inserire la direttiva `#pragma option -k-`) in testa al codice sorgente: essa evita la creazione di una struttura standard di stack per tutte le funzioni (standard stack frame) e pertanto le funzioni `void` prive di parametri vengono compilate come avviene per default con TURBO C 2.0 (e versioni precedenti).

Va ancora sottolineato che definire una funzione fittizia per ogni variabile (e attivare sempre l'opzione `-k-`) consente di aggirare le difficoltà cui si è fatto cenno.

Per ulteriori approfondimenti in tema di funzioni fittizie utilizzate quali contenitori di dati si vedano le pagg. , e .

C E CLIPPER

Clipper è un linguaggio compilato, sintatticamente compatibile in larga misura con l'interprete del dBase III, orientato al database management. Sin dalle prime versioni, Clipper ha implementato gli strumenti necessari all'interfacciamento con il C per la realizzazione di funzioni non presenti nelle sue librerie standard (soprattutto per la gestione della macchina a basso livello).

A tal fine è raccomandato l'utilizzo del Microsoft C (del resto le librerie Clipper sono scritte in Microsoft C); in realtà tutti i compilatori C in grado di generare, per il large memory model, moduli oggetto compatibili con l'architettura DOSSEG definita da Microsoft possono essere validamente impiegati in molti casi[183]. Il Microsoft C è indispensabile, per motivi di compatibilità, solo nel caso in cui si vogliano realizzare funzioni implementanti matematica in virgola mobile o routine grafiche[184].

Le funzioni scritte in C devono quindi essere compilate per il modello di memoria large[185] (pag. 143 e seguenti) e, se si utilizza il compilatore Microsoft, deve essere richiesta l'opzione "Floating Point Alternate Library" (`/FP`); è inoltre molto comodo includere nel sorgente il file `EXTEND.H`, fornito con il pacchetto Clipper, che definisce alcune macro e costanti manifeste e contiene i prototipi di tutte le funzioni, facenti parte nella libreria Clipper, che consentono lo scambio di parametri e valori restituiti tra le funzioni Clipper e quelle C. Il modulo oggetto risultante dalla compilazione può essere collegato all'object file prodotto da Clipper, oppure può essere inserito in una libreria.

PASSAGGIO DI PARAMETRI E RESTITUZIONE DI VALORI

Nello scrivere funzioni C richiamabili da programmi Clipper va tenuto sempre presente che non ha senso, per esigenze di compatibilità tra i due linguaggi, parlare di parametri formali (pag. 87 e seguenti). Le funzioni C possono accedere a tutti i parametri attuali della chiamata Clipper invocando le funzioni `_par...()` (facenti parte della libreria Clipper e dichiarate in `EXTEND.H`) e devono pertanto essere dichiarate prive di parametri formali (`void`). Vi sono numerose funzioni `_par...()`, ciascuna dedicata ad un particolare tipo di parametro.

Anche l'eventuale restituzione di un valore alla routine Clipper deve avvenire mediante le funzioni `_ret...()`, facenti parte della libreria Clipper e tra loro diversificate in base al tipo del valore da restituire. Le funzioni `_ret...()` non restituiscono il controllo alla routine Clipper, ma si limitano a predisporre la restituzione del valore; la funzione C cede il controllo solo al termine del proprio codice o eseguendo un'istruzione `return`). Ne segue che le funzioni C devono essere dichiarate `void`; inoltre le specifiche Clipper impongono che esse siano dichiarate anche `pascal` (pag. 92). Il file `EXTEND.H` definisce la macro `CLIPPER`, che può essere usata per dichiarare le funzioni C ed equivale proprio a `void pascal`.

---

[183] I compilatori Borland e Zortech, ad esempio, soddisfano detto requisito, ma, in alcune situazioni, la struttura delle librerie, ove differente da quella implementata da Microsoft, può essere causa di problemi.

[184] Come al solito, occorre fare attenzione. E' vero che le librerie Clipper sono scritte in Microsoft C, ma va precisato che si tratta della versione 5.1. Non è un particolare irrilevante, perché, pur utilizzando il compilatore Microsoft per scrivere funzioni C da interfacciare a Clipper, possono sorgere strani problemi se la versione usata non è, anch'essa, la 5.1. In alcuni casi è possibile utilizzare la `GRAPHICS.LIB` del Microsoft C 6.0 con la `LLIBCA.LIB` del Microsoft C 5.1, ma si tratta comunque di un pasticcio pericoloso.

[185] Con il Microsoft C viene perciò impiegata la libreria `LLIBCA.LIB`, mentre con il Borland la libreria usata è la `CL.LIB`.

Il prototipo di una funzione C richiamabile da Clipper è perciò analogo a[186]:

```
#include <EXTEND.H>
....
CLIPPER funzPerClipper(void);
```

Nell'ipotesi che `funzPerClipper()` accetti 4 parametri, la chiamata in Clipper è:

```
RETVAL = FUNZPERCLIPPER(PAR_1,PAR_2,PAR_3,PAR_4)
```

Si tratta ora di analizzare brevemente le funzioni `_par...()` e `_ret...()`, per scoprire come `funzPerClipper()` può accedere a `PAR_1`, `PAR_2`, `PAR_3` e `PAR_4` e restituire `RETVAL`.

La funzione

```
int _parinfo(int order);
```

restituisce il tipo del parametro che occupa la posizione `order` nella lista dei parametri attuali; `_parinfo(0)` restituisce il numero di parametri passati alla funzione. Il tipo è identificato dalle seguenti costanti manifeste, definite in `EXTEND.H`:

```
#define UNDEF      0
#define CHARACTER 1
#define NUMERIC   2
#define LOGICAL   4
#define DATE      8
#define MPTR     32         /* sommato al tipo effettivo se passato per reference */
#define MEMO     65
#define ARRAY   512
```

La costante `MPTR` ha un significato particolare: essa è sommata al tipo del parametro per indicare che esso è stato passato come *reference* (preceduto, secondo la sintassi Clipper, da una `@`): perciò, se `_parinfo(1)` restituisce `33`, significa che il primo parametro attuale è una stringa passata per reference.

La funzione

```
int _parinfa(int order,int index);
```

restituisce il tipo dell'elemento in posizione `index` nell'array che a sua volta è parametro attuale di posizione `order`. Così, se `_parinfa(2,5)` restituisce `1`, significa che il quinto elemento dell'array ricevuto come secondo parametro attuale è una stringa[187]. La chiamata

```
_parinfa(order,0);
```

restituisce il numero di elementi dell'array che occupa la posizione `order` tra i parametri attuali. Il secondo parametro (`index`) può essere omesso: in tal caso `_parinfa()` equivale a `_parinfo()`[188].

---

[186] Si presti però attenzione al fatto che Clipper è un linguaggio case-insensitive: ciò non pone problemi, a patto di non differenziare mai nomi nei sorgenti C solamente in base alle maiuscole/minuscole.

[187] In Clipper gli elementi dell'array sono numerati a partire da `1` e non da `0` come in C. Inoltre, ogni elemento di un array può appartenere ad un tipo differente da quello degli altri.

[188] L'extend system implementato dal Clipper 5 include alcune macro, dichiarate in `EXTEND.H`, utilizzabili, in luogo di `_parinfo()` e `_parinfa()`, per il controllo dei parametri passati alla funzione: la macro

Vediamo ora una rapida rassegna delle altre principali funzioni `_par...()`. Per i dettagli sintattici si rimanda alla documentazione del linguaggio Clipper; va detto, tuttavia, che nell'uso di tali funzioni il secondo parametro, `index`, può essere omesso: in particolare, esso deve essere utilizzato solo quando il parametro attuale la cui posizione è espressa dal primo parametro `order` sia un array. In tal caso, `index` individua uno specifico elemento all'interno dell'array medesimo.

La funzione

```
int _parni(int order,int index);
```

"recupera" il parametro attuale di posizione `order` (se esso è un array è possibile specificarne un singolo elemento con `index`; in caso contrario `index` può essere omesso) e lo restituisce sotto forma di intero: ad esempio, la chiamata

```
#include <extend.h>
     ....
     int parm_1;
     ....
     parm_1 = _parni(1);
     ....
```

consente di accedere al primo parametro, un integer, passato dalla routine Clipper alla funzione C. Del tutto analoghe alla `_parni()` sono le funzioni

```
long   _parnl(int order,int index);
double _parnd(int order,int index);
int    _parl(int order,int index);
char  *_pards(int order,int index);
char  *_parc(int order,int index);
```

Tutte, infatti, restituiscono il parametro attuale di posizione `order` (con la possibilità, se esso è un array, di individuarne un elemento mediante `index`). La `_parnl()` restituisce un `long` e la `_parnd()` un `double`, mentre la `_parl()` restituisce un intero che rappresenta un campo logico Clipper (`0` equivale a `.F.` e `1` a `.T.`).

```
PCOUNT
```

restituisce il numero di parametri che la funzione C ha ricevuto. La macro

```
ALENGTH(int parmno)
```

accetta il numero d'ordine del parametro ricevuto dalla funzione C e, se questo è un array, restituisce il numero di elementi che lo costituiscono. Infine, le macro elencate di seguito sono concepite per controllare il tipo del parametro la cui posizione è loro passata in input:

```
ISARRAY(int parmno);      // restituisce non-zero se il parametro parmno e' un array
ISBYREF(int parmno);// restit. non-zero se parametro parmno e' passato per reference
ISCHAR(int parmno);    // restituisce non-zero se il parametro parmno e' una stringa
ISDATE(int parmno);       // restituisce non-zero se il parametro parmno e' una data
ISLOG(int parmno); // restituisce non-zero se il parametro parmno e' un campo logico
ISMEMO(int parmno);  // restituisce non-zero se il parametro parmno e' un campo memo
ISNUM(int parmno);     // restituisce non-zero se il parametro parmno e' un numerico
```

Il valore minimo che il parametro `parmno` può assumere è `1` e indica il primo parametro passato alla funzione C.

La `_pards()` restituisce una stringa derivata da un campo Clipper di tipo *date*; la data è espressa nella forma `"AAAAMMGG"`. E' opportuno copiare la stringa restituita in un buffer appositamente allocato, in quanto il buffer gestito dalla funzione `_pards()` è statico e viene sovrascritto ad ogni chiamata alla stessa. Ad esempio, nel codice

```
#include <stdio.h>
#include <extend.h>
    ....
    char *date_1, date_2;
    ....
    date_1 = _pards(1);
    date_2 = _pards(2);
    printf("data 1 = %s\ndata 2 = %s\n",date_1,date_2);
```

la `printf()` visualizza due volte la data passata come secondo parametro, perché la seconda chiamata a `_pards()` sovrascrive il buffer statico che essa utilizza internamente. Per ottenere un funzionamento corretto è sufficiente apportare la modifica seguente:

```
....
#include <string.h>
    ....
    strcpy(date_1,_pards(1));
    strcpy(date_2,_pards(2));
    ....
```

La funzione `_parc()` restituisce un puntatore a carattere. Va però ricordato che la gestione delle stringhe in Clipper è differente da quella implementata in C: in particolare, le stringhe possono contenere anche caratteri nulli (lo zero binario). Per conoscere la lunghezza effettiva di una stringa Clipper può essere utilizzata la funzione

```
int _parclen(int order,int index);
```

che non include nel computo il `NULL` che chiude la stringa stessa, mentre la

```
int _parcsiz(int order,int index);
```

restituisce il numero di byte effettivamente allocati per contenere la stringa (o, più in generale, l'array di caratteri), incluso l'eventuale `NULL` che la chiude. Se il parametro è una costante stringa, `_parcsiz()` restituisce `0`.

Passiamo ora ad un rapido esame delle funzioni `_ret...()`, che consentono alla funzione C di restituire un valore alla routine Clipper, con la precisazione che, essendo possibile restituire un unico valore, la funzione C può chiamare una sola volta una funzione `_ret...()` prima di terminare.

E' il caso di citare per prima una funzione leggermente particolare: si tratta della

```
void _ret(void);
```

che non restituisce a Clipper alcun valore (esattamente come se la funzione C terminasse con una semplice `return`), ma gestisce lo stack in modo tale che la routine Clipper possa chiamare la funzione C con un'istruzione `DO` (come se fosse, cioè, una *procedure* e non una *function*).

Vi è poi una `_ret...()` per ogni tipo di dato, analogamente a quanto visto per le funzioni `_par...()`:

```
void _retni(int ival);
void _retnl(long lval);
void _retnd(double dval);
```

```
void _retl(int bval);
void _retds(char *datestr);
void _retc(char *string);
```

La `_retni()` restituisce alla routine Clipper, sotto forma di campo numerico, l'intero che le è passato come parametro. Del tutto analoghe sono la `_retnl()` e la `_retnd()`, che devono essere utilizzate per restituire, rispettivamente, un `long` e un `double`.

La `_retl()` restituisce a Clipper, sotto forma di campo logico, l'intero che riceve come parametro. Questo deve rappresentare un valore booleano, cioè deve essere `1` o `0`, convertiti rispettivamente in `.T.` e `.F.`.

La `_retds()` restituisce sotto forma di campo *date* la stringa passatale come parametro, che deve essere nella forma `"AAAAMMGG"`.

La `_retc()` restituisce a Clipper il puntatore a carattere (stringa) che le è passato come parametro. Alla `_retc()` si affianca la

```
void _retclen(char *buffer,int len);
```

che restituisce al Clipper, oltre all'indirizzo della stringa, anche la lunghezza della medesima. La `_retclen()` si rivela utile nei casi in cui il puntatore a carattere indirizza un buffer contenente anche byte nulli (zeri binari).

REFERENCE E PUNTATORI

Con Clipper Summer '87 non esiste modo per passare ad una funzione C l'indirizzo di una variabile Clipper (cioè il puntatore ad essa): ciò avviene, ma implicitamente, solo nel caso delle stringhe. La libreria Clipper 5 include invece alcune funzioni, le `_stor...()`, che consentono al codice C di accedere alle variabili ricevute per *reference*, cioè passate come parametri attuali anteponendo al nome il carattere `@`, in modo analogo a quello realizzabile (in C puro) mediante il passaggio di puntatori. La documentazione Clipper sottolinea che passare per reference una variabile ad una funzione C non significa comunque passarne l'indirizzo: in realtà la gestione del passaggio di parametri è, in Clipper, piuttosto complessa; tuttavia l'utilizzo dei reference comporta l'inserimento nello stack di informazioni relative all'indirizzo della variabile, ed è proprio grazie a tali informazioni che le nuove funzioni cui si è accennato consentono di "simulare" l'impiego di veri e propri puntatori.

La funzione

```
int _storni(int n,int order,int index);
```

memorizza l'intero `n` nella variabile Clipper passata via reference alla funzione C come parametro di posto `order`; il terzo parametro è facoltativo e deve essere utilizzato solo nel caso in cui il parametro ricevuto in posizione `order` sia un array: in tal caso `index` individua un preciso elemento al suo interno. La funzione restituisce `0` in caso di errore, `1` altrimenti.

Analoghe alla `_storni()` sono le funzioni

```
int _stornd(double n,int order,int index);
int _stornl(long n,int order,int index);
int _storl(int logical,int order,int index);
int _stords(char *string,int order,int index);
int _storc(char *string,int order,int index);
```

La `_stornd()` è utilizzabile per memorizzare un valore come `double`, mentre la `_stornl()` memorizza un long integer. La `_storl()` accetta come primo parametro un intero e lo memorizza come valore logico (`0` equivale a `.F.`; un valore diverso da `0` equivale a `.T.`); la

`_stords()` memorizza come data la stringa `string`, che deve avere formato `"AAAAMMGG"`. La `_storc()` memorizza una stringa; in alternativa può essere utilizzata la

```
int _storclen(char *buffer,int length,int order,int index);
```

che consente di memorizzare un array di caratteri indicandone la lunghezza[189].

ALLOCAZIONE DELLA MEMORIA

Le funzioni C destinate ad interfacciarsi con Clipper non possono gestire l'allocazione dinamica della memoria mediante `malloc()` e le altre funzioni della libreria C allo scopo predisposte (pag. 109): per esigenze di compatibilità è necessario utilizzare due funzioni che la libreria Clipper Summer '87 rende disponibili, ancora una volta, tramite i prototipi dichiarati in `EXTEND.H`. In particolare, la funzione

```
unsigned char *_exmgrab(unsigned int size);
```

alloca in modo compatibile con Clipper un buffer ampio `size` byte e ne restituisce l'indirizzo sotto forma di puntatore a `unsigned char`[190]. In caso di errore viene restituito `NULL`. L'analogia con `malloc()` è evidente. La memoria allocata da `_exmgrab()` può essere gestita con le comuni tecniche C (puntatori, indirezioni, etc.); tuttavia essa deve essere disallocata con l'apposita funzione Clipper:

```
void _exmback(unsigned char *pointer,unsigned int size);
```

che svolge un ruolo analogo a quello della funzione C `free()`; a differenza di questa, però, `_exmback()` richiede che, oltre all'indirizzo dell'area di memoria da disallocare, le sia passato anche il numero di byte da liberare. Se si intende disallocare l'intera area di RAM, il parametro `size` passato a `_exmback()` deve essere identico a quello omologo passato a `_exmgrab()`. Vediamo un semplice esempio:

```
#include <extend.h>
....
    unsigned size = 1000;
    unsigned char *buffer;
    ....
    if(!(buffer = _exmgrab(size))) {
        ....                                                // gestione errore
    }
    ....                                                  // utilizzo del buffer
    _exmback(buffer,size);
    ....
```

L'indirizzo di un buffer allocato da `_exmgrab()` può essere restituito a Clipper tramite la `_retclen()`; con riferimento all'esempio precedente la chimata potrebbe essere

```
_retclen(buffer,size);
```

[189] Utile quando non si tratti di una vera e propria stringa (terminata da `NULL`), ma di una sequenza di caratteri qualsiasi, da gestire in modo binario.

[190] Forse non guasta ricordare che si tratta per default di un puntatore `far`, dal momento che il modello di memoria utilizzato è sempre il large model. Identica considerazione vale per tutti i puntatori a stringa (`char *`) utilizzati da alcune delle funzioni `_par...()` e `_ret...()`.

E' palese che il buffer non deve essere disallocato né prima né dopo la chiamata a `_retclen()`.

La libreria del più recente Clipper 5 comprende invece 3 funzioni per l'allocazione dinamica della RAM, due delle quali sostituiscono quelle appena descritte. La

```
void *_xalloc(unsigned int size);
```

rimpiazza la `_exmgrab()`. Anche la `_xalloc()` restituisce `NULL` in caso di errore. Alla `_xalloc()` si affianca una funzione di nuova concezione, la

```
void *_xgrab(unsigned int size);
```

alloca anch'essa `size` byte nello heap di Clipper ma, a differenza della `_xalloc()` genera un run-time error in caso di errore. La `_exmback()` è stata sostituita dalla

```
void _xfree(void *mem);
```

Come si vede, la `_xfree()` accetta un unico parametro, rappresentante l'indirizzo dell'area di memoria da liberare: non è più possibile, quindi, richiedere la disallocazione di una parte soltanto della memoria in precedenza allocata.

ALCUNI ESEMPI

Presentiamo di seguito alcuni esempi di funzioni C richiamabili da programmi Clipper. La prima permette di utilizzare alcuni servizi DOS da procedure Clipper.

```
/********************

    BARNINGA_Z! - 1993

    CL_BDOS.C - cl_bdos()

    void pascal cl_bdos(void);

    Sintassi per Clipper:

    int cl_bdos(int dosfn,int dosdx,int dosal);
    int dosfn    numerico intero rappresentante il numero di servizio dell'int 21h
    int dosdx    numerico intero rappresentante il registro DX
    int dosal    numerico intero rappresentante il registro AL

    Restituisce: il valore restituito dall'int 21h nel regsitro AX

    COMPILABILE CON MICROSOFT C 5.1

        CL /c /AL /Oalt /FPa /Gs /Zl cl_bdos.c

********************/
#include <extend.h>
#include <dos.h>

#define  PNUM    3
#define  ERROR  -1

CLIPPER cl_bdos(void)
{
    register i;
    int fn, dx, al;
```

```
    if(_parinfo(0) != PNUM) {           // la proc. Clipper chiamante passa 3 param.
        _retni(ERROR);
        return;
    }
    for(i = 1; i <= PNUM; i++)
        if(_parinfo(i) != NUMERIC) {
            _retni(ERROR);
            return;
        }
    _retni(bdos(_parni(1),_parni(2),_parni(3)));
    return;
}
```

Ed ecco un esempio di chiamata alla `cl_bdos()` in una routine Clipper:

```
DOS_FN = 12  && servizio 0Ch int 21h (vuota buffer tastiera e invoca altro servizio)
REG_AL = 7              && servizio da invocare in servizio 0Ch (7 = attende tasto)
RET_VAL = CL_BDOS(DOS_FN,0,REG_AL)                           // chiama interrupt 21h
IF RET_VAL = -1
    @ 10,12 SAY "Errore!"
ELSE
    RET_VAL = RET_VAL % 256                                           // calcola AL
    ....                            // utilizza il valore restituito (tasto premuto)
ENDIF
```

Il frammento di codice Clipper presentato utilizza la `cl_bdos()` per invocare l'int 21h, servizio 0Ch, con `AL` = 7: l'operazione che il servizio esegue consiste nel vuotare il buffer di tastiera e interrompere l'esecuzione del programma in attesa della pressione di un tasto. La `cl_bdos()` restituisce il valore di `AX` a sua volta restituito dall'int 21h: da questo viene poi "estratto" il valore di `AL`, cioè il codice ASCII del tasto premuto[191]. La formule per ricavare il valore di un registro a 16 bit a partire dai due sottoregistri a 8 bit che lo compongono è molto semplice:

```
RX = (RH * 256) + RL
```

dove `RX` indica un generico registro a 16 bit, mentre `RH` e `RL` rappresentano, rispettivamente, il sottoregistro "alto" (gli 8 bit più significativi) e quello "basso" (gli 8 bit meno significativi). Inoltre sono valide le seguenti:

```
RH = RX / 256
RL = RX % 256
```

Si può cioè affermare che gli 8 bit più significativi sono ottenibili dividendo il valore a 16 bit per `256`, senza considerare resto o decimali, mentre gli 8 bit meno significativi sono il resto della precedente divisione.

Ed ora il secondo esempio. E' noto che in ogni programma C `main()` può essere dichiarata con alcuni parametri formali (vedere pag. 105): il secondo di questi, solitamente chiamato `argv`, è un array di puntatori a stringa (o, meglio, a carattere: `char **argv`), la prima delle quali (`argv[0]`) rappresenta il nome del programma eseguibile, completo di pathname. In Clipper è possibile accedere ai parametri della command line[192], ma non si ha modo di conoscere, in modo quasi "automatico", come nei programmi C, nome e pathname del programma stesso. L'ostacolo può essere aggirato con una funzione C, il cui modulo oggetto deve essere collegato all'object file prodotto dal compilatore Clipper.

[191] Una funzione C dedicata allo svuotamento del buffer di tastiera è presentata a pag. 527.

[192] E' sufficiente dichiararli come parametri in testa al programma.

```
/********************

    BARNINGA_Z! - 1991

    CL_EXENM.C - cl_exename()

    void pascal cl_exename(void);

    char *cl_exename();
    Restituisce:  il puntatore ad una stringa che rappresenta il nome del
                  programma eseguibile completo di pathname

    COMPILABILE CON BORLAND C++ 3.1

        tcc -O -d -c -ml cl_exenm.c

********************/
#include <extend.h>
#include <dir.h>
#include <dos.h>

CLIPPER cl_exename(void)
{
    register i;
    unsigned PSPseg;
    char *ENVptr;
    static char exeName[MAXPATH];

    _AH = 0x62;
    geninterrupt(0x21);
    PSPseg = _BX;
    ENVptr = MK_FP(*(unsigned far *)MK_FP(PSPseg,0x2C),0);
    for(;;) {
        if(!*ENVptr++)
            if(!*ENVptr)
                break;
    }
    for(; *ENVptr != 1;)
        ENVptr++;
    for(ENVptr += 2; *ENVptr; )
        exeName[i++] = *ENVptr++;
    exeName[i] = 0;
    _retc(exeName);
}
```

La `cl_exename()` utilizza il servizio 62h dell'int 21h per conoscere l'indirizzo di segmento del Program Segment Prefix del programma (vedere pag. 324) e lo utilizza per costruire un puntatore all'environment, cioè alla prima delle variabili d'ambiente. Infatti, la word ad offset 2Ch nel PSP rappresenta l'indirizzo di segmento dell'area allocata dal DOS all'environment. L'espressione

```
*(unsigned far *)MK_FP(PSPseg,0x2C)
```

restituisce detta word, pertanto l'espressione

```
MK_FP(*(unsigned far *)MK_FP(PSPseg,0x2C),0)
```

restituisce il puntatore (`far`) alla prima stringa dell'environment. Le stringhe rappresentanti variabili d'ambiente sono memorizzate l'una di seguito all'altra; ogni stringa è terminata da un byte nullo, che in questo caso funge anche da "separatore". L'ultima stringa nell'environment è conclusa da due byte nulli: la sequenza `00h 00h`, che indica anche la fine dell'environment, è ricercata dal primo ciclo `for`. Il secondo ciclo cerca il byte 01h, che segnala la presenza della stringa contenente nome e path

dell'eseguibile: il byte 01h è seguito da un byte nullo, dopo il quale inizia la stringa; la clausola di inizializzazione

```
ENVptr += 2;
```

del terzo ciclo `for` "scavalca" la sequenza `01h 00h`; l'istruzione che costituisce il corpo del ciclo stesso può così copiare la stringa, un byte ad ogni iterazione, nel buffer statico `exeName`. La scelta dell'allocazione statica evita il ricorso a `_exmgrab()`; del resto non sarebbe possibile dichiarare il buffer come semplice variabile automatica in quanto l'area di memoria da esso occupata verrebbe rilasciata in uscita dalla funzione.

La stringa è esplicitamente terminata da un byte nullo e il suo indirizzo è restituito a Clipper mediante `_retc()`.

La chiamata a `cl_exename()` in Clipper può essere effettuata come segue:

```
EXEPATH = ""
....
EXEPATH = CL_EXENAME()
@ 10,12 say "Questo programma è " + EXEPATH
....
```

Qualora il programma Clipper deallochi il proprio environment (eventualmente utilizzando una funzione scritta in C), è indispensabile che la chiamata a `cl_exename()` avvenga prima di detta operazione.

Va ancora sottolineato che il sorgente di `cl_exename()` può essere compilato con il compilatore Borland: d'altra parte non referenzia alcuna funzione di libreria C.

Vediamo un ultimo esempio: una funzione in grado di suddividere un numero in virgola mobile in parte intera e parte frazionaria, basata sulla funzione di libreria C `modf()`.

```
/********************

    BARNINGA_Z! - 1994

    CL_MODF.C - cl_modf()

    void pascal cl_modf(void);

    Sintassi per Clipper:

    double cl_modf(double n,double @ipart);
    double n        numerico in virgola mobile che si vuole suddividere in parte
                    intera e parte frazionaria
    double @ipart   reference a numerico in virgola mobile destinato a contenere
                    la parte intera

    Restituisce: la parte frazionaria di n

    COMPILABILE CON MICROSOFT C 5.1

        CL /c /AL /Oalt /FPa /Gs /Zl cl_modf.c

********************/
#include <extend.h>
#include <math.h>

#define  PNUM    2
#define  ERROR   0.0

CLIPPER cl_modf(void)
{
```

```
    double ip, fp;

    if(PCOUNT != PNUM) {                // la proc. Clipper chiamante passa 2 param.
        _retnd(ERROR);
        return;
    }
    if(! ISNUM(1)) {
        _retnd(ERROR);
        return;
    }
    if(!(ISNUM(2) && ISBYREF(2))) {
        _retnd(ERROR);
        return;
    }
    fp = modf(_parnd(1),&ip);
    if(!_storni(ip,2)) {
        _retnd(ERROR);
        return;
    }
    _retnd(fp);
    return;
}
```

Segue esempio di chiamata alla `cl_modf()` in una routine Clipper:

```
DOUBLE_NUM = 12.5647
INT_PART = 0.0
FRAC_PART = CL_MODF(DOUBLE_NUM,@INT_PART)
@ 10,12 SAY "PARTE INTERA: "+STR(INT_PART)+"   PARTE FRAZIONARIA: "+STR(FRAC_PART)
```

Il programma Clipper visualizza il numero `12` come parte intera e il numero `0.5647` come parte frazionaria. La restituzione di due valori (parte intera e parte frazionaria di `DOUBLE_NUM`) alla routine chiamante è resa possibile dal passaggio per reference della variabile `INT_PART` alla `CL_MODF()`: questa, mediante la `_stornd()` è in grado di memorizzare all'indirizzo della `INT_PART`, cioè nella `INT_PART` stessa, il valore che, a sua volta, la `modf()` ha scritto all'indirizzo di `ip`, cioè nella `ip` medesima.

E' evidente che l'utilizzo delle macro `PCOUNT`, `ISNUM()` e `ISBYREF()`, nonché della funzione `_storni()`, rende la `CL_MODF()` utilizzabile esclusivamente da programmi compilati da Clipper 5.

GESTIONE A BASSO LIVELLO DELLA MEMORIA

Il presente capitolo non ha la pretesa di analizzare dal punto di vista tecnico il comportamento del DOS o delle funzioni di allocazione dinamica presenti nella libreria C: esso si propone, piuttosto, di fornire qualche spunto su particolarità non sempre evidenti[193]. Alcuni cenni di carattere tecnico sono, tuttavia, indispensabili.

IL COMPILATORE C

La libreria C comprende diverse funzioni dedicate all'allocazione dinamica della RAM: esse possono essere suddivise, forse un poco grossolanamente, in due gruppi.

Da un lato vi sono quelle che gestiscono la memoria secondo modalità, per così dire, tipiche della libreria C: `malloc()`, `realloc()`, `free()` e, in sostanza, tutte le funzioni dichiarate nel file `ALLOC.H` (o `MALLOC.H`)[194]: se ne parla a pagina 109.

Dall'altro lato troviamo le funzioni basate sui servizi di allocazione della memoria resi disponibili dall'int 21h[195]: `allocmem()`, `setblock()` e `freemem()`, dichiarate in `DOS.H`. Ecco la descrizione dei servizi testè citati:

INT 21H, SERV. 48H: ALLOCA UN BLOCCO MEMORIA

Input	AH	48h
	BX	Numero di paragrafi da allocare
Output	AH	Indirizzo di segmento dell'area allocata, oppure il codice di errore se `CarryFlag = 1`. In questo caso `BX` contiene il massimo numero di paragrafi disponibili per l'allocazione.
Note		Se la funzione è eseguita con successo, `AX:0000` punta all'area allocata. Invocare la funzione con `BX = FFFFh` è un metodo per conoscere la quantità di memoria libera.

[193] Che cosa vi aspettavate? Questa non è una guida di riferimento tecnico per il sistema operativo, né un supplemento alla manualistica dei compilatori C. Questa è... beh... chissà.

[194] Il C Borland include `ALLOC.H`; il C Microsoft `MALLOC.H`.

[195] Si noti che i servizi DOS gestiscono la memoria in unità minime di 16 byte, dette paragrafi. Inoltre, ogni blocco allocato dal DOS si trova sempre ad un indirizzo allineato a paragrafo (divisibile, cioè, per 16), esprimibile con un'espressione del tipo `segmento:0000`.

INT 21H, SERV. 49H: DEALLOCA UN BLOCCO DI MEMORIA

Input	AH ES	49h Segmento dell'indirizzo dell'area da liberare
Output	AX	Codice di errore, se il CarryFlag = 1.
Note		Questo servizio restituisce al DOS un'area allocata mediante il servizio 48h. ES contiene il valore da questo restituito in AX.

INT 21H, SERV. 4AH: MODIFICA L'AMPIEZZA DEL BLOCCO DI MEMORIA ALLOCATO

Input	AH BX ES	4Ah Nuova dimensione in paragrafi del blocco Segmento dell'indirizzo del blocco da modifcare
Output	AX	Codice di errore, se CarryFlag = 1. In questo caso, se il blocco doveva essere espanso, BX contiene il massimo numero di paragrafi disponibili.
Note		Questa funzione è utilizzata per espandere o contrarre un blocco precedentemente allocato via servizio 48h.

E' evidente che un programma il quale intenda interagire con il DOS nell'allocazione della RAM deve necessariamente utilizzare le funzioni appartenenti al secondo gruppo oppure ricorrere direttamente all'int 21h.

MEMORIA CONVENZIONALE

La *memoria convenzionale* è costituita dai primi 640 Kb (o meno di 640) di RAM installati sulla macchina: essi sono compresi tra gli indirizzi 0000:0000 e 9FFF:000F[196]. Essi sono l'unica parte di RAM che il DOS è in grado di utilizzare senza artifici per l'esecuzione di se stesso e dei programmi applicativi. L'uso della memoria convenzionale è descritto graficamente in figura 7.

Il primo Kilobyte, dall'indirizzo 0000:0000 (origine) a 003F:000F, è occupato dalla tavola dei vettori (vedere pag.). I successivi 256 byte costituiscono un'area a disposizione del BIOS per la gestione di dati come il modo video attuale, il timer, etc.; essi sono seguiti da un'area di 512 byte usata in modo analogo dal DOS. A 0070:0000 è caricato, in fase di bootstrap, il primo dei due file nascosti di sistema, di solito chiamato, a seconda della versione di DOS e del suo produttore, IBMBIO.COM o MSDOS.SYS[197]. Tutti i restanti oggetti evidenziati in figura 7 (a partire dal secondo file nascosto,

[196] Forse è opportuno ricordare che gli indirizzi segmento:offset sono una rappresentazione (coerente con i registri a 16 bit della CPU) di un indirizzo a 20 bit; 9FFF:000F equivale a 9FFFF.

[197] Tutte le versioni di DOS, inclusa la 6.2, sembrano caricare il primo file nascosto proprio all'indirizzo 0070:0000.

IBMDOS.COM o IO.SYS) sono caricati ad indirizzi variabili che dipendono dalla versione del sistema operativo e dalla configurazione della macchina (dal tipo e dal numero di device driver caricati, per fare un esempio). Il confine tra le aree "Programmi Applicativi" e "COMMAND.COM: parte transiente" è tratteggiato, in quanto la parte transiente dell'interprete dei comandi può essere sovrascritta in qualsiasi momento dai programmi di volta in volta eseguiti: essa è caricata nella parte alta della memoria convenzionale, ma lo spazio occupato non viene considerato un'area protetta[198]. L'allocazione della memoria per tutti gli oggetti caricati in RAM successivamente a IO.SYS (o IBMDOS.COM) è gestita mediante i Memory Control Block (MCB): ciascuno di essi contiene le informazioni necessarie alla gestione dell'area di memoria della quale costituisce l'intestazione[199].

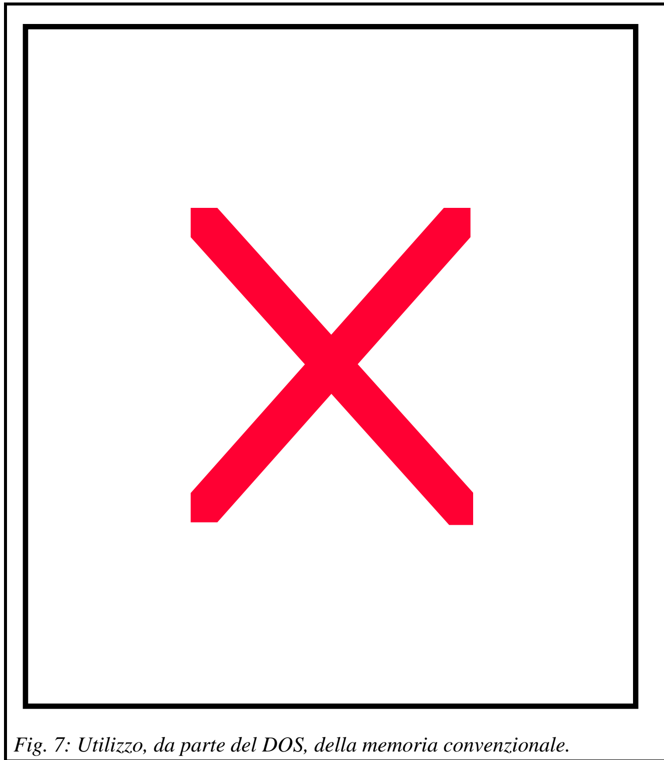

Fig. 7: Utilizzo, da parte del DOS, della memoria convenzionale.

Facciamo un esempio: dopo il bootstrap vi è una porzione (solitamente ampia) di RAM libera, disponibile per l'esecuzione dei programmi. In testa a tale area vi è un MCB. Quando viene caricato ed eseguito un programma, il DOS gli assegna due aree: la prima, di norma piccola, contiene una copia delle variabili d'ambiente (l'environment); la seconda è riservata al programma stesso. L'area libera è stata così suddivisa in tre parti: le prime due appartengono al programma, mentre la terza è libera. Ciascuna delle tre ha il proprio MCB: da ogni MCB è possibile risalire al successivo, ricostruendo così tutta la catena (e quindi la mappa dell'utilizzo della RAM). Supponiamo che il programma non utilizzi il proprio environment, e quindi restituisca al DOS la memoria da quello occupata: le aree sono sempre tre, ma solo la seconda è allocata, mentre la prima e la terza sono libere. Quando, infine, il programma termina, la RAM da esso occupata torna ad essere libera: per evitare inutili frazionamenti della RAM il DOS riunisce tutte le aree libere contigue. Si ritorna perciò alla situazione di partenza: un'unica area, libera, con un unico MCB.

A questo punto è indispensabile analizzare gli MCB con maggiore dettaglio: la figura 8 ne descrive la struttura.

Come si vede, ciascuno di essi ha ampiezza pari a un paragrafo (16 byte) ed è suddiviso in campi.

---

[198] Il DOS non distingue le due aree: dopo il bootstrap, tutta la RAM al di sopra dell'environment di COMMAND.COM è considerata un'unica area, libera, a disposizione dei programmi. La parte transiente di COMMAND.COM, se sovrascritta, viene ricaricata da disco all'occorrenza.

[199] In sostanza, il DOS gestisce la RAM per aree (che possono essere allocate ad un programma oppure libere), in testa ad ognuna delle quali crea un MCB. Un po' di pazienza, tra breve analizzeremo i MCB in dettaglio.

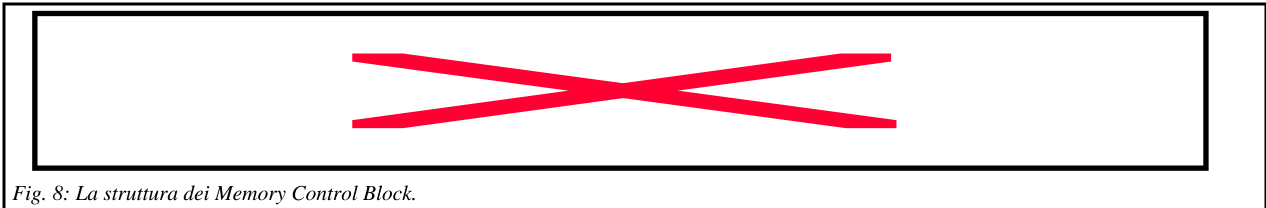

Fig. 8: La struttura dei Memory Control Block.

Il campo POS, di un solo byte, indica la posizione del MCB: se questo è l'ultimo (l'area di RAM che esso controlla è quella che inizia al maggiore indirizzo) il campo contiene il carattere 'Z', altrimenti il carattere 'M'[200].

Il campo PSP, una word (unsigned int, dal punto di vista del C), indica l'indirizzo di segmento del Program Segment Prefix del programma a cui appartiene l'area di memoria[201]. Nell'esempio precedente, i campi PSP del MCB del programma e del suo environment hanno il medesimo contenuto. Il campo PSP del MCB di un'area libera assume valore zero. I valori 6 e 8 indicano che l'area è riservata al DOS; in particolare, 8 è il valore che assume il campo PSP del MCB dell'area allocata ai device driver. Detto MCB è, tra l'altro, il primo della catena[202].

Il campo DIM, una word, esprime la dimensione, in paragrafi, dell'area di memoria (escluso il MCB medesimo). Incrementando di uno la somma tra l'indirizzo di segmento di un MCB e il suo campo DIM si ottiene l'indirizzo di segmento del successivo MCB. Se il calcolo è effettuato con riferimento all'ultimo MCB della catena, il valore ottenuto è il cosiddetto Top Of Memory (A000h nel caso di 640 Kb installati)[203].

I 3 byte del campo RESERVED attualmente non sono utilizzati.

Il campo NAME, di 8 byte, a partire dal DOS 4.0 contiene il nome del programma[204] a cui l'area è assegnata (se questa contiene il Program Segment Prefix del programma: rifacendosi ancora all'esempio riportato, il nome non appare nel MCB dell'environment). Se il nome non occupa tutti gli 8 byte

[200] Questa è la regola generale. A partire dal DOS 4.0, però, l'area di RAM allocata ai device driver ha un MCB regolare, recante la lettera 'M' nel campo POS, ma è a sua volta suddivisa in tante sub-aree quanti sono i driver installati, ciascuna dotata, in testa, di un proprio MCB. In tali Memory Control Block il campo POS indica il tipo di driver; il suo contenuto può essere: 'D' blocco device driver (installato dal comando DEVICE in CONFIG.SYS), 'F' blocco FILES, 'X' blocco FCBS, 'B' blocco BUFFERS, 'C' blocco buffer EMS, 'I' blocco IFS, 'L' blocco LASTDRIVE, 'S' blocco STACKS, 'E' blocco device driver appendage.

[201] Il PSP è un record di 256 byte che il DOS prepara in testa al codice del programma eseguito. Per ogni programma caricato in memoria si ha dunque un MCB, immediatamente seguito dal PSP, a sua volta seguito dal codice del programma stesso. L'indirizzo di segmento del PSP di un programma è pertanto pari a quello del suo MCB, incrementato di uno.

[202] In effetti, l'area dei device driver è quella che immediatamente segue la RAM riservata al secondo file nascosto, come evidenziato in figura 1.

[203] Tutte le aree di RAM gestite mediante servizi DOS hanno dimensione (in byte) divisibile per 16 (multiple per paragrafi: non è più una novità). Anche il loro indirizzo è divisibile per 16 (cioè allineato a paragrafo) ed è esprimibile mediante la sola parte segmento (seg:0000). Ne segue che anche i MCB sono allineati a paragrafo, dal momento che occupano il paragrafo immediatamente precedente l'area allocata. Vale infine la pena di sottolineare che i servizi 48h, 49h e 4Ah dell'int 21h restituiscono e/o richiedono in input l'indirizzo (sotto forma di word, la sola parte segmento) dell'area e non quello del MCB (ricavabile decrementando di uno quello dell'area).

[204] Le versioni di DOS anteriori alla 4.0 non utilizzano questo campo; con esse il solo metodo per conoscere il nome del programma è andare a curiosare in coda all'environment di questo (vedere pag. 185 per un esempio di metodo valido, comunque, anche con DOS 4 e successivi). Qui il nome è memorizzato completo di drive e pathname; tuttavia esso scompare se la RAM allocata all'environment viene liberata.

disponibili, quelli restanti sono riempiti con spazi (ASCII 32, esadecimale 20). Se l'area è riservata al DOS, il nome, quando presente, è solitamente una stringa significativa per il solo DOS, e il carattere tappo può essere l'ASCII 256 (FFh).

Qui giunti, conosciamo quanto basta (con un po' di ottimismo e di fortuna) per lavorare alla pari con il DOS. Si tratta, ora, di illustrare alcuni metodi (gran parte dei quali non documentati ufficialmente) per individuare gli indirizzi degli oggetti di cui abbiamo discusso.

Cominciamo dal secondo file nascosto. Il servizio 34h dell'int 21h restituisce l'indirizzo, nel segmento di memoria allocato al DOS, dell'InDOS flag[205]:

INT 21H, SERV. 34H: INDIRIZZO DELL'INDOS FLAG

Input	AH	34h
Output	ES:BX	Indirizzo (seg:off) dell'InDOS flag.

La parte segmento dell'indirizzo dell'InDOS flag (restituita in ES) è l'indirizzo di segmento al quale è caricato il secondo file nascosto; esso si trova, in altri termini, a ES:0000. La funzione getdosseg() è un esempio di come è possibile procedere per ottenere detto indirizzo.

```
/********************

    BARNINGA_Z! - 1991

    DOSSEG.C - getdosseg()

    unsigned cdecl getdosseg(void);
    Restituisce: l'indirizzo di segmento di IO.SYS o IBMDOS.COM

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx dosseg.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

unsigned cdecl getdosseg(void)
{
    union REGS regs;
    struct SREGS sregs;

    regs.h.ah = 0x34;
    intdos(&regs,&regs);
    segread(&sregs);
    return(sregs.es);
}
```

Circa la parola chiave cdecl vedere pag. 92. Una versione più efficiente della getdosseg() è listata di seguito:

```
/********************

    BARNINGA_Z! - 1991
```

[205] Alcune interessanti particolarità relative all'InDOS Flag sono discusse alle pagine e seguenti.

```
    DOSSEG.C - getdosseg()

    unsigned cdecl getdosseg(void);
    Restituisce: l'indirizzo di segmento di IO.SYS o IBMDOS.COM

    COMPILABILE CON BORLAND C++ 2.0

        bcc -O -d -c -k- -mx dosaddr.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-                                       // per maggiore efficienza

unsigned cdecl getdosseg(void)
{
    _AX = 0x34;
    asm int 21h;
    return(_ES);
}
```

Passiamo ai Memory Control Block. Come si è detto, la prima area gestita tramite MCB è quella dei device driver. Sfortunatamente, anche in questo caso non esistono metodi ufficiali per conoscerne l'indirizzo. Esiste, però, un servizio non documentato dell'int 21h, la funzione 52h, detto "*GetDosListOfLists*", che restituisce l'indirizzo di una tavola di parametri ad uso interno del sistema. La word che precede questa tavola è l'indirizzo (segmento) del primo MCB.

INT 21H, SERV. 52H: INDIRIZZO DELLA LISTA DELLE LISTE

Input	AH	52h
Output	ES:BX	Indirizzo (segmento:offset) della lista delle liste.

Di seguito riportiamo un esempio di funzione che restituisce l'indirizzo di segmento del primo MCB.

```
/********************

    BARNINGA_Z! - 1991

    FIRSTMCB.C - getfirstmcb()

    unsigned cdecl getfirstmcb(void);
    Restituisce: l'indirizzo di segmento del primo MCB

    COMPILABILE CON BORLAND C++ 2.0

        bcc -O -d -c -mx firstmcb.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

unsigned cdecl getfirstmcb(void)
{
    union REGS regs;
```

```
    struct SREGS sregs;

    regs.h.ah = 0x52;
    intdos(&regs,&regs);
    segread(&sregs);
    return(*(unsigned far *)MK_FP(sregs.es,regs.x.bx-2));
}
```

La macro `MK_FP()` è descritta a pag. 24. Anche in questo caso riportiamo la versione basata sullo inline assembly:

```
/********************

    BARNINGA_Z! - 1991

    FIRSTMCB.C - getfirstmcb()

    unsigned cdecl getfirstmcb(void);
    Restituisce: l'indirizzo di segmento del primo MCB

    COMPILABILE CON BORLAND C++ 2.0

        bcc -O -d -c -k- -mx firstmcb.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-                                    // per maggiore efficienza

unsigned cdecl getfirstmcb(void)
{
    asm mov ah,52h;
    asm int 21h;
    asm mov ax,es:[bx-2];
    return(_AX);
}
```

A scopo di chiarezza, ripetiamo che `getfirstmcb()` non restituisce l'indirizzo della prima area di RAM controllata da MCB, bensì quello del primo MCB. L'indirizzo (di segmento) dell'area si ottiene, ovviamente, sommando uno al valore restituito.

Ora che sappiamo dove trovarli, i MCB possono essere comodamente manipolati con l'aiuto di una struttura:

```
struct MCB {
    char     pos;
    unsigned psp;
    unsigned dim;
    char     reserved[3];
    char     name[8];
};
```

Attenzione: il campo `name` della struttura di tipo MCB non è una vera e propria stringa, in quanto privo del `NULL` finale. Ecco, ora, il listato di una funzione in grado di copiare un Memory Control Block in un buffer appositamente predisposto.

```
/********************

    BARNINGA_Z! - 1991
```

```
    PARSEMCB.C - parsemcb()

    unsigned cdecl parsemcb(struct MCB *mcb,unsigned mcbseg);
    struct MCB *mcb;     puntatore ad una struttura di tipo MCB: deve
                         essere gia' allocata
    unsigned   ncbseg;   indirizzo (segmento) del MCB da copiare

    Restituisce: l'indirizzo (segmento) dell'area di RAM controllata dal
                 MCB dopo avere copiato il contenuto del MCB nella
                 struttura mcb.

    COMPILABILE CON BORLAND C++ 2.0

        bcc -O -d -c -mx parsemcb.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

#include <dos.h>

unsigned cdecl parsemcb(struct MCB *mcb,unsigned mcbseg)
{
    asm push ds;

#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)

    asm push ds;                    // il compilatore assume che nei modelli "piccoli"
    asm pop es;                        // SS e DS coincidano, percio' si puo' usare DS
    asm mov di,mcb;

#else

    asm les di,dword ptr mcb;          // modello dati "grandi": mcb e' una doubleword

#endif

    asm mov ds,mcbseg;                              // DS:SI punta al Memory Control Block
    asm xor si,si;                                       // ES:DI punta alla struttura mcb
    asm mov cx,8;                                              // copia 8 words (16 bytes)
    asm cld;
    asm cli;
    asm rep movsw;
    asm sti;
    asm pop ds;
    return(mcbseg+1);
}
```

Lo inline assembly rende il codice compatto e veloce[206]; la funzione potrebbe comunque essere realizzata facilmente in C puro. La `parsemcb()` copia i dati di un MCB in una struttura di template

[206] Il puntatore `mcb` non è dichiarato `near` né `far` (pag. 21): il suo tipo dipende perciò dal modello di memoria scelto per la compilazione (pag. 143); esso, in particolare, è `near` nei modelli tiny, small e medium. All'interno della funzione non è possibile sapere se la struttura a cui `mcb` punta è stata allocata nello heap con una chiamata a `malloc()` (con indirizzo relativo a `DS`), nell'area dati statici e globali (indirizzo ancora relativo a `DS`) o nello stack come variabile automatica (indirizzo relativo a `SS`). In tutti gli esempi di funzione presentati nel testo, in casi come quello analizzato, si assume che nei modelli small e medium `DS` e `SS` coincidano (e si utilizza dunque `DS` per ricavare la parte segmento dell'indirizzo), in quanto questo è il default di comportamento del compilatore. Solo con particolari e pericolose opzioni della riga di comando è infatti possibile richiedere che `DS` e `SS`, in detti modelli di memoria, non siano necessariamente uguali.

MCB e restituisce l'indirizzo del Memory Control Block incrementato di uno, cioè l'indirizzo (segmento) dell'area di memoria controllata da quel MCB.

Abbiamo tutto ciò che occorre per ricostruire la mappa della memoria convenzionale: basta collegare i vari frammenti in modo opportuno.

```
/********************

    BARNINGA_Z! - 1991

    MCBCHAIN.C - getmcbchain()

    struct MCB * cdecl getmcbchain(unsigned basemcb);
    unsigned basemcb;    indirizzo (segmento) del primo MCB della catena.
    Restituisce: un puntatore ad un array di strutture di tipo MCB.

    COMPILABILE CON BORLAND C++ 2.0

        bcc -O -d -c -mx mcbchain.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <stdio.h>                                                // per NULL
#include <alloc.h>                                   // per malloc() e realloc()

struct MCB * cdecl getmcbchain(unsigned basemcb)
{
    register i;
    unsigned mcbseg;
    struct MCB *mcb;

    if(!(mcb = (struct MCB *)malloc(sizeof(struct MCB))))
        return(NULL);
    mcbseg = parsemcb(mcb,basemcb);
    mcbseg += mcb->dim;             // segmento MCB + 1 + dimensione MCB = segmento
    i = 1;                                                 // del MCB successivo
    do {
        if(!(mcb = (struct MCB *)realloc(mcb,(i+1)*sizeof(struct MCB))))
            return(NULL);
        mcbseg = parsemcb(mcb+i,mcbseg);
        mcbseg += mcb[i].dim;
    } while(mcb[i++].pos != 'Z');            // i e' incrementata dopo il confronto
    return(mcb);
}
```

La `getmcbchain()` prende come parametro l'indirizzo di segmento del primo MCB della catena, facilmente ottenibile mediante la `getfirstmcb()`: come si può vedere, nulla di complicato. Per avere una mappa completa della memoria convenzionale basta ricavare l'indirizzo del secondo file nascosto con una chiamata alla `getdosseg()`. La mappa può poi essere arricchita individuando la strategia utilizzata dal DOS nell'allocazione della memoria, cioè l'algoritmo con il quale il DOS ricerca un blocco libero di dimensioni sufficienti. Le strategie possibili sono tre: la prima, detta *FirstFit*, consiste nel ricercare il blocco a partire dall'origine della RAM; la seconda, *BestFit*, nell'allocare il blocco nella minore area libera disponibile; la terza, *LastFit*, si esplica nell'allocare la parte alta dell'ultimo blocco libero. La strategia di allocazione è gestita dal servizio 58h dell'int 21h.

INT 21H, SERV. 58H: GESTIONE DELLA STRATEGIA DI ALLOCAZIONE

Input	AH	58h
	AL	00h: ottiene la strategia di allocazione 01h: determina la strategia di allocazione
	BX	solo per AL = 01h (set strategy): 0 : FirstFit 1 : BestFit >= 2 : LastFit
Output	AX	codice di errore se CarryFlag = 1; altrimenti: solo per AL = 00h (get strategy): 0 : FirstFit 1 : BestFit >= 2 : LastFit

```
/********************

    BARNINGA_Z! - 1992

    ALCSTRAT.C - getallocstrategy()

    int cdecl getallocstrategy(void);
    Restituisce: la strategia di allocazione DOS. In caso di errore
                 restituisce -1.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx alcstrat.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-                                        // maggiore efficienza

int cdecl getallocstrategy(void)
{
    _AX = 0x5800;
    asm int 21h;
    asm jnc EXITFUNC;
    _AX = -1;
EXITFUNC:
    return(_AX);
}
```

Per un esempio di modifica della strategia DOS di allocazione vedere pag. .

UPPER MEMORY

Il metodo di puntamento basato sulla coppia segmento:offset consente al DOS di indirizzare, su macchine a 16 bit, un megabyte di RAM[207]. I 384 Kb compresi tra i primi 640 Kb e il Mb sono, di norma, utilizzati come indirizzi per il ROM-BIOS o sue estensioni, per la gestione del video, etc.: uno schema è riprodotto in figura 9.

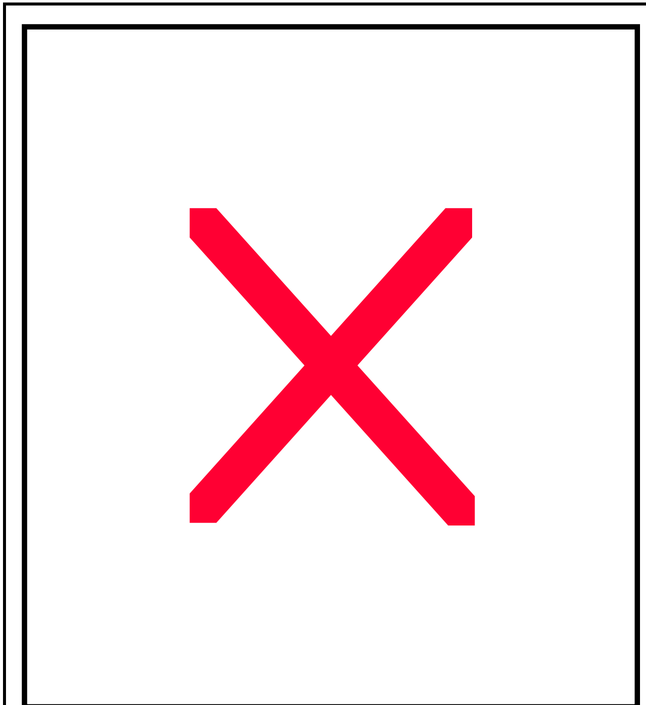

Fig. 9: Utilizzo degli indirizzi di memoria tra i 640 Kb e il megabyte.

Gli indirizzi compresi tra `C000:0` e `EFFF:000F` sono disponibili per le estensioni ROM-BIOS: possono, cioè, essere utilizzati dalle schede di supporto per il networking o per particolari periferiche (fax, scanner, etc.). Ad esempio, l'intervallo che si estende da da `C000:0` a `C7FF:000F` è di norma occupato dal BIOS delle schede VGA; inoltre molti calcolatori tipo notebook o laptop dispongono di estensioni ROM-BIOS, spesso dedicate al controllo del video LCD, nel range da `E000:0` a `EFFF:000F`.

Gli indirizzi non occupati possono essere impiegati per simulare l'esistenza di aree di memoria che il DOS gestisce in modo analogo a quelle presenti nella memoria convenzionale: a tal fine è indispensabile un driver in grado di rimappare gli indirizzi tra `A000:0` e `FFFF:000F` alla memoria fisicamente presente, in quanto a detti indirizzi non corrisponde RAM installata. Tali driver utilizzano, per effettuare il remapping, memoria espansa: ne consegue la necessità che sulla macchina ne sia installata una quantità sufficiente (almeno pari all'ampiezza totale delle aree da simulare)[208]. Il DOS, a partire dalla versione 5.0, include il software necessario alla gestione, su macchine 80386 e superiori, di aree di RAM tra i 640 Kb e il Mb, dette Upper Memory Block. Inoltre, sono reperibili in commercio diversi prodotti mediante i quali è possibile ottenere prestazioni analoghe o migliori (per efficienza e flessibilità) sia su macchine 80386/80486 che 80286, con o senza DOS 5.0.

La tecnica di gestione degli Upper Memory Block (UMB), analogamente a quanto avviene per la memoria convenzionale, si basa sui Memory Control Block; sfortunatamente, i driver DOS e quelli di produttori indipendenti definiscono le aree e comunicano con il sistema (programmi, etc.) mediante tecniche differenti: insomma, il caos regna sovrano. Gli esempi che seguono intendono fornire gli elementi minimi necessari a ricavare una mappa della Upper Memory: essi vanno comunque "presi con le pinze", dal momento che si basano esclusivamente sui risultati di una ostinata sperimentazione.

Il DOS 5.0 crea (tramite `HIMEM.SYS` e `EMM386.EXE`) la catena di MCB per gli Upper Memory Block restringendo l'ultima area di memoria convenzionale di 16 byte, nei quali definisce il

---

[207] Per la precisione: un megabyte e 64 Kb meno 16 byte (`FFFF:FFFF`). I (circa) 64 Kb eccedenti il Mb sono denominati HMA (High Memory Area; vedere pag. e seguenti). Le macchine a 32 bit (80386, 80486, etc.) possono indirizzare linearmente grandi quantità di RAM, ma il limite descritto permane in ambiente DOS.

[208] Quanto detto è vero per le macchine 80286. Le macchine basate su processore 80386 o 80486 (comprese le versioni SX) possono utilizzare anche memoria estesa, se al bootstrap è installato un driver in grado di emulare la memoria espansa attraverso quella estesa.

primo MCB della nuova catena: nel caso di 640 Kb di RAM convenzionale, esso si trova a `9FFF:0`. Il suo campo `POS` contiene il carattere `'M'`; il campo `PSP` è valorizzato a `8`; il campo `NAME` contiene la stringa "`SC`", seguita da sei `NULL`. Sommando il valore contenuto nel campo `DIM` all'indirizzo del MCB si ottiene la parte segmento[209] dell'indirizzo di un successivo MCB[210], il cui campo `NAME` contiene la stringa "`UMB`" seguita da 5 blanks. Il campo `PSP` è pari all'indirizzo del MCB stesso, incrementato di uno; il campo `DIM` esprime la dimensione, in paragrafi, dell'Upper Memory Block. All'interno di questo vi sono i MCB necessari per la definizione delle aree allocate e libere[211]. Il campo `POS` vale `'Z'` se vi è questo UMB soltanto, `'M'` altrimenti: in questo caso, sommando il campo `DIM` incrementato di uno all'indirizzo dell'attuale UMB si ottiene l'indirizzo del successivo MCB. Questo, a sua volta, potrebbe avere lo scopo di "proteggere" un'area non rimappabile[212]. La catena continua sino ad esaurimento degli indirizzi liberi.

Basandosi su queste caratteristiche (lo ripetiamo: individuate empiricamente) è possibile realizzare una funzione in grado di determinare se nel sistema è disponibile Upper Memory gestita dal DOS 5.0:

```
/********************

    BARNINGA_Z! - 1991

    UMBDOS.C - testforDOS()

    unsigned cdecl testforDOS(void);
    Restituisce: l'indirizzo (segmento) del MCB corrispondente al primo
                 UMB (area di RAM sopra i 640 Kb). Restituisce NULL se
                 non riesce ad individuare la catena di UMB.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx umbdos.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <stdio.h>                                                  // per NULL
#include <string.h>                                             // per strncmp()

unsigned cdecl testforDOS(void)
{
    unsigned segbase, umbseg;
    struct MCB umb;                        // struct MCB e parsemcb() sono gia' note
```

[209] Forse vale la pena di ricordare che la parte segmento di un indirizzo equivale alla word più significativa di un puntatore C `far` o `huge`.

[210] Il primo MCB ha lo scopo di escludere dal remapping il buffer video EGA/VGA (`A000:0-AFFF:000F`). La dimensione del MCB può variare a seconda delle opzioni presenti sulla riga di comando del driver che gestisce la Upper Memory.

[211] La logica è analoga a quella descritta circa il caricamento dei device driver in memoria convenzionale.

[212] Riprendendo l'esempio precedente: a `9FFF:0` vi è il primo MCB dell'Upper Memory; la formula `indirizzo+DIM+1` fornisce `B001h`. Se sulla macchina è installato un video a colori, l'intervallo `B001:0-B7FF:0` costituisce il primo UMB (a `B000:0` vi è il suo proprio MCB), che può contenere uno o più MCB. A `B7FF:0` si trova un MCB che ha lo scopo di proteggere l'intervallo `B800:0-C7FF:000F` (nell'ipotesi di scheda VGA presente): la formula `indirizzo+DIM+1` fornisce `C801h`. A `C801:0` vi è un altro UMB (il suo proprio MCB è a `C000:0`), che può contenere diversi MCB, e così via.

```
    segbase = ((*(unsigned far *)0x413)*64)-1;
    parsemcb(&umb,segbase);
    if((umb.pos == 'M') && (umb.psp == 0x0008)) {
        parsemcb(&umb,umbseg = segbase+umb.dim);
        if(((umb.pos == 'M') || (umb.pos == 'Z')) &&
                (umb.psp == umbseg+1) && (!strncmp(umb.name,"UMB     ",8)))
            return(segbase);
    }
    return(NULL);
}
```

La `testforDOS()` ipotizza che gli utlimi 16 byte di memoria convenzionale[213] siano un MCB: se il presunto campo `POS` vale `'M'` e il presunto `PSP` è `8`, allora il controllo prosegue secondo le linee indicate in precedenza. Se i 16 byte all'indirizzo `segbase+umb.dim` sono un MCB il cui campo `PSP` è pari al proprio indirizzo incrementato di uno e il cui campo `NAME` contiene "`UMB     `", allora `testforDOS()` presume che gli ultimi 16 byte di memoria convenzionale siano realmente il primo MCB della catena che gestisce gli Upper Memory Block e ne restituisce l'indirizzo, altrimenti restituisce `NULL` (a significare che non vi è Upper Memory disponibile, o non è stato possibile individuarne la presenza). La mappa della Upper Memory può essere ottenuta semplicemente passando alla `getmcbchain()` di pag. proprio l'indirizzo restituito dalla `testforDOS()`.

Per quanto riguarda i driver reperibili in commercio (non facenti parte del pacchetto DOS) l'esempio che segue fa riferimento al `QEMM386.SYS`, prodotto dalla Quarterdeck Office Systems, il quale, su macchine dotate di processore 80386 o superiore, fornisce supporto per la memoria estesa ed espansa, nonché per gli Upper Memory Block. Il metodo utilizzato per la loro gestione differisce significativamente da quello implementato dal DOS. In primo luogo, non esistono UMB contenitori di aree di RAM: ogni UMB rappresenta un'area a se stante, dotata di un proprio MCB, in modo del tutto analogo alle aree definite entro i primi 640 Kb. Inoltre, il primo UMB si trova al primo indirizzo disponibile sopra la memoria convezionale (e non negli ultimi 16 byte di questa); gli indirizzi non disponibili sono protetti con un UMB il cui MCB presenta nel campo `PSP` il memdesimo valore del campo `PSP` del MCB dell'area allocata, tra i device driver, a `QEMM386.SYS`[214]. Questo, infine, incorpora un gestore per l'interrupt 2Fh, tramite il quale comunica con il sistema. Invocando l'int 2Fh dopo avere caricato con opportuni valori i registri è possibile desumere, dai valori in essi restituiti, se `QEMM386.SYS` è attivo e qual è l'indirizzo del primo UMB.

```
/********************

    BARNINGA_Z! - 1991

    UMBQEMM.C - testforQEMM386()

    unsigned cdecl testforQEMM386(void);
    Restituisce: l'indirizzo (segmento) del MCB corrispondente al primo
                 UMB (area di RAM sopra i 640 Kb). Restituisce NULL se
                 non riesce ad individuare la catena di UMB.

    COMPILABILE CON TURBO C++ 2.0
```

[213] La word a `0:0413` contiene i Kb di memoria convenzionale installati.

[214] Esempio: se sulla macchina è installato un video a colori, a `B000:0` vi è il primo MCB dell'Upper Memory (l'area UMB è a `B001:0`); la formula `indirizzo+DIM+1` fornisce l'indirizzo del successivo MCB. A `B7FF:0` si trova un MCB che ha lo scopo di proteggere l'intervallo `B800:0-C7FF:000F` (nell'ipotesi di scheda VGA presente): la formula `indirizzo+DIM+1` fornisce `C800h`. Qui vi è un altro MCB (l'area UMB è a `C001:0`), per il quale la formula `indirizzo+DIM+1` fornisce l'indirizzo del successivo MCB, e così via.

```
        tcc -O -d -c -mx umbqemm.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-                    // non indispensabile, ma aggiunge efficienza

#include <stdio.h>                                                         // per NULL
#include <string.h>                                                   // per strncmp()

unsigned cdecl testforQEMM386(void)
{
    asm {
        mov ax,0D200h;
        mov bx,05144h;
        mov cx,04D45h;
        mov dx,04D30h;
        int 2Fh;
        cmp ax,0D2FFh;
        jne NOTQEMM386;
        cmp bx,04D45h;
        jne NOTQEMM386;
        cmp cx,04D44h;
        jne NOTQEMM386;
        cmp dx,05652h;
        jne NOTQEMM386;
        mov ax,0D201h;
        mov bx,04849h;
        mov cx,05241h;
        mov dx,04D30h;
        int 0x2F;
        cmp ax,0D201h;
        jne NOTQEMM386;
        cmp bx,04F4Bh;
        jne NOTQEMM386;
        jmp EXITFUNC;
    }
NOTQEMM386:
    asm xor cx,cx;
EXITFUNC:
    return(_CX);
}
```

La `testforQEMM386()` invoca due volte l'int 2Fh. Nella prima richiede il servizio 0 (`AL` = `0`; può essere una richiesta di conferma dell'avvenuta installazione): `AH`, `BX`, `CX` e `DX` contengono, presumibilmente, valori aventi funzione di "parola d'ordine" (vedere, per alcuni dettagli circa l'int 2Fh, pag.). Se `QEMM386.SYS` è installato ed attivo `AX`, `BX`, `CX` e `DX` contengono valori prestabiliti, controllati dalla funzione. La seconda chiamata all'int 2Fh richiede il servizio 1 (`AL` = `1`; appare essere la richiesta dell'indirizzo del primo MCB per UMB): anche in questo caso `AH`, `BX`, `CX` e `DX` sono caricati con valori costanti. `QEMM386.SYS` restituisce in `AX` e `BX` ancora valori prestabiliti, ed in `CX` la parte segmento dell'indirizzo del primo MCB di controllo per gli Upper Memory Block. La `testforQEMM386()` restituisce `NULL` se non ha individuato la presenza di `QEMM386.SYS`. Anche in questo caso il valore restituito, se diverso da `NULL`, può essere parametro attuale della `getmcbchain()` per ricavare la mappa della Upper Memory.

Gli UMB (pag. 226) sono definiti dalla XMS (eXtended Memory Services) Specification, a cui si rimanda per la descrizione dei servizi che consentono la loro allocazione e deallocazione (pag. 246)[215].

MEMORIA ESPANSA

La memoria espansa consiste in pagine[216] di RAM che possono essere copiate dentro e fuori lo spazio fisico di indirizzamento oltre il limite dei 640 Kb. E', in sostanza, un metodo per superare il limite dei 640 Kb tramite un driver in grado di gestire lo scambio di dati tra la RAM direttamente indirizzabile

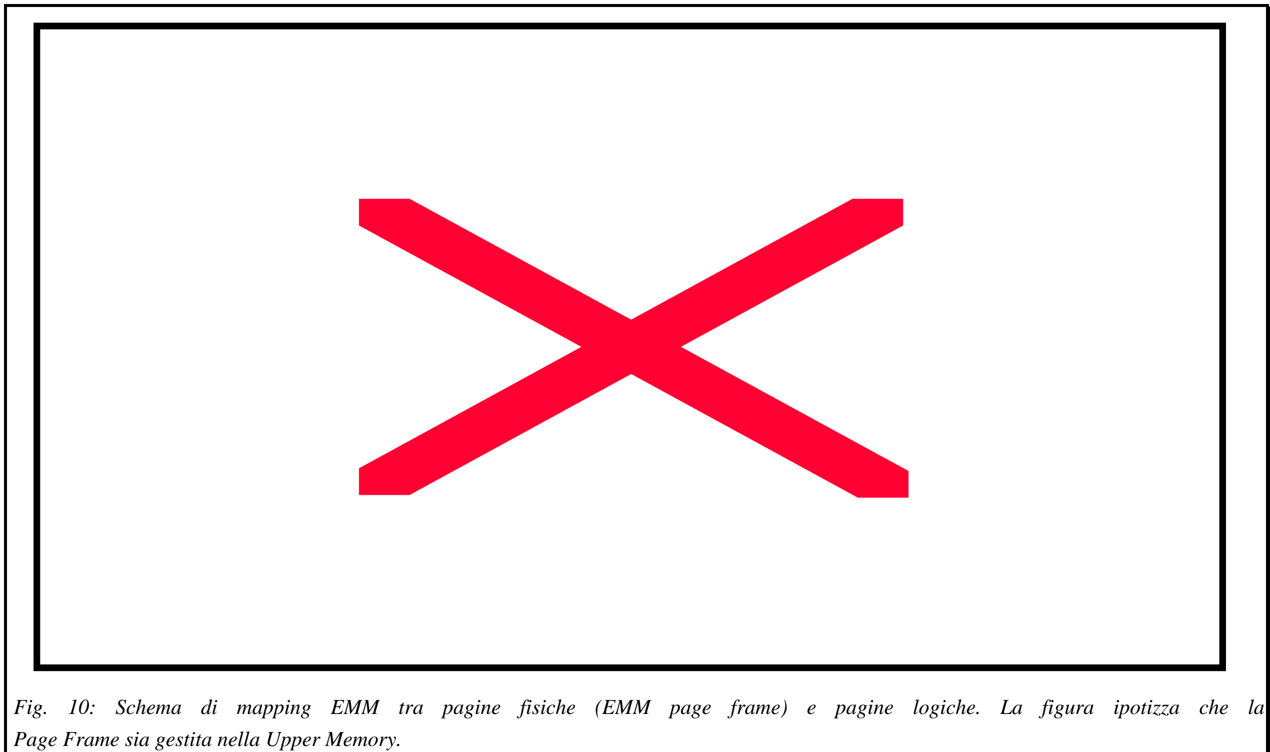

Fig. 10: Schema di mapping EMM tra pagine fisiche (EMM page frame) e pagine logiche. La figura ipotizza che la Page Frame sia gestita nella Upper Memory.

dal DOS e la memoria presente oltre il primo megabyte, attraverso la *page frame*, un'area definita entro il primo Mb stesso (memoria convenzionale o upper memory) e utilizzata come buffer di "transito". Secondo le specifiche LIM 4.0[217] la page frame ha dimensione pari a 64 Kb (4 pagine fisiche), e può essere utilizzata per gestire fino a 32 Mb di RAM, suddivisa in pagine logiche (in genere di 16 Kb). Ad ogni gruppo di pagine logiche è associato uno handle (concetto analogo ma non coincidente con quello di pag. 126), che lo identifica in modo univoco: il driver si occupa di creare una corrispondenza trasparente tra pagine fisiche e pagine logiche associate ad un certo handle (figura 10); il funzionamento del meccanismo sarà sperimentato nel corso del paragrafo. Su macchine 80386 o superiori la memoria espansa può essere emulata (mediante appositi driver) utilizzando la memoria estesa (vedere pag.).

---

[215] Gli esempi su allocazione e disallocazione degli UMB presumono la conoscenza della modalità di chiamata dei servizi XMS, descritta proprio nel capitolo dedicato alla memoria estesa.

[216] Una pagina equivale a 16 Kb.

[217] Lo standard industriale di specifiche per la gestione della memoria espansa definito da Lotus, Intel e Microsoft.

Il driver che gestisce la memoria espansa (detta memoria EMS) installa un proprio gestore dell'int 67h e rende disponibile un device che ha il nome convenzionale `EMMXXXX0`. Esso è detto EMM (EMS Manager). Di seguito sono presentati i listati[218] di alcune funzioni basate sui servizi dell'int 67h.

Prima di effettuare qualsiasi operazione mediante il driver EMS si deve stabilire se esso è effettivamente installato. Il metodo raccomandato dalle specifiche LIM consiste nel tentare l'apertura del device `EMMXXXX0`:

```
/********************

    BARNINGA_Z! - 1991

    EMMTEST.C - testEMM()

    int cdecl testEMM(void);
    Restituisce: 1 se il driver EMM e' installato
                 0 altrimenti

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmtest.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  warn -pia

#include <stdio.h>                                                      // per NULL
#include <dos.h>
#include <io.h>
#include <fcntl.h>

int cdecl testEMM(void)                                    /* uso della memoria espansa */
{
    int handle, retcode;
    struct REGPACK r;

    if((handle = open("EMMXXXX0",O_RDONLY)) == -1)                         // apre handle
        return(NULL);
    retcode = NULL;
    r.r_ax = 0x4400;                                             // e' proprio un device?
    r.r_bx = handle;
    intr(0x21,&r);
    if(!(r.r_flags & 1) && (r.r_dx & 0x80)) {
        r.r_ax = 0x4407;                                           // il device e' ready?
        intr(0x21,&r);
        if(!(r.r_flags & 1) && (r.r_ax & 0xFF))
            retcode = 1;
    }
    close(handle);
    return(retcode);
}
```

La `testEMM()` apre il device `EMMXXXX0` (nome di default dell'Expanded Memory Manager) e, mediante la subfunzione 0 del servizio 44h dell'int 21h controlla che esso sia effettivamente un device (bit 7 di `DX` = `1`) e non un file[219]. Tramite la subfunzione 7 del medesimo servizio, `getEMMusage()`

[218] Parte delle funzioni è scritta in C puro, parte, a scopo esemplificativo, si basa sull'inline assembly.

[219] Se il bit 7 di `DX` è `0`, allora esiste nella directory corrente un file avente nome `EMMXXXX0`: la `open()` ha aperto detto file (e non il device EMM). Quando si dice la sfortuna...

controlla che il device sia in stato di ready (AL = FFh); in caso affermativo può essere comunicato (retcode = 1) alla funzione chiamante che il driver è installato e pronto ad eseguire i servizi richiesti via int 67h.

E' possibile utilizzare un metodo alternativo per controllare la presenza del driver EMM, basato sulle specifiche Microsoft per la struttura dei device driver (vedere pag. 355). Vediamo una seconda versione di testEMM(), più snella della precedente[220].

```
#include <dos.h>
#include <string.h>

#define  EMMNAME    "EMMXXXX0"

int cdecl testEMM2(void)
{
    struct REGPACK r;

    r.r_ax = 0x3567;                              // richiede il vettore dell'int 67h
    intr(0x21,&r);
    return(!_fstrncmp((char far *)MK_FP(r.r_es,10),EMMNAME,strlen(EMMNAME)));
}
```

La testEMM2() costruisce un puntatore far a carattere la cui parte segmento è data dalla parte segmento del vettore dell'int 67h (ottenuto tramite il servizio 35h dell'int 21h) e la cui parte offset equivale a 10 (ad offset 10 del blocco di memoria allocato ad un device driver si trova il nome dello stesso); la funzione _fstrncmp() è utilizzata per vedere se la sequenza di caratteri che si trova a quell'indirizzo è proprio EMMXXXX0. Se testEMM2() è compilata con uno dei modelli di memoria tiny, small o medium (pag. 143) l'indirizzo della costante manifesta EMMNAME è near, ma il compilatore provvede, in base al prototipo di _fstrncmp() dichiarato in STRING.H, a convertirlo opportunamente in puntatore far. L'uso di _fstrncmp() in luogo di _fstrcmp() è imposto dal fatto che il nome dei device driver non è gestito, nell'area di RAM loro allocata, come una stringa C (in altre parole, non è seguito dal NULL). La _fstrncmp() restituisce 0 se le stringhe confrontate sono uguali, cioè nel caso in cui il driver sia installato: l'operatore di not logico ("!"; vedere pag. 63) "capovolge" il risultato, perciò anche questa versione di testEMM() restituisce un valore non nullo se il driver è presente e 0 se non lo è (circa MK_FP() vedere pag. 24).

La versione del gestore EMM installato è ottenibile via int 67h, servizio 46h, sul quale si basa la getEMMversion().

INT 67H, SERV. 46H: VERSIONE EMM

Input	AH	46h
Output	AH AL	Stato dell'operazione (errore se != 0; vedere pag. 225). Versione e revisione del gestore EMM. La versione è nei bit 4-7, la revisione nei bit 0-3.

```
/********************

    BARNINGA_Z! - 1991

    EMMVER.C - getEMMversion()
```

[220] Tra l'altro questo algoritmo è facilmente implementabile anche all'interno di gestori di interrupt.

```
    unsigned cdecl getEMMversion(void);
    Restituisce: la versione del driver EMM (versione nel byte meno
                 significativo, revisione nel byte piu' significativo:
                 4.0 e' restituito come 0x0004).
                 Se si e' verificato un errore restituisce un numero
                 negativo (il codice d'errore EMS cambiato di segno).

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- emmver.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-

unsigned cdecl getEMMversion(void)                    /* ottiene la versione LIM */
{
    asm {
        mov ah,0x46;
        int 0x67;
        cmp ah,0;
        je SETVER;
        mov al,ah;
        xor ah,ah;
        neg ax;
        jmp EXITFUNC;
    }
SETVER:
    asm {
        mov cx,4;
        mov ah,al;
        shr al,cl;
        and ax,0F0Fh;
    }
EXITFUNC:
    return(_AX);
}
```

La `getEMMversion()` restituisce un unsigned integer, il cui byte meno significativo rappresenta la versione, e quello più significativo la revisione del gestore EMM[221]. In caso di errore è restituito un valore negativo (il codice di errore cambiato di segno).

La funzione che segue, `getEMMframeAddr()`, restituisce l'indirizzo della page frame, ottenuto invocando servizio 41h dell'int 67h.

INT 67H, SERV. 41H: INDIRIZZO DELLA PAGE FRAME

Input	AH	41h
Output	AH BX	Stato dell'operazione (errore se != 0; pag. 225). Indirizzo (segmento) della page frame.

[221] Il valore restituito dall'int 67h è "risistemato" in modo coerente con il servizio 30h dell'int 21h, che restituisce in AL la versione e in AH la revisione del DOS.

```
/********************

    BARNINGA_Z! - 1991

    EMMFRAME.C - getEMMframeAddr()

    unsigned cdecl getEMMframeAddr(void);
    Restituisce: l'indirizzo (segmento) della Page Frame.
                 Se si e' verificato un errore restituisce un numero
                 negativo (il codice d'errore EMS cambiato di segno).

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- emmframe.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-                  /* non indispensabile; accresce l'efficienza */

unsigned cdecl getEMMframeAddr(void)
{
    asm {
        mov ah,0x41;
        int 0x67;
        cmp ah,0;
        je EXITFUNC;
        mov al,ah;
        xor ah,ah;
        neg ax;
        mov bx,ax;
    }
EXITFUNC:
    return(_BX);
}
```

Ancora, attraverso i servizi dell'int 67h, è possibile conoscere lo stato della memoria espansa (numero di pagine totali e libere, stato degli handle, etc.).

INT 67H, SERV. 42H: NUMERO DI PAGINE

Input	AH	42h
Output	AH	Stato dell'operazione (errore se != 0; pag. 225).
	BX	Numero di pagine non allocate.
	DX	Numero totale di pagine EMS.

```
/********************

    BARNINGA_Z! - 1991

    EMMTOTP.C - getEMMtotPages()

    int cdecl getEMMtotPages(void);
    Restituisce: il numero di pagine totali EMS
                 Se < 0 si e' verificato un errore; il valore cambiato di segno
                 e' il codice di errore EMS
```

```
    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmtotp.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

int cdecl getEMMtotPages(void)
{
    struct REGPACK r;

    r.r_ax = 0x4200;
    intr(0x67,&r);
    if(!(r.r_ax & 0xFF00))
        return(r.r_dx);
    return(-((r.r_ax >> 8) & 0xFF));
}
```

La `getEMMtotPages()` restituisce il numero di pagine logiche EMS disponibili nel sistema: se il valore restituito è negativo rappresenta, cambiato di segno, il codice di errore EMS (l'operazione non è stata eseguita correttamente); inoltre il dato restituito può evidenziare una quantità di memoria EMS maggiore della quantità di memoria fisica installata sulla macchina, quando sia attivo un ambiente in grado di creare memoria virtuale[222]. Analoghe considerazioni valgono per la `getEMMfreePages()`, che restituisce il numero di pagine logiche EMS non ancora allocate (e quindi disponibili per i programmi).

```
/********************

    BARNINGA_Z! - 1991

    EMMFREEP.C - getEMMfreePages()

    int cdecl getEMMfreePages(void);
    Restituisce: il numero di pagine libere EMS
                 Se < 0 si e' verificato un errore; il valore cambiato di segno
                 e' il codice di errore EMS

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmfreep.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

int cdecl getEMMfreePages(void)
{
    struct REGPACK r;

    r.r_ax = 0x4200;
    intr(0x67,&r);
    if(!(r.r_ax & 0xFF00))
        return(r.r_bx);
```

[222] In grado, cioè, di utilizzare porzioni dello spazio libero su disco come RAM aggiuntiva. E' il caso, ad esempio, di Microsoft Windows 3.x su macchine 80386 o superiori, se attivo in modalità "80386 Avanzata".

```
    return(-((r.r_ax >> 8) & 0xFF));
}
```

INT 67H, SERV. 4BH: NUMERO DI HANDLE EMM APERTI

Input	AH	4Bh
Output	AH BX	Stato dell'operazione (errore se != 0; pag. 225). Numero di handle aperti.

INT 67H, SERV. 4DH: PAGINE ALLOCATE AGLI HANDLE

Input	AH ES:DI	4Dh Buffer costituito da tante coppie di word quanti sono gli handle aperti.
Output	AH BX	Stato dell'operazione (errore se != 0; pag. 225). Numero di handle attivi. Le words dell'array in ES:DI sono riempite, alternativamente, con un numero di handle e il numero delle pagine allocate a quello handle.

```
/********************

    BARNINGA_Z! - 1991

    EMMOHNDL.C - getEMMopenHandles()

    int cdecl getEMMopenHandles(void);
    Restituisce: il numero di handles EMS aperti
                 Se < 0 si e' verificato un errore; il valore cambiato di segno
                 e' il codice di errore EMS

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmohndl.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

int cdecl getEMMopenHandles(void)
{
    struct REGPACK r;

    r.r_ax = 0x4B00;
    intr(0x67,&r);
    if(!(r.r_ax & 0xFF00))
        return(r.r_bx);
    return(-((r.r_ax >> 8) & 0xFF));
}
```

Anche la `getEMMopenHandles()` in caso di errore restituisce il codice d'errore EMS cambiato di segno (negativo); un valore maggiore o uguale a `0` esprime invece il numero di handle EMS aperti, cioè utilizzati nel sistema. La `getEMMpagesPerHandle()` utilizza invece il servizio 4Dh dell'int 67h per conoscere il numero di pagine allocate a ciascuno handle aperto e memorizza i dati nell'array di strutture di tipo `EMMhnd`, il cui indirizzo le è passato come parametro.

```
struct EMMhnd {                         // struttura per i dati relativi agli handles
    int       handle;                          // n. dello handle per Expanded Memory
    unsigned  pages;                                  // pagine assegnate allo handle
};
```

L'array di strutture deve essere allocato a cura del programmatore, ad esempio con una chiamata a `malloc()`:

```
    struct EMMhnd *emmHnd;
    unsigned oHnd;
    ....
    if((oHnd = getEMMopenHandles()) < 0) {
        ....                                              // gestione dell'errore EMS
    }
    else
        if(!(emmHnd = (struct EMMhnd *)malloc(oHnd*sizeof(struct EMMhnd)))) {
            ....                               // gestione dell'errore di allocazione
        }
```

In assenza di errori può essere invocata la `getEMMpagesPerHandle()`, listata di seguito, passandole come parametro il puntatore `emmHnd`.

```
/********************

    BARNINGA_Z! - 1991

    EMMPPH.C - getEMMpagesPerHandle()

    int cdecl getEMMpagesPerHandle(struct EMMhnd *emmHnd);
    struct EMMhnd *emmHnd;   puntatore ad array di strutture EMMhnd, gia' allocato
                             con tanti elementi quanti sono gli handles aperti.
    Restituisce: >= 0 se l'operazione e' stata eseguita correttamente. Il valore
                 rappresenta il numero di handles EMS attivi.
                 Se < 0 si e' verificato un errore; il valore cambiato di segno
                 e' il codice di errore EMS

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmpph.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  warn -pia

#include <dos.h>
#include <errno.h>

int cdecl getEMMpagesPerHandle(struct EMMhnd *emmHnd)   // uso della memoria espansa
{
    struct REGPACK r;

    r.r_ax = 0x4D00;
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
```

```
    r.r_es = _DS;
    r.r_di = (unsigned)emmHnd;
#else
    r.r_es = FP_SEG(emmHnd);
    r.r_di = FP_OFF(emmHnd);
#endif
    intr(0x67,&r);
    if(!(r.r_ax & 0xFF00))
        return(r.r_bx);
    return(-((r.r_ax >> 8) & 0xFF));
}
```

Circa l'uso di `DS` nella funzione vedere pag. 196; le macro `FP_SEG()` e `FP_OFF()` sono descritte a pag. 24.

Ad ogni handle può essere associato un nome, mediante la subfunzione 1 del servizio 53h dell'int 67h. I nomi associati agli handle aperti possono essere conosciuti tramite la subfunzione 0 del medesimo servizio.

INT 67H, SERV. 53H: NOME DELLO HANDLE EMS

Input	`AH`	`53h`
	`AL`	`00h`: richiede il nome di uno handle `DX` = Numero dello handle EMM. `ES:DI` = Indirizzo di un buffer ampio almeno 8 byte, in cui è copiato il nome dello handle. 01h: assegna il nome ad uno handle `DX` = Numero dello handle EMM `DS:SI` = Indirizzo di un buffer ampio almeno 8 byte, contenente il nome da assegnare allo handle (ASCIIZ).
Output	`AH`	Stato dell'operazione (errore se `!=` 0; pag. 225).

```
/********************

    BARNINGA_Z! - 1991

    EMMGHNAM.C - getEMMhandleName()

    int cdecl getEMMhandleName(unsigned handle,char *EMMhname);
    int EMMhandle;      handle EMM di cui si vuole conoscere il nome.
    char *EMMhname;     buffer di 9 bytes in cui memorizzare il nome.
    Restituisce: lo stato dell'operazione. Se < 0, il valore, cambiato di segno,
                 e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmghnam.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

#include <string.h>
```

```
int cdecl getEMMhandleName(int EMMhandle,char *EMMhname)
{
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm push ds;
    asm pop es;
    asm mov di,EMMhname;
#else
    asm les di,dword ptr EMMhname;
#endif
    asm mov dx,EMMhandle;
    asm mov ax,0x5300;
    asm int 0x67;
    asm cmp ah,00h;
    asm jne EXITERROR;
    EMMhname[8] = NULL;
    return(0);
EXITERROR:
    asm mov al,ah;
    asm xor ah,ah;
    asm neg ax;
    return(_AX);
}
```

La `getEMMhandleName()` utilizza il servizio descritto (subfunzione 0) per conoscere il nome associato allo handle `EMMhandle`. Il buffer `EMMhname` deve comprendere 9 byte[223]: nei primi 8 è copiato il nome, mentre l'ultimo è valorizzato a `NULL` per costruire una normale stringa (circa l'uso di `DS` vedere pag. 196). In caso di errore è restituito un valore negativo che, cambiato di segno, rappresenta il codice dell'errore EMS: la funzione non potrebbe segnalare l'errore semplicemente copiando in `EMMhname` una stringa vuota, dal momento che questa è un nome valido. La differente gestione dei puntatori nei diversi modelli di memoria rende necessaria, come già in `getEMMusage()`, la compilazione condizionale delle istruzioni relative al caricamento dei registri `ES:DI` con l'indirizzo del buffer.

Del tutto analoga appare la `setEMMhandleName()`, che assegna un nome ad uno handle tramite la subfunzione 1 del solito int 67h, servizio 53h.

```
/********************

    BARNINGA_Z! - 1991

    EMMSHNAM.C - setEMMhandleName()

    int cdecl setEMMhandleName(unsigned handle,char *EMMhname);
    int EMMhandle;     handle EMM a cui si vuole assegnare il nome.
    char *EMMhname;    buffer contenente il nome (stringa chiusa da NULL).
    Restituisce: lo stato dell'operazione. Se < 0, il valore, cambiato di segno,
                 e' il codice di errore EMS. Se e' 0, l'operazione e' stata
                 eseguita correttamente.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmshnam.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
```

[223] Se la sua dimensione supera i 9 byte, sono comunque utilizzati solamente i primi 9.

```
#include <string.h>

int cdecl setEMMhandleName(int EMMhandle,char *EMMhname)
{
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm mov si,EMMhname;
#else
    asm push ds;
    asm lds si,dword ptr EMMhname;
#endif
    asm mov dx,EMMhandle;
    asm mov ax,0x5301;
    asm int 0x67;
    asm mov al,ah;
    asm cmp ax,00h;
    asm je EXITFUNC;
    asm xor ah,ah;
    asm neg ax;
EXITFUNC:
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    return(_AX);
}
```

Le funzioni sin qui presentate (eccetto `setEMMhandleName()`) consentono esclusivamente di analizzare l'attuale utilizzo della expanded memory: si tratta ora di definire (in breve!) un algoritmo di utilizzo della medesima, per dotarci degli strumenti che ci consentano di sfruttarla attivamente nei nostri programmi. In primo luogo è necessario allocare un certo numero di pagine logiche ad uno handle; in altre parole dobbiamo stabilire quanta memoria espansa ci occorre, tenendo presente che essa è solitamente allocata, per compatitbilità con le prime versioni delle specifiche LIM EMS, in blocchi multipli di 16 Kb (le pagine). Così, se dobbiamo disporre di 40 Kb di expanded memory, è indispensabile allocare 3 pagine logiche. Il driver EMM assegna al gruppo di pagine logiche allocate un numero identificativo (lo handle) al quale è necessario riferirsi per tutte le operazione successivamente effettuate su di esse.

Allocare pagine logiche significa destinarle ad uso esclusivo del programma. L'area di memoria espansa è quindi identificata da due "coordinate": lo handle e il numero di pagina logica all'interno dell'area stessa[224]. L'allocazione delle pagine logiche è effettuata dal servizio 43h dell'int 67h:

INT 67H, SERV. 43H: ALLOCA PAGINE LOGICHE NELLA MEMORIA ESPANSA

Input	AH	43h
	BX	numero di pagine logiche che si desidera allocare
Output	AH	Stato dell'operazione (errore se != 0; pag. 225).
	DX	Handle (se AX = 0)

[224] Se, ad esempio, si richiede al driver di allocare un gruppo di 3 pagine logiche, queste saranno numerate da 0 a 2. Ciascuna di esse è identificabile univocamente mediante lo handle e il proprio numero. Nonostante il paragone sia un po' azzardato, si può pensare ad un blocco di pagine logiche come ad un array: lo handle identifica l'array stesso, ed ogni pagina ne è un elemento, che può essere referenziato tramite il proprio numero (l'indice).

```
/********************

    BARNINGA_Z! - 1991

    EMMALLOC.C - allocEMMpages()

    int cdecl allocEMMpages(int pages);
    int pages;     numero di pagine da allocare.
    Restituisce:   un intero. Se e' maggiore di zero e' lo handle associato
                   alle pagine allocate. Se e' = zero si e' verificato un errore
                   non identificato. Se < 0, il valore, cambiato di segno,
                   e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmalloc.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl allocEMMpages(int pages)
{
    asm mov ah,043h;
    asm mov bx,pages;
    asm int 067h;
    asm cmp ah,00h;
    asm je EXITFUNC;
    asm mov dl,ah;
    asm xor dh,dh;
    asm neg dx;
EXITFUNC:
    return(_DX);
}
```

La `allocEMMpages()` restituisce lo handle associato alle pagine allocate: si tratta di un numero maggiore di `0`. Se viene restituito un numero pari o inferiore a zero, si è verificato un errore e le pagine non sono state allocate: il valore zero non è un codice di errore vero e proprio, ma indica una situazione "strana", in quanto lo handle `0` è riservato al sistema operativo. Un valore minore di zero, cambiato di segno, rappresenta invece un normale codice di errore EMS. Una tabella dei codici di errore è presentata a pag. 225.

Come utilizzare le pagine disponibili? Il driver è in grado di effettuare un mapping trasparente delle pagine logiche con le pagine fisiche. Ciò significa che tutte le operazioni effettuate su queste ultime (definite all'interno del primo Mb, dunque indirizzabili mediante comuni puntatori di tipo `far` o `huge`) vengono trasferite, senza che il programmatore debba preoccuparsene, sulle pagine logiche poste in corrispondenza (cioè mappate) con esse. Effettuare il mapping di una pagina logica con una pagina fisica significa, in pratica, porre la prima in corrispondenza biunivoca con la seconda: è il driver a riportare nella pagina logica tutte le modifiche apportate dal programma al contenuto della pagina fisica. Vediamo una possibile implementazione del servizio 44h dell'int 67h.

INT 67H, SERV. 44H: EFFETTUA IL MAPPING DI PAGINE LOGICHE A PAGINE FISICHE

Input	AH	44h
	AL	numero della pagina fisica su cui mappare la pagina logica
	BX	numero della pagina logica da mappare su quella fisica
	DX	handle associato alla pagina logica (o al gruppo di pagine logiche di cui questa fa parte)
Output	AH	Stato dell'operazione (errore se != 0; pag. 225).

```
/********************

    BARNINGA_Z! - 1991

    EMMPGMAP.C - mapEMMpages()

    int cdecl mapEMMpages(int physicalPage,int logicalPage,int EMMhandle);
    int physicalPage;     numero della pag. fisica su cui mappare la pag. logica
    int logicalPage;      numero della pag. logica da mappare su quella fisica
    int EMMhandle;        handle associato alla pagina logica
    Restituisce:   0 se l'operazione e' stata eseguita correttamente
                   Se < 0, il valore, cambiato di segno, e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmpgmap.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl mapEMMpages(int physicalPage,int logicalPage,int EMMhandle)
{
    asm mov dx,EMMhandle;
    asm mov bx,logicalPage;
    asm mov al,byte ptr physicalPage;
    asm mov ah,044h;
    asm int 067h;
    asm mov al,ah;
    asm cmp ax,00h;
    asm je EXITFUNC;
    asm xor ah,ah;
    asm neg ax;
EXITFUNC:
    return(_AX);
}
```

Va ancora precisato che una medesima pagina fisica può essere utilizzata, senza difficoltà operative, per effettuare il mapping di più pagine logiche, purché in momenti diversi. Per fare un esempio concreto, possiamo mappare[225] alla pagina fisica 0 la pagina logica 2 associata ad un certo handle e memorizzare in quella pagina fisica i nostri dati, secondo necessità. Successivamente è possibile mappare

[225] Mappare.... orrendo!

alla stessa pagina fisica 0 un'altra pagina logica, associata o no al medesimo handle. I dati che erano presenti nella pagina fisica 0 non vengono persi, perché il driver, in modo trasparente, riflette le operazioni sulla pagina logica associata, e quindi essi sono memorizzati in quest'ultima. E' facile constatare che effettuando nuovamente il mapping della prima pagina logica dell'esempio alla solita pagina fisica 0 ritroviamo in quest'ultima i dati originariamente memorizzati.

Confusi? Niente paura, un programmino ci aiuterà a capire... tuttavia, dobbiamo ancora discutere un dettaglio: la deallocazione della memoria espansa. E' importante ricordare che i programmi, prima di terminare, devono sempre disallocare esplicitamente la memoria espansa allocata. Il DOS, infatti, non si immischia affatto nella gestione EMS (e, del resto, il driver EMS e il sistema operativo non interagiscono se non nella fase del caricamento del primo da parte del secondo durante bootstrap): tutta la memoria espansa allocata da un programma e dal medesimo non rilasciata è inutilizzabile per tutti gli altri programmi (e per il DOS medesimo) fino al successivo bootstrap della macchina.

INT 67H, SERV. 45H: DEALLOCA LE PAGINE ASSOCIATE AD UNO HANDLE

Input	AH	45h
	DX	handle associato alla pagina logica (o gruppo di pagine logiche) da deallocare (sono sempre disallocate tutte le pagine logiche associate allo handle)
Output	AH	Stato dell'operazione (errore se != 0; pag. 225).

```
/********************

    BARNINGA_Z! - 1991

    EMMFREEH.C - freeEMMhandle()

    int cdecl freeEMMhandle(int EMMhandle);
    int EMMhandle;        handle associato alle pagine logiche da rilasciare.
    Restituisce:   0 se l'operazione e' stata eseguita correttamente
                   Se < 0, il valore, cambiato di segno, e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmfreeh.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl freeEMMhandle(int EMMhandle)
{
    asm mov ah,045h;
    asm mov dx,EMMhandle;
    asm int 067h;
    asm mov al,ah;
    asm cmp ax,00h;
    asm je EXITFUNC;
    asm xor ah,ah;
    asm neg ax;
EXITFUNC:
    return(_AX);
}
```

Tutte la pagine associate allo handle passato a `freeEMMhandle()` sono disallocate e lo handle rilasciato: questo non è più utilizzabile per alcuna operazione EMS. Anche la `freeEMMhandle()` restituisce `0` se l'operazione è stata eseguita correttamente e, in caso di errore, un valore minore di zero che, cambiato di segno, rappresenta il codice di errore.

E' tempo di affrontare la... realtà: di seguito è listato il programma che utilizza alcune delle funzioni presentate sin qui per effettuare operazioni di allocazione, utilizzo e deallocazione della memoria espansa.

```
/*******************************************************************************

    EMS.C - Barninga Z! - 1994

    Programma di test per alcune funzioni di servizio EMS.
    Il sorgente non include i listati delle funzioni.

    Compilato con Borland C++ 3.1

    bcc ems.c

*******************************************************************************/
#pragma  warn -pia

#include <stdio.h>
#include <dos.h>
#include <string.h>

int      allocEMMpages(int pages);
int      freeEMMhandle(int EMMhandle);
unsigned getEMMframeAddr(void);
int      getEMMhandleName(int EMMhandle,char *EMMhname);
int      mapEMMpages(int physicalPage,int logicalPage,int EMMhandle);
int      setEMMhandleName(int EMMhandle,char *EMMhname);

void main(void)
{
    register i, j;
    int retcode;
    unsigned frameSeg;
    char far *physicalPage0;
    char strBuf[80];
    int EMMhandles[2];
    static char *EMMhandleNames[] = {"Handle_1","Handle_2"};

    frameSeg = getEMMframeAddr();
    physicalPage0 = (char far *)MK_FP(frameSeg,0);
    printf("Page Frame Address: %Fp\n",physicalPage0);
    for(i = 0; i < 2; i++) {
        if((EMMhandles[i] = allocEMMpages(2)) <= 0) {
            printf("Error %X allocating 2 EMM pages.\n",-EMMhandles[i]);
            break;
        }
        if(retcode = setEMMhandleName(EMMhandles[i],EMMhandleNames[i])) {
            printf("Error %X setting name of handle %d.\n",-retcode,EMMhandles[i]);
            break;
        }
    }
    if(i < 2) {
        for(j = 0; j < i; j++)
            if(retcode = freeEMMhandle(EMMhandles[j]))
                printf("Error %X deallocating EMM pages of handle %d.\n",-retcode,
                                                                    EMMhandles[j]);
        return;
```

```
    }
    for(i = 0; i < 2; i++)
        if(retcode = getEMMhandleName(EMMhandles[i],strBuf))
            printf("Error %X getting name of handle %d.\n",-retcode,EMMhandles[i]);
        else
            printf("Handle %d name: %s\n",EMMhandles[i],buffer);
    if(retcode = mapEMMpages(0,0,EMMhandles[0]))
        printf("Error %X mapping Log. page %d of handle %d to Phys. page %d.\n",
                                                  -retcode,0,EMMhandles[0],0);
    _fstrcpy(physicalPage0,"@@@ A String For Logical Page 0 @@@");
    if(retcode = mapEMMpages(0,1,EMMhandles[1]))
        printf("Error %X mapping Log. page %d of handle %d to Phys. page %d.\n",
                                                  -retcode,1,EMMhandles[1],0);
    _fstrcpy(physicalPage0,"XXXXXXXXXXXXXXXXXXXX");
    if(retcode = mapEMMpages(0,0,EMMhandles[0]))
        printf("Error %X mapping Log. page %d of handle %d to Phys. page %d.\n",
                                                  -retcode,0,EMMhandles[0],0);
    _fstrcpy(strBuf,physicalPage0);
    printf("Logical Page 0 content: %s\n",strBuf);
    for(i = 0; i < 2; i++)
        if(retcode = freeEMMhandle(EMMhandles[i]))
            printf("Error %X deallocating EMM pages of handle %d.\n",-retcode,
                                                              EMMhandles[i]);
}
```

In testa al programma sono dichiarati i prototipi delle funzioni descritte in precedenza (si presume che esse si trovino, già compilate, in un object file o una libreria da specificare in fase di compilazione e linking). La prima operazione svolta da `main()` consiste nel chiamare la `getEMMframeAddr()` per conoscere l'indirizzo di segmento della page frame. Dato che le quattro pagine fisiche in cui essa è suddivisa sono numerate in ordine crescente a partire dalla sua "origine", l'indirizzo della page frame coincide con quello della pagina fisica `0`. Si tratta, ovviamente, di un indirizzo del tipo *seg:off* (vedere pag. 189), che viene ottenuto "accostando", mediante la macro `MK_FP()` definita in `DOS.H` (pag. 24), un offset pari a zero al segmento restituito da `getEMMframeAddr()`.

Il ciclo `for` successivo gestisce l'allocazione di due blocchi di memoria espansa, ciascuno costituito di `2` pagine logiche; `2` è infatti il parametro passato a `allocEMMpages()`, che, ad ogni chiamata, restituisce lo handle associato al singolo blocco di pagine logiche. Gli handle sono memorizzati negli elementi dell'array di interi `EMMhandles`. Se il valore restituito è minore o uguale a `0` si è verificato un errore: viene visualizzato un apposito messaggio e il ciclo di allocazione è interrotto (`break`). Nello stesso ciclo ad ogni handle allocato è assegnato un nome, mediante la `setEMMhandleName()`: i nomi (normali stringhe) per gli handle sono contenuti nell'array `EMMhandleNames`; anche in questo caso la restituzione di un codice di errore determina l'interruzione del ciclo. Se in uscita dal ciclo `i` è minore di `2`, significa che esso è stato interrotto (da un errore): il programma termina, ma prima vengono rilasciati gli handle eventualmente allocati (di fatto, se vi è stato un errore, gli handle allocati sono uno solo, o nessuno).

Il flusso elaborativo prosegue con un altro ciclo `for`, all'interno del quale sono visualizzati, ricavandoli tramite la `getEMMhandleNames()`, i nomi precedentemente assegnati agli handle; va inoltre precisato che non è affatto necessario assegnare un nome ad ogni handle per utilizzare le pagine logiche ad esso associate. Il nostro programma non perde alcuna delle sue (strabilianti) funzionalità, pur eliminando le chiamate a `setEMMhandleNames()` e `getEMMhandleNames()`. Va ancora precisato che una chiamata a `getEMMhandleNames()` per richiedere il nome di uno handle che ne è privo non determina un errore, ma semplicemente l'inizializzazione a stringa vuota del buffer il cui indirizzo le è passato come parametro (`EMMhname`).

Siamo così giunti alla parte più interessante del listato, quella, cioè, in cui vengono finalmente effettuati il mapping delle pagine logiche ed il loro utilizzo effettivo.

La prima chiamata a `mapEMMpages()` mappa la pagina logica `0` sulla pagina fisica `0`. Da questo momento tutte le operazioni effettuate sulla pagina fisica `0` (la prima delle quattro in cui è

suddivisa la page frame) si riflettono automaticamente su quella pagina logica. Scopo del programma è proprio verificare quanto appena affermato: si tratta di scrivere qualcosa nella pagina fisica `0`, mappare ad essa un'altra pagina logica, in cui scrivere altri dati, e poi mappare nuovamente la pagina logica attualmente associata, per verificare se i dati originariamente scritti sono ancora lì. In effetti, quelle descritte sono proprio le operazioni che il programma esegue.

La prima chiamata a `_fstrcpy()` copia nella pagina fisica `0` la stringa `"@@@ A String For Logical Page 0 @@@"`. Immediatamente dopo, la seconda chiamata a `mapEMMpages()` mappa alla solita pagina fisica `0` la seconda (`1`) delle due (`0` e `1`) pagine logiche associate al secondo handle (`EMMhandles[1]`) e la `_fstrcpy()` scrive nella pagina fisica `0` la stringa `"XXXXXXXXXXXXXXXXXXXXXX"`, che viene in realtà scritta nella pagina logica appena mappata. Il dubbio che questa operazione di scrittura possa sovrascrivere la stringa precedente (copiata allo stesso indirizzo fisico, ma non allo stesso indirizzo logico) è fugato dalle operazioni immediatamente seguenti.

Il mapping effettuato dalla terza chiamata a `mapEMMpages()` associa nuovamente alla pagina fisica `0` la prima pagina logica (`0`) delle due (`0` e `1`) allocate al primo handle (`EMMhandles[0]`). Questa volta `_fstrcpy()` copia nel buffer `strBuf`, che è `near`, la stringa che si trova all'inizio della pagina fisica `0`. L'operazione è necessaria perché `printf()`, nell'ipotesi di compilare il programma con modello di memoria tiny, small o medium (pag. 143) non può accettare come parametro l'indirizzo `far` della page frame. Il momento della verità è giunto: `printf()` visualizza `"@@@ A String For Logical Page 0 @@@"`, proprio come previsto.

Il programma si chiude con un ciclo che chiama `freeEMMhandle()`, per rilasciare tutta la memoria espansa allocata, al fine di renderla nuovamente disponibile ai processi eseguiti successivamente.

Non crediate di cavarvela così facilmente: i guai, con la memoria EMS, non finiscono qui. Il programma presentato poco fa lavora nella presunzione, peraltro fondata, che nessuno gli mescoli le carte in tavola: in altre parole assume che la situazione di mapping non venga modificata da eventi esterni. In realtà ciò può avvenire in almeno due situazioni: se il programma lancia un'altra applicazione che utilizza memoria EMS oppure se è un TSR (pag. 275) a farne un uso poco corretto[226]. Va da sé che nel caso di un TSR o un device driver "maleducato" c'è poco da fare, ma se sappiamo in partenza che un evento generato dal nostro programma può modificare la mappatura delle pagine EMS è meglio correre ai ripari, salvando lo stato della mappatura delle pagine logiche su quelle fisiche (*page map*). Niente di tremendo: l'int 67h mette a disposizione un servizio progettato proprio per questo.

[226] Alcune notizie relative all'utilizzo della memoria EMS nei TSR sono date a pag. 283.

INT 67H, SERV. 4EH: SALVA E RIPRISTINA LA PAGE MAP

Input	AH	`4Eh`
	AL	`00h` salva la page map `ES:DI` = punta a un buffer in cui salvare la page map `01h` rispristina la page map `DS:SI` = punta a un buffer da cui caricare la page map `02h` salva page map e ne carica una salvata precedentemente `DS:SI` = punta a un buffer da cui caricare la page map `ES:DI` = punta a un buffer in cui salvare la page map `03h` restituisce la dimensione del buffer per la page map
Output	AH	Stato dell'operazione (errore se `!= 0`; pag. 225).
	AL	Solo per subfunzione `03h`: dimensione in byte del buffer necessario a contenere la page map

Ecco alcuni esempi di funzioni basate sul servizio 4Eh dell'int 67h:

```
/********************

    BARNINGA_Z! - 1991

    EMMGPMD.C - getEMMpageMapDim()

    int cdecl getEMMpageMapDim(void);
    Restituisce:   Se > 0 è la dimensione in bytes dell'array necessario a
                   contenere la page map
                   Se < 0, il valore, cambiato di segno, e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmgpmd.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl getEMMpageMapDim(void)
{
    asm mov ax,04E03h;
    asm int 067h;
    asm mov bl,al;
    asm mov al,ah;
    asm xor ah,ah;
    asm cmp ax,0;
    asm je EXITFUNC;
    asm neg ax;
    return(_AX);
EXITFUNC:
```

```
    return(_BL);
}
```

La `getEMMpageMapDim()` deve essere chiamata per conoscere la dimensione del buffer che deve contenere la page map; questo può essere allocato, ad esempio, tramite `malloc()`, prima di chiamare la `saveEMMpageMap()`, listata di seguito (circa l'uso di `DS` vedere pag. 196).

```
/********************

    BARNINGA_Z! - 1991

    EMMSPM.C - saveEMMpageMap(unsigned char *destBuf)

    int cdecl saveEMMpageMap(unsigend char *destBuf);
    unsigned char *destBuf;     buffer che conterra' la page map.
    Restituisce:   Se 0 l'operazione è stata eseguita correttamente
                   Se < 0, il valore, cambiato di segno, e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmspm.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl saveEMMpageMap(unsigned char *destBuf)
{
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm push ds;
    asm pop es;
    asm mov di,destBuf;
#else
    asm les di,dword ptr destBuf;
#endif
    asm mov ax,04E00h;
    asm int 067h;
    asm mov al,ah;
    asm cmp ax,00h;
    asm je EXITFUNC;
    asm xor ah,ah;
    asm neg ax;
EXITFUNC:
    return(_AX);
}
```

Dopo avere salvato la page map il programma può lanciare il child process, senza preoccuparsi di come esso utilizza la memoria EMS. Al rientro è sufficiente chiamare la `restoreEMMpageMap()` per riportare lo stato del driver alla situazione salvata.

```
/********************

    BARNINGA_Z! - 1991

    EMMRPM.C - restoreEMMpageMap(unsigned char *sourceBuf)

    int cdecl saveEMMpageMap(unsigned char *sourceBuf);
    unsigned char *sourceBuf;     buffer che contiene la page map da riattivare.
    Restituisce:   Se 0 l'operazione è stata eseguita correttamente
                   Se < 0, il valore, cambiato di segno, e' il codice di errore EMS.
```

```
    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx emmrpm.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl restoreEMMpageMap(unsigned char *sourceBuf)
{
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm mov si,sourceBuf;
#else
    asm push ds;
    asm lds si,dword ptr sourceBuf;
#endif
    asm mov ax,04E00h;
    asm int 067h;
    asm mov al,ah;
    asm cmp ax,00h;
    asm je EXITFUNC;
    asm xor ah,ah;
    asm neg ax;
EXITFUNC:
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    return(_AX);
}
```

Per un esempio di funzione basata sulla subfunzione 02h, vedere pag. 283.

Con la versione 4.0 dello standard LIM sono state introdotte interessanti funzionalità di trasferimento dati tra memoria espansa e aree di memoria convenzionale, rendendo così possibile "scavalcare" la page frame. Il servizio che implementa tale funzionalità è il 57h; la subfunzione 00h consente di trasferire dati direttamente dalla memoria convenzionale alla memoria EMS e richiede che i registri `DS:SI` siano caricati con l'indirizzo di un buffer contenente le informazioni necessarie per effettuare il trasferimento. La subfunzione 01h del medesimo servizio permette invece di scambiare il contenuto di due aree di memoria, che possono essere indifferentemente situate in memoria convenzionale o espansa.

INT 67H, SERV. 57H: TRASFERISCE DA MEM. CONVENZ. A MEM. EMS E VICEVERSA

Input	AH	`57h`
	AL	`00h` trasferisce da memoria convenzionale a memoria EMS `DS:SI` = punta al buffer di info per il trasferimento `01h` scambia il contenuto di due aree di memoria `DS:SI` = punta al buffer di info per lo scambio
Output	AH	Stato dell'operazione (errore se `!= 0`; pag. 225).

Il formato del buffer richiesto dal servizio analizzato è il seguente:

FORMATO DEL BUFFER UTILIZZATO DALL'INT 67H, SERV. 57H

OFFSET	BYTE	DESCRIZIONE
`00h`	4	Lunghezza in byte dell'area di memoria da trasferire
`04h`	1	Tipo della memoria sorgente (`0` = convenzionale; `1` = EMS)
`05h`	2	Handle EMS sorgente (`0` se memoria convenzionale)
`07h`	2	Offset dell'area sorgente (rispetto al segmento se memoria convenzionale; rispetto alla pagina logica se memoria EMS)
`09h`	2	Segmento dell'area sorgente (se memoria convenzionale) o pagina logica (se memoria EMS)
`0Bh`	1	Tipo della memoria destinazione (`0` = convenzionale; `1` = EMS)
`0Ch`	2	Handle EMS destinazione (`0` se memoria convenzionale)
`0Eh`	2	Offset dell'area destinazione (rispetto al segmento se memoria convenzionale; rispetto alla pagina logica se memoria EMS)
`10h`	2	Segmento dell'area destinazione (se memoria convenzionale) o pagina logica (se memoria EMS)

Il buffer può essere facilmente rappresentato con una struttura:

```
struct EMMmove {
    unsigned long length;
    unsigned char sourceType;
    unsigned int  sourceHandle;
    unsigned int  sourceOffset;
    unsigned int  sourceSegment;
    unsigned char destType;
    unsigned int  destHandle;
    unsigned int  destOffset;
    unsigned int  destSegment;
};
```

L'esempio seguente copia direttamente 32 Kb dal buffer video colore (`B800:0000`) ad offset `256` nella terza pagina del blocco EMS allocato allo handle `09h`.

```
#include <dos.h>
    ....
    struct REGPACK r;
    struct EMMmove EMMinfoBuf;
    ....
    EMMinfoBuf.length = 32*1024;                                    // 32 Kb
    EMMinfoBuf.sourceType = 0;                  // sorgente: memoria convenzionale
    EMMinfoBuf.sourceHandle = 0;               // handle sempre 0 per mem. convenz.
    EMMinfoBuf.sourceOffset = 0;                                // segmento:offset
    EMMinfoBuf.sourceSegment = 0xB800;                                // B800:0000
```

```
    EMMinfoBuf.destType = 1;                            // destinazione: memoria EMS
    EMMinfoBuf.destHandle = 0x09;               // handle (da int 67h, servizio 43h)
    EMMinfoBuf.destOffset = 256;              // offset all'interno della pag.logica
    EMMinfoBuf.destSegment = 2;                                  // terza pag.logica
    r.r_ax = 0x5700;
    r.r_ds = FP_SEG((struct EMMmove far *)&EMMinfoBuf);
    r.r_si = (unsigned)&EMMinfoBuf;
    intr(0x67,&r);
    if(r.r_ax & 0xFF00) {
        ....                                       // Gestione dell'errore (AH != 0)
```

Per effettuare l'operazione inversa è sufficiente scambiare tra loro i valori dei campi "`source...`" e "`dest...`" omologhi nella struttura `EMMinfoBlock`.

Vediamo un esempio di funzione basata sul servizio testè descritto.

```
/********************

    BARNINGA_Z! - 1992

    EMMMOVM.C - EMMmoveMem()

    int EMMmoveMem(struct EMMmove *EMMbuffer,int operation);
    struct EMMmove *EMMbuffer;     puntatore al buffer che descrive l'operaz.
    int             operation;     0 = sposta il contenuto della memoria
                                   1 = scambia il contenuto di due aree di
                                       memoria
    Restituisce:   0 se l'operazione e' stata eseguita correttamente.
                  -1 parametro operation errato
                 > 0 e' il codice di errore EMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- emmmovm.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl EMMmoveMem(struct EMMmove *EMMbuffer,int operation)
{
    if((operation < 0) || (operation > 1))
        return(-1);
    _AL = (char)operation;
    _AH = 0x57;
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm mov si,EMMbuffer;
#else
    asm push ds;
    asm lds si,dword ptr EMMbuffer;
#endif
    asm int 067h;
#if defined(__COMAPCT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    asm xor al,al;
    asm cmp ah,0;
    asm je EXITFUNC;
    asm xchg ah,al;
EXITFUNC:
    return(_AX);
}
```

La `EMMmoveMem()` riceve due parametri: il primo è il puntatore alla struttura `EMMmove`; il secondo è un intero che, a seconda del valore assunto, stabilisce se debba essere effettuata un'operazione di spostamento del contenuto di un'area di memoria (`operation` = `0`) o di scambio del contenuto di due aree (`operation` = `1`). La funzione restituisce `0` se non si è verificato alcun errore, mentre è restituito `-1` se il parametro `operation` contiene un valore illecito; la restituzione di un valore maggiore di `0` indica che si è verificato un errore EMS (del quale il valore stesso rappresenta il codice).

Le specifiche LIM 4.0 hanno introdotto un altro servizio degno di nota: il mapping di pagine multiple. Il servizio 44h dell'int 67h, descritto poco sopra, si limita ad effettuare il mapping di una sola pagina logica su una pagina fisica. Se il programma deve agire su una quantità di dati superiore ai 16 Kb è necessario richiamare più volte il servizio stesso, ad esempio all'interno di un ciclo, per mappare tutte le pagine necessarie (al massimo 4). La subfunzione 00h del servizio 50h supera detta limitazione e consente di mappare in un'unica operazione fino a 4 pagine logiche, non necessariamente consecutive, su 4 diverse pagine fisiche, anch'esse non consecutive. La subfunzione 01h consente di effettuare il mapping delle pagine logiche direttamente su aree di memoria convenzionale, senza necessità di utilizzare la page frame (pagine fisiche).

INT 67H, SERV. 50H: MAPPING MULTIPLO E SU MEMORIA CONVENZIONALE

Input	`AH`	`50h`
	`AL`	`00h` mapping di più pagine logiche su più pagine fisiche `DS:SI` = punta al buffer di info per il mapping `01h` mapping di più pagine logiche su memoria convenzionale `DS:SI` = punta al buffer di info per il mapping
	`CX`	Numero di pagine da mappare
	`DX`	Handle delle pagine logiche
Output	`AH`	Stato dell'operazione (errore se `!= 0`; pag. 225).

Il buffer di informazioni è una sequenza di 4 coppie di word (`unsigned int`). Per la subfunzione 00h, in ogni coppia di word il primo `unsigned` rappresenta il numero della pagina logica da mappare, mentre il secondo contiene quello della pagina fisica su cui deve avvenire il mapping. Esempio:

```
#include <dos.h>
....
    struct REGPACK r;
    unsigned info[8];
    ....
    info[0] = 4;                                        // mappare la pag. logica 4
    info[1] = 0;                                             // sulla pag. fisica 0
    info[2] = 6;                                        // mappare la pag. logica 6
    info[3] = 3;                                             // sulla pag. fisica 3
    info[4] = 7;                                        // mappare la pag. logica 7
    info[5] = 1;                                             // sulla pag. fisica 1
    r.r_ax = 0x5000;                               // servizio 50h, subfunzione 00h
    r.r_cx = 3;                                                // 3 pagine in tutto
    r.r_dx = 09h;                       // handle delle pag. logiche (da serv. 43h)
    r.r_ds = FP_SEG((int far *)info);
    r.r_si = (unsigned)info;
```

```
    intr(0x67,&r);
    if(r.r_ax & 0xFF00) {
        ....                                        // Gestione dell'errore (AH != 0)
```

Il codice dell'esempio richiede il mapping di tre pagine logiche (la 4, la 6 e la 7, associate allo handle 09h) rispettivamente sulle pagine fisiche 0, 3 e 1. Gli utlimi due elementi dell'array `info` non vengono inizializzati, in quanto il servizio conosce (attraverso `r.r_cx`) il numero di pagine da mappare.

Per la subfunzione 01h varia leggermente il significato dei valori contenuti nel buffer: la seconda word di ogni coppia contiene l'indirizzo di segmento a cui mappare la pagina logica specificata nella prima. Il limite evidente è che il mapping multiplo di pagine logiche EMS su memoria convenzionale può essere effetuato solo ad indirizzi il cui offset (implicito) è `0`.

A completamento di queste note sulla memoria espansa e la sua gestione in C vale la pena di elencare i codici restituiti in `AH` dall'int 67h in uscita dai diversi servizi che esso implementa. La tabella contiene solamente i codici relativi ai servizi descritti.

CODICI DI ERRORE EMS

AH	SIGNIFICATO DEL CODICE
`00h`	Operazione eseguita correttamente
`80h`	Errore interno
`81h`	Errore hardware
`82h`	EMM già installato
`83h`	Handle non valido
`84h`	Funzione non supportata
`85h`	Nessuno handle disponibile (allocazione fallita)
`86h`	Errore nella gestione del mapping
`87h`	Pagine EMS insufficienti (allocazione fallita)
`88h`	Pagine EMS libere insufficienti (allocazione fallita)
`89h`	Richiesta l'allocazione di `0` pagine (allocazione fallita)
`8Ah`	Numero di pagina logica fuori dall'intervallo valido per lo handle
`8Bh`	Numero di pagina fisica fuori dall'intervallo valido (0-3)
`8Ch`	Memoria insufficiente per gestire il mapping
`8Dh`	Errore nella gestione dei dati di mapping (in fase di salvataggio)
`8Eh`	Errore nella gestione dei dati di mapping (in fase di ripristino)

`8Fh`	Subfunzione non supportata
`92h`	Trasferimento avvenuto; parte della regione sorgente è stata sovrascritta
`93h`	E' stato richiesto il trasferimento di un'area di dimensioni maggiori della memoria allocata
`94h`	L'area di memoria convenzionale e l'area di memoria espansa si sovrappongono
`95h`	L'offset indicato è fuori dall'intervallo valido
`96h`	La lunghezza indicata è maggiore di un Mb
`97h`	Trasferimento non avvenuto: le aree sorgente e destinazione si sovrappongono
`98h`	Il tipo di memoria specificato non è valido (deve essere 0 o 1)
`A2h`	L'indirizzo di memoria iniziale sommato alla lunghezza supera il Mb
`A3h`	Puntatore non valido o contenuto dell'array sorgente corrotto

MEMORIA ESTESA, HIGH MEMORY AREA E UMB

E' detta memoria estesa la RAM oltre il primo megabyte: essa è indirizzabile linearmente dai programmi solo se operanti in modalità protetta[227]; in modalità reale l'accesso alla memoria estesa è possibile, grazie a un driver XMM (eXpanded Memory Manager) in grado di fornire i servizi XMS[228]. I primi 64 Kb (meno 16 byte) di memoria estesa costituiscono la High Memory Area (HMA), che occupa gli indirizzi `FFFF:0010-FFFF:FFFF`[229]; questa è indirizzabile da una coppia di registri a 16 bit, quando la A20 Line è abilitata[230].

---

[227] I programmi DOS possono operare in modo protetto solo su macchine dotate di processore 80286 o superiore. La modalità standard di lavoro in ambiente DOS, possibile su tutte le macchine, è la cosiddetta reale (real mode).

[228] XMS è acronimo di eXtended Memory Specification, standard industriale per la gestione della memoria estesa. La XMS include anche regole per la gestione degli UMB.

[229] Ne segue che la memoria estesa in senso stretto si trova oltre i primi 1088 Kb (`FFFF:FFFF`).

[230] Forse è il caso di spendere due parole di chiarimento circa la HMA. Questa è l'unica parte di memoria estesa indirizzabile da un processore 80286 o superiore senza necessità di lavorare in modalità protetta. Infatti l'indirizzo `F000:FFFF` punta all'ultimo byte di memoria entro il primo Mb: detto indirizzo è normalizzato in `FFFF:000F`. Incrementandolo di uno si ottiene `FFFF:0010`, cioè l'indirizzo del primo byte di memoria estesa, equivalente all'indirizzo lineare `100000h`, esprimibile mediante 21 bit. Le macchine 8086 dispongono di sole 20 linee di indirizzamento della RAM e possono quindi gestire indirizzi "contenuti" in 20 bit. Per questo motivo l'indirizzo `FFFF:0010` subisce su di esse il cosiddetto *address wrapping* e diviene `0000:0000`. Al contrario, le macchine 80286 dispongono di 24 bit per l'indirizzamento della memoria; quelle basate sul chip 80386 ne hanno 32. La A20 line è la linea hardware (sono convenzionalmente indicate con A0...A31) di indirizzamento della RAM corrispondente al ventunesimo bit: essa consente, se attiva (i servizi XMS lo consentono anche in modalità reale), di indirizzare i primi `FFF0h` byte (`65520`) al di là del primo Mb, che rappresentano, appunto, la HMA. Per ulteriori notizie circa l'indirizzamento della RAM vedere pag. 16.

I servizi XMS per la memoria estesa

Per utilizzare i servizi XMS occorre innanzitutto determinare, mediante l'int 2Fh, se un driver XMM è installato e, in caso affermativo, individuare l'indirizzo (entry point) della routine di servizio driver.

INT 2FH: PRESENZA DEL DRIVER XMM

Input	AX	4300h
Output	AL	80h se il driver è installato.
Note		Non si tratta di un servizio XMS.

INT 2FH: ENTRY POINT DEL DRIVER XMM

Input	AX	4310h
Output	ES:BX	Indirizzo (seg:off) del driver XMM.
Note		L'indirizzo restituito in ES:BX è l'entry point del driver, cioè l'indirizzo al quale il programma che richiede un servizio XMS deve trasferire il controllo.

Ecco un esempio di utilizzo delle chiamate descritte.

```
/********************

    BARNINGA_Z! - 1992

    XMMADDR.C - getXMMaddress()

    void (far *cdecl getXMMaddress(void))(void);
    Restituisce: l'entry point del driver XMM. Se non è installato alcun
                 driver XMM restituisce NULL.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmmaddr.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-

#include <stdio.h>                                                    /* per NULL */

void (far *cdecl getXMMaddress(void))(void)
{
    _AX = 0x4300;
    asm int 2Fh;
    if(_AL != 0x80)
        return((void(far *)(void))NULL);
    _AX = 0x4310;
```

```
    asm int 2Fh;
    return((void _es *)_BX);
}
```

La `getXMMaddress()` restituisce un puntatore a funzione `void`, che rappresenta l'entry point del driver XMM.

A differenza dei servizi EMM, gestiti tramite un interrupt (l'int 67h), quelli resi disponibili dal driver XMM sono accessibili invocando direttamente la routine che si trova all'entry point del driver, dopo avere opportunamente caricato i registri. In termini di assembler si tratta di eseguire una `CALL`; chi preferisce il C può utilizzare l'indirezione del puntatore all'entry point stesso. Alcuni esempi serviranno a chiarire le idee.

SERVIZIO XMS 00H: VERSIONE DEL DRIVER XMM E ESISTENZA DELLA HMA

Input	AH	00h
Output	AH	Versione XMS supportata.
	AL	Revisione XMS supportata.
	BH	Versione del driver XMM.
	BL	Revisione del driver XMM.
	DX	1 se esiste HMA, 0 altrimenti
Note		Se in uscita AX = 0, si è verificato un errore. In tal caso BL contiene il numero di errore XMS (vedere pag. 248).

```
/********************

    BARNINGA_Z! - 1992

    XMMVERS.C - getXMMversion()

    int cdecl getXMMversion(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver.
    Restituisce: un intero il cui byte meno significativo rappresenta la versione
                 XMS supportata e quello piu' significativo la revisione;
                 Se < 0 e' il codice di errore XMS cambiato di segno.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmmvers.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl getXMMversion(void (far *XMMdriver)(void))
{
    _AH = 0;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
```

```
    asm neg bx;
    return(_BX);
EXITFUNC:
    asm xchg ah,al;
    return(_AX);
}
```

La `getXMMversion()` restituisce un unsigned integer, il cui byte meno significativo rappresenta la versione, e quello più significativo la revisione[231] della specifica XMS supportata. Se il valore restituito è minore di `0` si è verificato un errore: detto valore è il codice di errore XMS cambiato di segno. L'indirezione del puntatore all'entry point del driver trasferisce ad esso il controllo: dal momento che il parametro `XMMdriver` rappresenta proprio quell'indirizzo, l'istruzione C

```
    (*XMMdriver)();
```

è equivalente allo inline assembly

```
    asm call dword ptr XMMdriver;
```

Circa i puntatori a funzione vedere pag. 93.

```
/********************

    BARNINGA_Z! - 1992

    XMMDVERS.C - getXMMdrvVersion()

    int cdecl getXMMdrvVersion(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver.
    Restituisce: un intero il cui byte meno significativo rappresenta la versione
                 del driver XMM e quello piu' significativo la revisione;
                 Se < 0 e' il codice di errore XMS cambiato di segno.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmmdvers.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl getXMMdrvVersion(void (far *XMMdriver)(void))
{
    _AH = 0;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
EXITFUNC:
    asm xchg bh,bl;
    return(_BX);
}
```

[231] Il valore restituito dall'int 67h è "risistemato" in modo coerente con il servizio 30h dell'int 21h, che restituisce in `AL` la versione e in `AH` la revisione del DOS.

Anche la `getXMMdrvVersion()` restituisce un `unsigned int`, il cui byte meno significativo rappresenta la versione, e quello più significativo la revisione[232] del gestore XMM. Se il valore restituito è minore di `0` si è verificato un errore: detto valore è il codice di errore XMS cambiato di segno.

Il servizio XMS 08h consente di conoscere la quantità di memoria XMS libera.

SERVIZIO XMS 08H: QUANTITÀ DI MEMORIA XMS DISPONIBILE

Input	AH	08h
Output	AX	La dimensione in Kb del maggiore blocco libero.
	DX	La quantità totale, in Kb, di memoria XMS libera.
Note		In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248). La quantità di memoria XMS libera non include mai la HMA, neppure nel caso in cui questa non sia allocata.

```
/********************

    BARNINGA_Z! - 1992

    XMSFREEM.C - getXMSfreeMem()

    long getXMSfreeMem(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    Restituisce: La quantità di memoria XMS libera, in Kilobytes.
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmsfreem.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

long cdecl getXMSfreeMem(void (far *XMMdriver)(void))
{
    _AH = 8;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    asm mov dx,bx;
EXITFUNC:
    return((long)_DX);
}
```

[232] Il valore restituito dall'int 67h è "risistemato" in modo coerente con il servizio 30h dell'int 21h, che restituisce in AL la versione e in AH la revisione del DOS.

```
/********************

    BARNINGA_Z! - 1992

    XMSFREEB.C - getXMSfreeBlock()

    long getXMSfreeBlock(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    Restituisce: la dimensione in Kb del maggiore blocco XMS disponibile;
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmsfreeb.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

long cdecl getXMSfreeBlock(void (far *XMMdriver)(void))
{
    _AH = 8;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    asm mov ax,bx;
EXITFUNC:
    return((long)_AX);
}
```

Non esiste un servizio XMS per determinare la quantità totale di memoria estesa installata sulla macchina; essa è sempre almeno pari ai Kb liberi ai quali vanno sommati, se la HMA esiste, altri 64 Kb. La dimensione della memoria XMS può essere maggiore di quella fisicamente presente sulla macchina nel caso in cui sia attivo un ambiente in grado di creare memoria virtuale (vedere pag.).

La quantità di memoria estesa fisica (la memoria realmente installata sulla macchina) è la word memorizzata nel CMOS[233] ad offset 17h: ecco un suggerimento per conoscerla.

```
/********************

    BARNINGA_Z! - 1992

    EXTINST.C - getEXTinstalled()

    unsigned getEXTinstalled(void);
    Restituisce: il numero di Kb di memoria estesa fisica installati.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- extinst.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

unsigned cdecl getEXTinstalled(void)
```

[233] Del CMOS si parla, con maggiore dettaglio, a pagina .

```
{
    asm {
        mov al,17h;
        out 70h,al;                         // prepara lettura CMOS offset 17h
        jmp $+2;                                          // introduce ritardo
        in al,71h;                           // legge CMOS, byte ad offset 17h
        mov ah,al;                                         // salva byte letto
        mov al,18h;                  // ripete procedura per byte ad offset 18h
        out 70h,al;
        jmp $+2;
        in al,71h;
        xchg ah,al;                        // ricostruisce la word (backwords)
    }
    return(_AX);
}
```

La quantità di memoria estesa installata e non gestita da un driver XMM è conoscibile attraverso l'int 15h[234].

INT 15H, SERV. 88H: MEMORIA ESTESA (NON XMS) DISPONIBILE

Input	AH	88h
Output	AX	La quantità in Kb di memoria estesa installata non gestita da un driver XMM.
Note		In caso di errore `CarryFlag` = `1`; il valore in `AX` non è significativo.

```
/********************

    BARNINGA_Z! - 1992

    EXTFREE.C - getEXTfree()

    unsigned getEXTfree(void);
    Restituisce: il numero di Kb di memoria estesa installati e attualmente
                 non sotto il controllo di un driver XMM.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- extfree.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
```

[234] L'uso dell'int 15h rappresenta uno standard (precedente al rilascio delle specifiche XMS) in base al quale il programma che intende allocare memoria estesa chiama l'int 15h e memorizza la quantità di memoria estesa esistente: tale valore rappresenta anche l'indirizzo dell'ultimo byte di memoria fisicamente installata. Il programma stesso deve poi installare un proprio gestore dell'int 15h, che restituisce al successivo chiamante un valore inferiore: la differenza rappresenta proprio la quantità di memoria estesa che il programma intende riservare a sé. Un altro standard, anch'esso precedente a quello XMS, è il cosiddetto sistema "VDISK", dal nome del driver Microsoft per ram-disk che lo implementò per primo. L'algoritmo è analogo a quello dell'int 15h, ma la memoria estesa è allocata al programma a partire "dal basso", cioè dall'indirizzo `100000h` (con lo standard dell'int 15h la memoria è, evidentemente, allocata a partire dall'alto). I tre sistemi (int 15h, VDISK e XMS) sono beatamente incompatibili tra loro... c'era da dubitarne?

```
#pragma  inline

unsigned cdecl getEXTfree(void)
{
    _AH = 0x88;
    asm int 15h;
    return(_AX);
}
```

Il valore restituito da `getEXTfree()` è sempre minore o uguale di quello restituito da `getEXTinstalled()`, salvo il caso in cui il sistema utilizzi memoria virtuale.

Quando si conosca la quantità di memoria XMS libera, l'operazione preliminare al suo utilizzo è l'allocazione di un Extended Memory Block (EMB), il quale altro non è che un'area di memoria XMS, della dimensione desiderata, alla quale il driver associa uno handle di riferimento: l'analogia con i servizi EMS (pag. 212) è evidente.

SERVIZIO XMS 09H: ALLOCA UN BLOCCO DI MEMORIA XMS

Input	AH	09h
	DX	La dimensione del blocco, in Kilobyte
Output	AX	1 se l'allocazione è riuscita correttamente.
	DX	Lo handle associato al blocco.
Note		In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248).

Vediamo un esempio di funzione basata sul servizio 09h.

```
/********************

    BARNINGA_Z! - 1992

    EMBALLOC.C - allocEMB()

    int allocEMB(void (far *XMMdriver)(void),unsigned EMBkb);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    unsigned EMBkb;                dimensione in Kb del blocco richiesto.
    Restituisce: Lo handle associato al blocco allocato.
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- emballoc.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl allocEMB(void (far *XMMdriver)(void),unsigned EMBkb)
{
    _DX = EMBkb;
    _AH = 9;
    (*XMMdriver)();
    asm cmp ax,0;
```

```
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    asm mov dx,bx;
EXITFUNC:
    return(_DX);
}
```

La `allocEMB()` tenta di allocare un Extended Memory Block della dimensione (in Kb) specificata con il parametro `EMBkb`. Se l'operazione riesce viene restituito lo handle che il driver XMS ha associato al blocco, altrimenti il valore restituito, cambiato di segno, è il codice di errore XMS.

Per copiare dati dalla memoria convenzionale ad un EMB o viceversa è disponibile il servizio XMS 0Bh, che può essere utilizzato anche per copiare dati da un EMB ad un secondo EMB, nonché tra due indirizzi in memoria convenzionale: l'operazione desiderata è infatti descritta al driver tramite un buffer composto di 5 campi, il primo dei quali indica la lunghezza in byte dell'area da copiare; i campi successivi possono essere suddivisi in due coppie (una per l'area sorgente ed una per l'area destinazione), in ciascuna delle quali il significato del secondo campo dipende dal valore contenuto nel primo. In particolare, se il primo campo contiene un valore diverso da `0`, questo è interpretato come handle di un EMB, e dunque il secondo campo della coppia indica l'offset lineare a 32 bit all'interno dell'EMB. Se, al contrario, il primo campo è `0`, allora il secondo è interpretato come un normale indirizzo `far` a 32 bit in memoria convenzionale. Valorizzando opportunamente i campi è possibile richiedere operazioni di copia in qualsiasi direzione.

SERVIZIO XMS 0BH: COPIA TRA AREE DI MEMORIA CONVENZIONALE O XMS

Input	`AH`	`0Bh`
	`DS:SI`	Buffer descrittivo dell'operazione (vedere tabella seguente)
Output	`AX`	`1` se l'operazione di copia è riuscita correttamente.
Note		In caso di fallimento, `AX` è `0` e `BL` contiene il codice di errore (vedere pag. 248).

Il formato del buffer richiesto dal servizio è il seguente:

FORMATO DEL BUFFER UTILIZZATO DAL SERVIZIO XMS 0BH

OFFSET	BYTE	DESCRIZIONE
`00h`	4	Lunghezza in byte dell'area di memoria da trasferire (deve essere un numero pari)
`04h`	2	Handle XMS sorgente (`0` se memoria convenzionale)
`06h`	4	Se il campo precedente è `0` questo campo è un puntatore `far` ad un'area di memoria convenzionale; se invece il campo predecente è diverso da 0, questo campo rappresenta un offset lineare a 32 bit nel blocco di memoria estesa associato allo handle. In entrambi i casi l'indirizzo è inteso come sorgente.
`0Ah`	2	Handle XMS destinazione (`0` se memoria convenzionale)
`0Ch`	4	Se il campo precedente è 0 questo campo è un puntatore `far` ad un'area di memoria convenzionale; se invece il campo predecente è diverso da 0, questo campo rappresenta un offset lineare a 32 bit nel blocco di memoria estesa associato allo handle. In entrambi i casi l'indirizzo è inteso come destinazione.

Il buffer può essere rappresentato da una struttura:

```
struct EMBmove {
    unsigned long length;                            // lunghezza in bytes dell'area
    unsigned srcHandle;              // handle EMB sorgente (0 se mem. convenzionale)
    long srcOffset;   // offset nell'EMB sorgente o far pointer a mem. convenzionale
    unsigned dstHandle;             // handle EMB destinaz. (0 se mem. convenzionale)
    long dstOffset;  // offset nell'EMB destinaz. o far pointer a mem. convenzionale
};
```

La struttura `EMBmove` è il secondo parametro della funzione `XMSmoveMem()`:

```
/********************

    BARNINGA_Z! - 1992

    XMSMOVM.C - XMSmoveMem()

    int XMSmoveMem(void (far *XMMdriver)(void),struct EMBmove *EMBbuffer);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    struct EMBmove *EMBbuffer;     puntatore al buffer che descrive l'operazione.
    Restituisce: 0 se l'operazione e' stata eseguita correttamente.
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmsmovm.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
```

```
int cdecl XMSmoveMem(void (far *XMMdriver)(void),struct EMBmove *EMBbuffer)
{
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm mov si,EMBbuffer;
#else
    asm push ds;
    asm lds si,dword ptr EMBbuffer;
#endif
    _AH = 0x0B;
    (*XMMdriver)();
#if defined(__COMAPCT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    asm cmp ax,0;
    asm je EXITFUNC;
    return(0);
EXITFUNC:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
}
```

L'esempio che segue utilizza la `XMSmoveMem()` per copiare 96 Kb da memoria convenzionale a memoria estesa. I dati sono copiati ad offset `10000h` nell'EMB.

```
    ....
    unsigned char far *convMemPtr;
    void (far *XMMdriver)(void);         // puntatore all'entry point del driver XMM
    struct EMBmove EMBbuffer;
    unsigned EMBhandle;
    ....
    EMBbuffer.length = 96 * 1024;          // lunghezza in bytes dell'area da copiare
    EMBbuffer.srcHandle = 0;                         // sorgente memoria convenzionale
    EMBbuffer.srcOffset = (long)convMemPtr;      // indirizzo sorgente in mem. conv.
    EMBbuffer.dstHandle = EMBhandle;                       // restituito da allocEMB()
    EMBbuffer.dstOffset = 0x10000;          // offset nell'EMB associato a EMBhandle
    if(XMSmoveMem(XMMdriver,&EMBbuffer) < 0)
        ....                                                     // gestione errore XMS
```

Si tenga presente che il servizio XMS 0Bh non garantisce che la copia sia effettuata correttamente se l'area sorgente e l'area destinazione si sovrappongono anche parzialmente e l'indirizzo della prima è maggiore dell'indirizzo della seconda. Inoltre non è necessario abilitare la A20 line per utilizzare il servizio.

Prima di terminare i programmi devono esplicitamente liberare gli EMB eventualmente allocati e la HMA se utilizzata, al fine di renderli nuovamente disponibili agli altri processi. Il DOS non interagisce con il driver XMM nella gestione della memoria estesa, pertanto in uscita dai programmi non effettua alcuna operazione di cleanup riguardante gli EMB e la HMA: se non rilasciati dal processo che termina, questi risultano inutilizzabili da qualsiasi altro programma sino al reset della macchina (la situazione è analoga a quella descritta con riferimento ai servizi EMS: vedere pag. 215).

SERVIZIO XMS 0AH: DEALLOCA UNO HANDLE XMS

Input	AH DX	0Ah Handle XMS da deallocare
Output	AX	1 se la disallocazione è riuscita correttamente. Tutta la memoria estesa a cui è associato lo handle è liberata.
Note		In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248).

Segue, come sempre, un esempio di funzione basata sul servizio appena descritto.

```
/********************

    BARNINGA_Z! - 1992

    EMBFREE.C - freeEMB()

    int freeEMB(void (far *XMMdriver)(void),unsigned EMBhandle);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    unsigned EMBhandle;            handle EMB da deallocare.
    Restituisce: 0 se l'operazione è stata eseguita con successo.
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- embfree.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl freeEMB(void (far *XMMdriver)(void),unsigned EMBhandle)
{
    _DX = EMBhandle;
    _AH = 0x0A;
    (*XMMdriver)();
    asm cmp ax,0;
    asm je EXITFUNC;
    return(0);
EXITFUNC:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
}
```

Anche in questo caso è restituito 0 se l'operazione è stata eseguita correttamente, mentre un valore minore di 0 segnala che si è verificato un errore XMS e ne rappresenta, al tempo stesso, il codice cambiato di segno.

Vale ancora la pena di aggiungere che è possibile richiedere la modifica della dimensione di un EMB mediante il servizio XMS 0Fh:

SERVIZIO XMS 0FH: MODIFICA LA DIMENSIONE DI UN EMB

Input	AH	0Fh
	BX	Nuova dimensione richiesta in Kilobyte
	DX	Handle XMS associato all'EMB
Output	AX	1 se la disallocazione è riuscita correttamente. Tutta la memoria estesa a cui è associato lo handle è liberata.
Note		In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248).

Segue esempio:

```
/********************

    BARNINGA_Z! - 1992

    EMBRESIZ.C - resizeEMB()

    int resizeEMB(void (far *XMMdriver)(void),unsigned EMBhandle,unsigned newKb);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    unsigned EMBhandle;            handle XMS associato all'EMB da modificare.
    unsigned newKb;                nuova dimensione desiderata, in Kilobytes.
    Restituisce: 0 se l'operazione è stata eseguita con successo.
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- embresiz.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl resizeEMB(void (far *XMMdriver)(void),unsigned EMBhandle,unsigned newKb)
{
    _DX = EMBhandle;
    _BX = newKb;
    _AH = 0x0F;
    (*XMMdriver)();
    asm cmp ax,0;
    asm je EXITFUNC;
    return(0);
EXITFUNC:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
}
```

La `resizeEMB()` restituisce un valore secondo criteri coerenti con quelli adottati dalle funzioni sopra listate[235].

[235] Non è davvero il caso di ripetere sempre la medesima litania.

I servizi XMS per la HMA

Una parte dei servizi XMS è implementa la gestione della High Memory Area. Vediamo i listati di alcune funzioni.

```
/********************

    BARNINGA_Z! - 1992

    XMMISHMA.C - isHMA()

    int cdecl isHMA(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver.
    Restituisce: 1  HMA esistente
                 0  HMA non esistente
                 Se < 0 e' il codice di errore XMS cambiato di segno.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- xmmishma.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl isHMA(void (far *XMMdriver)(void))
{
    _AH = 0;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
EXITFUNC:
    return(_DX);
}
```

La `isHMA()` restituisce `1` se la HMA è presente, `0` in caso contrario. Un valore minore di `0` rappresenta il codice di errore XMS.

In caso di esistenza della HMA è possibile determinare se essa è allocata o libera tentando di allocarla: se l'operazione ha successo la HMA era disponibile (occorre allora disallocarla per ripristinare la situazione iniziale).

SERVIZIO XMS 01H: ALLOCA LA HMA (SE DISPONIBILE)

Input	AH DX	01h Byte necessari al programma nella HMA (valori possibili 0-FFF0).
Output	AX BL	1 se la HMA è stata allocata con successo. In caso di fallimento, AX è 0 e BL contiene il codice di errore: se BL = 91h, la HMA è già allocata (vedere pag. 248).
Note		Il numero di byte specificato in DX viene confrontato con il valore attribuito al parametro /HMAMIN sulla riga di comando del driver XMM: se è inferiore a quest'ultimo la HMA non viene allocata neppure se libera, in caso contrario è allocata tutta la HMA.

Circa il servizio XMS 01h, va precisato che, secondo la documentazione ufficiale, non è possibile dividere la HMA in subaree: essa è allocata interamente, anche se sono richiesti meno di 64Kb. Esiste però un servizio non documentato[236] dell'int 2Fh, supportato dal driver HIMEM.SYS del DOS, che dispone di due subfunzioni utili per allocare la parte libera di HMA quando questa sia già allocata:

INT 2FH, SERV. 4AH: GESTISCE LA PORZIONE LIBERA DI UNA HMA GIÀ ALLOCATA

Input	AH AL	4Ah 01h richiede quanti byte sono ancora liberi nella HMA 02h alloca spazio nella HMA BX numero di byte richiesti
Output	ES:DI BX	Indirizzo del primo byte libero nella HMA (FFFF:FFFF in caso di errore). Numero di byte liberi in HMA (solo subfunzione 01h).
Note		Questo servizio è disponibile solo se il DOS è caricato in HMA: in caso contrario viene restituito 0 in BX e FFFF:FFFF in ES:DI.

[236] E, pertanto, pericoloso. Non sembrano inoltre essere disponibili servizi per la disallocazione delle porzioni di HMA allocate mediante la subfunzione 02h del servizio 4Ah.

SERVIZIO XMS 02H: DEALLOCA LA HMA

Input	AH	02h
Output	AX BL	1 se la HMA è stata liberata con successo. In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248). codice di errore
Note		Il programma che alloca la HMA deve disallocarla prima di restituire il controllo al sistema; in caso contrario la HMA rimane inutilizzabile fino al primo bootstrap.

```
/********************

    BARNINGA_Z! - 1992

    ISHMAFRE.C - isHMAfree()

    int isHMAfree(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    Restituisce: 1 se la HMA e' libera;
                 0 se gia' allocata;
                 Se < 0 e' il codice di errore XMS cambiato di segno.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- ishmafre.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl isHMAfree(void (far *XMMdriver)(void))
{
    _AH = 1;
    _DX = 0xFFF0;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne DEALLOC;
    asm cmp bl,091h;
    asm jne ERROR;
    return(0);                                                  // HMA gia' allocata
ERROR:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
DEALLOC:
    _AH = 2;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm jmp ERROR;
EXITFUNC:
    return(1);
}
```

La `isHMAfree()` restituisce `0` se la HMA è già allocata o, al contrario, `1` se è libera. In caso di errore è restituito un valore negativo che rappresenta, cambiato di segno, il codice di errore XMS.

E' facile, "spezzando" la `isHMAfree()`, realizzare due funzioni in grado di allocare la HMA (`HMAalloc()`) e, rispettivamente, disallocarla (`HMAdealloc()`).

```
/********************

    BARNINGA_Z! - 1992

    HMAALLOC.C - HMAalloc()

    int *HMAalloc(void (far *XMMdriver)(void),unsigned nBytes);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    unsigned nBytes;               numero di bytes da allocare.
    Restituisce: 0 se l'allocazione e' stata effettuata con successo;
                 Se < 0 e' il codice di errore XMS

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- hmaalloc.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl HMAalloc(void (far *XMMdriver)(void),unsigned nBytes)
{
    _DX = nBytes;
    _AH = 1;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    return(_BX);                                      // HMA gia' allocata o errore
EXITFUNC:
    return(0);
}


/********************

    BARNINGA_Z! - 1992

    HMADEALL.C - HMAdealloc()

    int HMAdealloc(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    Restituisce: 0 se la HMA e' stata deallocata con successo;
                 Se < 0 e' il codice di errore XMS cambiato di segno.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- hmadeall.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl HMAdealloc(void (far *XMMdriver)(void))
```

```
{
    _AH = 2;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
EXITFUNC:
    return(0);
}
```

Lo stato della A20 Line, attraverso la quale è possibile indirizzare la HMA, può essere indagato tramite il servizio 07h del driver XMM.

SERVIZIO XMS 07H: STATO DELLA A20 LINE

Input	AH	07h
Output	AX BL	1 se la A20 Line è abilitata, 0 se non lo è (in questo caso anche BL = 0). In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248). codice di errore

```
/********************

    BARNINGA_Z! - 1992

    ISA20ON.C - isA20enabled()

    int cdecl isA20enabled(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);     puntatore all'entry point del driver XMM.
    Restituisce: 1 se la A20 line e' abilitata;
                 0 se la A20 line e' disabilitata;
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- isa20on.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl isA20enabled(void (far *XMMdriver)(void))
{
    _AH = 7;
    (*XMMdriver)();
    asm cmp ax,0;
    asm jne EXITFUNC;
    asm cmp bl,0;
    asm je EXITFUNC;
    asm xor bh,bh;
    asm neg bx;
    asm mov ax,bx;
EXITFUNC:
    return(_AX);
}
```

La `isA20enabled()` restituisce `1` se la A20 line è abilitata, `0` se non lo è. In caso di errore è restituito un valore negativo che, cambiato di segno rappresenta il codice di errore XMS.

Per utilizzare la HMA occorre che la linea A20 sia abilitata; i servizi XMS 03h e 04h la abilitano e disabilitano specificamente per consentire l'accesso alla HMA[237].

SERVIZIO XMS 03H: ABILITA LA A20 LINE

Input	AH	03h
Output	AX	1 se la A20 Line è stata attivata correttamente.
Note		In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248). Questo servizio deve essere utilizzato solo se il programma ha allocato con successo la HMA mediante il servizio 01h.

SERVIZIO XMS 04H: DISABILITA LA A20 LINE

Input	AH	04h
Output	AX	1 se la A20 Line è stata disattivata correttamente.
Note		In caso di fallimento, AX è 0 e BL contiene il codice di errore (vedere pag. 248). Questo servizio deve essere utilizzato solo se il programma ha allocato con successo la HMA mediante il servizio 01h.

I due esempi di funzione che seguono utilizzano i servizi testè descritti.

```
/********************

    BARNINGA_Z! - 1992
```

[237] Quando la A20 line è attiva, una coppia di registri a 16 bit può esprimere indirizzi lineari a 21 bit, fino a `FFFF:FFFF` (non è più una novità). Ne segue che è possibile referenziare direttamente indirizzi all'interno della HMA tramite normali puntatori `far` o `huge` (pag. 21). Ad esempio l'istruzione

```
#include <dos.h>                                                    // per MK_FP()
....
    register i;
    unsigned char *cPtr;

    for(i = 0x10; i < 0x20; i++)
        *(unsigned char far *)MK_FP(0xFFFF,i) = *cPtr++;
```

copia 16 byte da un indirizzo in memoria convenzionale (referenziato da un puntatore `near`) all'inizio della HMA. I servizi XMS 05h e 06h, del tutto analoghi ai servizi 03h e 04h, consentono di attivare e, rispettivamente, disattivare la A20 line per indirizzare direttamente, mediante puntatori lineari a 32 bit, circa 1 Mb di memoria estesa (al di fuori della HMA).

```
    A20ENABL.C - enableA20()

    int cdecl enableA20(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);     puntatore all'entry point del driver XMM.
    Restituisce: 0 se la A20 line e' stata abilitata con successo;
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- a20enabl.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl enableA20(void (far *XMMdriver)(void))
{
    _AH = 3;
    (*XMMdriver)();
    asm cmp ax,0;
    asm je EXITFUNC;
    return(0);
EXITFUNC:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
}


/********************

    BARNINGA_Z! - 1992

    A20DISAB.C - disableA20()

    int cdecl disableA20(void (far *XMMdriver)(void));
    void (far *XMMdriver)(void);     puntatore all'entry point del driver XMM.
    Restituisce: 0 se la A20 line e' stata disabilitata con successo;
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- a20disab.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl disableA20(void (far *XMMdriver)(void))
{
    _AH = 4;
    (*XMMdriver)();
    asm cmp ax,0;
    asm je EXITFUNC;
    return(0);
EXITFUNC:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
}
```

Le funzioni `enableA20()` e `disableA20()` restituiscono `0` se hanno eseguito correttamente il loro compito; in caso di errore è restituito un valore negativo che rappresenta, cambiato di segno, il codice di errore XMS.

I servizi XMS per gli UMB

Riportiamo la descrizione dei servizi XMS (e gli esempi relativi) che consentono di allocare e deallocare gli Upper Memory Block. Per ulteriori e più approfonditi particolari circa gli UMB vedere pag. 198 e seguenti.

SERVIZIO XMS 0FH: ALLOCA UN UMB

Input	AH	`10h`
	DX	Dimensione richiesta in paragrafi (blocchi di 16 byte)
Output	AX	`1` se la disallocazione è riuscita correttamente.
	BX	Indirizzo di segmento dell'UMB allocato
	DX	Dimensione reale in paragrafi
Note		In caso di fallimento, `AX` è `0` e `BL` contiene il codice di errore (vedere pag. 248), mentre `DX` contiene la dimensione in paragrafi del massimo UMB disponibile.

Segue esempio:

```
/********************

    BARNINGA_Z! - 1992

    UMBALLOC.C - allocUMB()

    int allocUMB(void (far *XMMdriver)(void),unsigned UMBKb,unsigned *UMBseg);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    unsigned UMBKb;                dimensione desiderata, in paragrafi.
    unsigned *UMBseg;              usata per restituire l'indirizzo di segmento
                                   dell'UMB allocato. In caso di errore contiene la
                                   dimensione del massimo UMB disponibile.
    Restituisce: 0 se l'operazione e' stata eseguita con successo. All'indirizzo
                 UMBseg e' memorizzato l'indirizzo di segmento dell'UMB allocato;
                 Se < 0 e' il codice di errore XMS; All'indirizzo UMBseg e'
                 memorizzata la dimensione del massimo UMB disponibile.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- umballoc.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

int cdecl allocUMB(void (far *XMMdriver)(void),unsigned UMBkb,unsigned *UMBseg)
```

```
{
    _DX = UMBkb;
    _AH = 0x10;
    (*XMMdriver)();
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    asm mov si,UMBseg;
#else
    asm push ds;
    asm lds si,dword ptr UMBseg;
#endif
    asm cmp ax,0;
    asm je EXITFUNC;
    asm mov [si],bx;
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    return(0);
EXITFUNC:
    asm mov [si],dx;
    asm xor bh,bh;
    asm neg bx;
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    return(_BX);
}
```

SERVIZIO XMS 11H: DEALLOCA UN UN UMB

Input	AH DX	`11h` Indirizzo di segmento dell'UMB
Output	AX	`1` se la disallocazione è riuscita correttamente. L'UMB è nuovamente libero
Note		In caso di fallimento, `AX` è `0` e `BL` contiene il codice di errore (vedere pag. 248).

Segue esempio:

```
/********************

    BARNINGA_Z! - 1992

    UMBFREE.C - freeUMB()

    int freeUMB(void (far *XMMdriver)(void),unsigned UMBseg);
    void (far *XMMdriver)(void);   puntatore all'entry point del driver XMM.
    unsigned UMBseg;               indirizzo di segmento dell'UMB da liberare.
    Restituisce: 0 se l'operazione è stata eseguita con successo.
                 Se < 0 e' il codice di errore XMS.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -mx -k- umbfree.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh
```

```
********************/
#pragma  inline

int cdecl freeUMB(void (far *XMMdriver)(void),unsigned UMBseg)
{
    _DX = UMBseg;
    _AH = 0x11;
    (*XMMdriver)();
    asm cmp ax,0;
    asm je EXITFUNC;
    return(0);
EXITFUNC:
    asm xor bh,bh;
    asm neg bx;
    return(_BX);
}
```

Un UMB allocato con `allocUMB()` può essere referenziato mediante un puntatore `far` o `huge`, costruito con la macro `MK_FP()` (pag. 24):

```
#include <dos.h>                                                      // per MK_FP()
    ....
    unsigned umbSeg;
    char far *umbPtr;
    ....
    if(allocUMB(XMMdriver,10000,&umbSeg) < 0) {
        ....                                                  // gestione errore XMS
    }
    else {
        umbPtr = (char far *)MK_FP(umbSeg,0);
        ....                                                         // utilizzo UMB
        if(freeUMB(umbSeg) < 0) {
            ....                                              // gestione errore XMS
        }
    }
```

La tabella che segue riporta i codici di errore XMS.

CODICI DI ERRORE XMS

CODICE	DESCRIZIONE
80h	Funzione non valida
81h	E' installato VDISK
82h	Errore nella A20 line
8Eh	Errore interno al driver
8Fh	Errore irrecuperabile del driver
90h	La HMA non esiste
91h	La HMA è già in uso

`92h`	Richiesta allocazione in HMA di un numero di byte minore del parametro associato a `/HMAMIN` sulla riga di comando del driver XMM
`93h`	La HMA è libera
`94h`	Non è stato possibile disattivare la A20 line
`A0h`	Tutta la memoria estesa è già allocata
`A1h`	Tutti gli handles disponibili per la memoria estesa sono già utilizzati
`A2h`	Handle non valido
`A3h`	Handle sorgente non valido
`A4h`	Offset sorgente non valido
`A5h`	Handle destinazione non valido
`A6h`	Offset destinazione non valido
`A7h`	La lunghezza del blocco non è valida
`A8h`	Le aree sorgente e destinazione si sovrappongono
`A9h`	Errore di parità
`AAh`	Il blocco non è locked
`ABh`	Il blocco è locked
`ACh`	Overflow nel conteggio dei lock del blocco
`ADh`	Operazione di lock fallita
`B0h`	E' disponibile un UMB di dimensione minore a quella richiesta
`B1h`	Non sono disponibili UMB
`B2h`	Segmento UMB non valido

GLI INTERRUPT: GESTIONE

A pag. 115 e seguenti abbiamo visto come un programma C può sfruttare gli interrupt, richiamandoli attraverso le funzioni di libreria dedicate allo scopo. Ora si tratta di entrare nel difficile, cioè raccogliere le idee su come scrivere interrupt, o meglio funzioni C in grado di sostituirsi o affiancarsi alle routine DOS e al BIOS nello svolgimento dei loro compiti.

Si tratta di una tecnica indispensabile a tutti i programmi TSR (Terminate and Stay Resident, vedere pag.) e sicuramente utile ai programmi che debbano gestire il sistema in profondità (ad es.: ridefinire la tastiera, schedulare operazioni tramite il timer della macchina, etc.).

LA TAVOLA DEI VETTORI

Gli indirizzi (o vettori) degli interrupt si trovano nei primi 1024 byte della RAM; vi sono 256 vettori, pertanto ogni indirizzo occupa 4 byte: dal punto di vista del C si tratta di puntatori `far`, o meglio, di puntatori a funzioni `interrupt`, per quei compilatori che definiscono tale tipo di dato[238].

Per conoscere l'indirizzo di un interrupt si può utilizzare la funzione di libreria `getvect()`, che accetta, quale parametro, il numero dell'interrupt[239] stesso e ne restituisce l'indirizzo. La sua "controparte" `setvect()` consente di modificare l'indirizzo di un interrupt, prendendo quali parametri il numero dell'interrupt ed il suo nuovo indirizzo (un puntatore a funzione `interrupt`). Un'azione combinata `getvect()`/`setvect()` consente dunque di memorizzare l'indirizzo originale di un interrupt e sostituirgli, nella tavola dei vettori, quello di una funzione user-defined. L'effetto è che gli eventi hardware o software che determinano l'attivazione di quell'interrupt chiamano invece la nostra funzione. Una successiva chiamata a `setvect()` consente di ripristinare l'indirizzo originale nella tavola dei vettori quando si intenda "rimettere le cose a posto".

Fin qui nulla di particolarmente complicato; se l'utilità di `getvect()` e `setvect()` appare tuttavia poco evidente non c'è da preoccuparsi: tutto si chiarirà strada facendo. Uno sguardo più attento alla suddetta tavola consente di notare che, se ogni vettore occupa 4 byte, l'offset di un dato vettore rispetto all'inizio della tavola è ottenibile moltiplicando per quattro il numero del vettore stesso (cioè dell'interrupt corrispondente): ad esempio, il vettore dell'int 09h si trova ad offset `36` rispetto all'inizio della tavola (il primo byte della tavola ha, ovviamente, offset `0`). Dal momento che la tavola si trova all'inizio della RAM, l'indirizzo del vettore dell'int 09h è `0000:0024` (`24h = 36`). Attenzione: l'indirizzo di un vettore non è l'indirizzo dell'interrupt, bensì l'indirizzo del puntatore all'interrupt. Inoltre, i primi due byte (nell'esempio quelli ad offset `36` e `37`) rappresentano l'offset del vettore, i due successivi (offset `38` e `39`) il segmento, coerentemente con la tecnica backwords.

Queste considerazioni suggeriscono un metodo alternativo per la manipolazione della tavola dei vettori. Ad esempio, per ricavare l'indirizzo di un interrupt si può moltiplicarne il numero per quattro e leggere i quattro byte a quell'offset rispetto all'inizio della RAM:

```
int intr_num;
int ptr;
void(interrupt *intr_pointer)();
....
intr_num = 9;
```

[238] Se non è zuppa, è pan bagnato. Sempre di puntatori a 32 bit si tratta. Del resto, un puntatore a 32 bit non è che una tipizzazione particolare di un `long int`.

[239] Ogni interrupt è identificato da un numero, solitamente espresso in notazione esadecimale, da 0 a FFh (255).

```
ptr = intr_num*4;
intr_pointer = (void(interrupt *)())*(long far *)ptr;
....
```

L'integer `ptr`, che vale `36`, è forzato a puntatore `far` a `long`: `far` perché la tavola dei vettori deve essere referenziata con un segmento:offset, in cui segmento è sempre `0`; `long` perché `0:ptr` deve puntare ad un dato a 32 bit. L'indirezione di `(long far *)ptr` è il vettore (in questo caso dell'int 09h); il cast a puntatore ad `interrupt` rende la gestione del dato coerente con i tipi del C. Tale metodo comporta un vantaggio: produce codice compatto ed evita il ricorso a funzioni di libreria (parlando di TSR vedremo che, in quel genere di programmi, il loro utilizzo può essere fonte di problemi: pag.). Manco a dirlo, però, comporta anche uno svantaggio: l'istruzione

```
intr_pointer = (void(interrupt *)())*(long far *)ptr;
```

è risolta dal compilatore in una breve sequenza di istruzioni assembler. L'esecuzione di detta sequenza potrebbe essere interrotta da un interrupt, il quale avrebbe la possibilità di modificare il vettore che il programma sta copiando nella variabile `intr_pointer`, con la conseguenza che segmento e offset del valore copiato potrebbero risultare parti di due indirizzi diversi. Una soluzione del tipo

```
asm cli;
intr_pointer = (void(interrupt *)())*(long far *)ptr;
asm sti;
```

non elimina del tutto il pericolo, perche `cli`[240] non blocca tutti gli interrupt: esiste sempre, seppure molto piccola, la possibilità che qualcuno si intrometta a rompere le uova nel paniere.

In effetti, l'unico metodo documentato da Microsoft per leggere e scrivere la tavola dei vettori è costituito dai servizi 25h e 35h dell'int 21h, sui quali si basano, peraltro, `setvect()` e `getvect()`.

INT 21H, SERV. 25H: SCRIVE UN INDIRIZZO NELLA TAVOLA DEI VETTORI

Input	`AH`	`25h`
	`AL`	numero dell'interrupt
	`DS:DX`	nuovo indirizzo dell'interrupt

INT 21H, SERV. 35H: LEGGE UN INDIRIZZO NELLA TAVOLA DEI VETTORI

Input	`AH`	`35h`
	`AL`	numero dell'interrupt
Output	`ES:BX`	indirizzo dell'interrupt

Una copia della tavola dei vettori può essere facilmente creata così:

```
....
register i;
```

[240] `CLI`, CLear Interrupts, inibisce gli interrupts hardware; `STI`, STart Interrupts, li riabilita (l'assembler non distingue tra maiuscole e minuscole).

```
void(interrupt *inttable[256])();

for(i = 0; i < 256; i++)
    inttable[i] = getvect(i);
....
```

Chi ama il rischio può scegliere un algoritmo più efficiente:

```
....
void(interrupt *inttable[256])();

asm {
    push ds;                                                      // salva DS
    push ds;
    pop es;
    lea di,inttable;                                  // ES:DI punta a inttable
    push 0;
    pop ds;
    xor si,si;                              // DS:SI punta alla tavola dei vettori
    mov cx,512;
    cli;
    rep movsw;                                   // copia 512 words (1024 bytes)
    sti;
    pop ds;
}
....
```

Una copia della tavola dei vettori può sempre far comodo, soprattutto a quei TSR che implementino la capacità di disinstallarsi (cioè rimuovere se stessi dalla RAM) una volta esauriti i loro compiti: al riguardo vedere pag. e seguenti.

Questi dettagli sulla tavola dei vettori sono importanti in quanto un gestore di interrupt è una funzione che non viene mai invocata direttamente, né dal programma che la incorpora, né da altri, bensì entra in azione quando è chiamato l'interrupt da essa gestito. Perché ciò avvenga è necessario che il suo indirizzo sia scritto nella tavola dei vettori, in luogo di quello del gestore originale[241]. Il programma che installa un nuovo gestore di interrupt solitamente si preoccupa di salvare il vettore originale: esso deve essere ripristinato in caso di disinstallazione del gestore, ma spesso è comunque utilizzato dal gestore stesso, quando intenda lasciare alcuni compiti alla routine di interrupt precedentemente attiva.

LE FUNZIONI interrupt

Molti compilatori consentono di dichiarare `interrupt`[242] le funzioni: `interrupt` è un modificatore che forza il compilatore a dotare quelle funzioni di alcune caratteristiche, ritenute importanti per un gestore di interrupt. Vediamo quali.

Il codice C

```
#pragma option -k-

void interrupt int_handler(void)
{
    ....
}
```

---

[241] Con il termine "originale" non si intende indicare il gestore DOS o ROM-BIOS, ma semplicemente quello attivo in quel momento.

[242] Oppure `interrupt far`: mai standardizzare, ovviamente!

definisce una funzione, `int_handler()`, che non prende parametri, non restitusce alcun valore ed è di tipo interrupt. Il compilatore produce il seguente codice assembler[243]:

```
_int_handler proc far
    push ax
    push bx
    push cx
    push dx
    push es
    push ds
    push si
    push di
    push bp
    mov bp,DGROUP
    mov ds,bp
    ....
    pop bp
    pop di
    pop si
    pop ds
    pop es
    pop dx
    pop cx
    pop bx
    pop ax
    iret
_int_handler endp
```

Si nota innanzitutto che la `int_handler()` è considerata `far`: in effetti, i vettori sono indirizzi a 32 bit, pertanto anche quello di ogni gestore di interrupt deve esserlo.

Inoltre la funzione è terminata da una `IRET`: anche questa è una caratteristica fondamentale di tutte le routine di interrupt, in quanto ogni chiamata ad interrupt (sia quelle hardware che quelle software, via istruzione `INT`) salva sullo stack il registro dei flag. L'istruzione `IRET`, a differenza della `RET`, oltre a trasferire il controllo all'indirizzo `CS:IP` spinto dalla chiamata sullo stack, preleva da questo una word (una coppia di byte) e con essa ripristina i flag.

Il registro `DS` è inizializzato a `DGROUP`[244]: l'operazione non è indispensabile, ma consente l'accesso alle variabili globali definite dal programma che incorpora `int_handler()`.

La caratteristica forse più evidente del listato assembler è rappresentata dal salvataggio di tutti i registri sullo stack in testa alla funzione, ed il loro ripristino in coda, prima della `IRET`. Va chiarito che non si tratta di una caratteristica indispensabile ma, piuttosto, di una misura di sicurezza: un interrupt non deve modificare lo stato del sistema, a meno che ciò non rientri nelle sue specifiche finalità[245]. In altre

---

[243] Tutti gli esempi di questo capitolo presuppongono la presenza dell'opzione `-k-` sulla riga di comando del compilatore o l'inserimento della direttiva `#pragma option -k-` nel codice sorgente, onde evitare la generazione della standard stack frame, che richiede le istruzioni `PUSH BP` e `MOV BP,SP` in testa ad ogni funzione (incluse quelle che non prendono parametri e non definiscono variabili locali), nonché `POP BP` in coda, prima della `RET` finale. Nell'esempio di `int_handler()`, l'assenza di detta opzione avrebbe provocato l'inserimento di `MOV BP,SP` dopo la `MOV DS,BP`. Il modello di memoria è lo small, per tutti gli esempi.

[244] Nome convenzionalmente assegnato dal compilatore al segmento allocato ai dati statici e globali (è gestito in realtà come una label).

[245] Consideriamo il caso del già citato in 09h. La pressione di un tasto può avvenire in qualsiasi istante, e non necessariamente in risposta ad una richiesta del programma attivo in quel mentre. Ciò significa che il gestore dell'interrupt non può fare alcuna assunzione a priori sullo stato del sistema e deve evitare di modificarlo inopportunamente, in quanto, a sua volta, il processo interrotto può non avere previsto l'interruzione (e può non essersi neppure "accorto" di essa).

parole, il compilatore genera il codice delle funzioni `interrupt` in modo tale da evitare al programmatore la noia di preoccuparsi dei registri[246], creandogli però qualche problema quando lo scopo del gestore sia proprio modificare il contenuto di uno (o più) di essi. E' palese, infatti, che modificando direttamente il contenuto dei registri non si ottiene il risultato voluto, perché il valore degli stessi in ingresso alla funzione viene comunque ripristinato in uscita. L'ostacolo può essere aggirato dichiarando i registri come parametri formali della funzione: il compilatore, proprio perché si tratta di una funzione di tipo `interrupt`, consente di accedere a quei parametri nello stack gestendo quest'ultimo in modo opportuno: quindi, in definitiva, permette di modificare effettivamente i valori dei registri in uscita alla funzione. Torniamo alla `int_handler()`: se in essa vi fosse, ad esempio, l'istruzione

```
    _BX = 0xABCD;
```

il listato assembler sarebbe il seguente:

```
_int_handler proc far
    push ax
    push bx
    push cx
    push dx
    push es
    push ds
    push si
    push di
    push bp
    mov bp,DGROUP
    mov ds,bp

    mov bx,0ABCDh

    pop bp
    pop di
    pop si
    pop ds
    pop es
    pop dx
    pop cx
    pop bx
    pop ax
    iret
_int_handler endp
```

con l'ovvia conseguenza che la modifica apportata al valore di `BX` sarebbe vanificata dalla `POP BX` eseguita in seguito.

Ecco invece la definizione di `int_handler()` con un numero di paramateri formali (di tipo `int` o `unsigned int`) pari ai registri:

---

[246] Beh, non è sempre vero... Si consideri, ad esempio, una funzione dichiarata interrupt, compilata inserendo nel codice la direttiva `.386` (o `P80386`) e l'opzione `-3` sulla command line del compilatore (esse abilitano, rispettivamente da parte dell'assemblatore e del compilatore, l'uso delle istruzioni estese riconosciute dai processori 80386). Il compilatore non genera il codice necessario a salvare sullo stack i registri estesi (`EAX`, `EBX`, etc.), bensì si occupa solo dei registri a 16 bit. Se la funzione modifica i registri estesi (è evidente che il programma può comunque "girare" solo sugli 80386 e superiori), i 16 bit superiori di questi non vengono ripristinati automaticamente in uscita, con le immaginabili conseguenze per gli altri programmi sfruttanti le potenzialità avanzate del microprocessore. In casi come quello descritto il programmatore deve provvedere di propria iniziativa al salvataggio e rirpristino dei registri estesi modificati.

```
void interrupt int_handler(int Bp,int Di,int Si,int Ds,int Es,int Dx,
                           int Cx,int Bx,int Ax,int Ip,int Cs,int Flags)
{
    Bx = 0xABCD;                 // non usa il registro, bensi' il parametro formale
}
```

Il listato assembler risultante dalla compilazione è:

```
_int_handler proc far
    push ax
    push bx
    push cx
    push dx
    push es
    push ds
    push si
    push di
    push bp
    mov bp,DGROUP
    mov ds,bp
    mov bp,sp                                         ; serve ad accedere allo stack

    mov [bp+14],0ABCDh              ; non modifica BX, bensi' il valore nello stack

    pop bp
    pop di
    pop si
    pop ds
    pop es
    pop dx
    pop cx
    pop bx                                       ; carica in BX il valore modificato
    pop ax
    iret
_int_handler endp
```

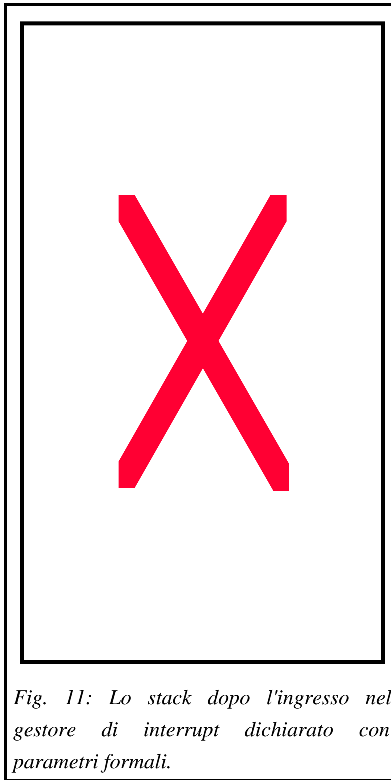

Fig. 11: Lo stack dopo l'ingresso nel gestore di interrupt dichiarato con parametri formali.

Come si vede, l'obiettivo è raggiunto. In cosa consiste la peculiarità delle funzioni `interrupt` nella gestione dello stack per i parametri formali (vedere pag. 87 e dintorni)? Esaminiamo con attenzione ciò che avviene effettivamente: la chiamata ad interrupt spinge sullo stack i `FLAGS`, `CS` e `IP`. Successivamente, la funzione `interrupt` copia, nell'ordine: `AX`, `BX`, `CX`, `DX`, `ES`, `DS`, `SI`, `DI` e `BP`. La struttura dello stack, dopo l'istruzione `MOV BP,SP` è dunque quella mostrata in figura 11. Proprio il fatto che la citata istruzione `MOV BP,SP` si trovi in quella particolare posizione, cioè dopo le `PUSH` dei registri sullo stack, e non in testa al codice preceduta solo da `PUSH BP` (posizione consueta nelle funzioni non `interrupt`), consente di referenziare i valori dei registri come se fossero i parametri formali della funzione. Se non vi fossero parametri formali, non vi sarebbe neppure (per effetto dell'opzione `-k-`) l'istruzione `MOV BP,SP`: il gestore potrebbe ancora referenziare le copie dei regsitri nello stack, ma i loro indirizzi andrebbero calcolati come offset rispetto a `SP` e non a `BP`. Da quanto detto sin qui, e dall'esame della figura 11, si evince inoltre che la lista dei parametri formali della funzione può essere allungata a piacere: i parametri addizionali consentono di accedere a quanto contenuto nello stack "sopra" i flag. Si tratta di un metodo, del tutto particolare, di passare parametri ad un interrupt: è sufficiente copiarli sullo stack prima della chiamata all'interrupt stesso (come al solito, si consiglia con insistenza di estrarli dallo stack al ritorno dall'interrupt). Ecco un esempio:

```
    ....
    _AX = 0xABCD;
    asm push ax;
    asm int 0x77;
    asm add sp,2;
    ....
```

L'esempio è valido nell'ipotesi che il programma abbia installato un gestore dell'int 77h definito come segue:

```
void interrupt newint77h(int BP,int DI,int SI,int DS,int ES,int DX,int CX,int BX,
                                              int AX,int IP,int CS,int FLAGS,int AddParm)
{
    ....
}
```

Il parametro `AddParm` può essere utilizzato all'interno della `newint77h()` e referenzia l'ultima word spinta sullo stack prima dei flag (cioè prima della chiamata ad interrupt): in questo caso `ABCDh`, il valore contenuto in `AX`. L'istruzione `ADD SP,2` ha lo scopo di estrarre dallo stack la word di cui sopra, eventualmente modificata da `newint77h()`.

Dal momento che l'ordine nel quale i registri sono copiati sullo stack è fisso, deve esserlo (ma inverso[247]) anche l'ordine nel quale sono dichiarati i parametri formali del gestore di interrupt. Si noti

[247] Meglio chiarire: `int Bp, int Di, int Si, int Ds, int Es, int Dx, int Cx, int Bx, int Ax, int Ip, int Cs, int Flags`, infine gli eventuali parametri aggiuntivi. Ciò a causa delle convenzioni C in fatto di passaggio dei parametri alle funzioni (il primo parametro spinto sullo stack è, in realtà, l'ultimo dichiarato: ancora una volta si rimanda a pag. 87 e seguenti); i nomi possono però essere scelti a piacere. Attenzione: l'ordine con cui i registri sono spinti sullo stack non è il medesimo per tutti i compilatori. Per eliminare

inoltre che non è sempre indispensabile dichiarare tutti i registri, da BP ai Flags, ma soltanto quelli che vengono effettivamente referenziati nel gestore, nonché quelli che li precedono nella dichiarazione stessa. Se, ad esempio, la funzione modifica AX e DX, devono essere dichiarati, nell'ordine, BP, DI, SI, DS, ES, DX, CX, BX e AX; non serve (ma nulla vieta di) dichiarare IP, CS e Flags. E' evidente che tutti i registri non dichiarati quali parametri del gestore possono essere referenziati da questo mediante lo inline assembly o gli pseudoregistri, ma il loro valore iniziale viene ripristinato in uscita. Esempio:

```
void interrupt int_handler(int Bp, int Di, int Si)
{
    ....                        // questo gestore può modificare solo BP, DI e SI
}
```

Consiglio da amico: è meglio non pasticciare con CS e IP, anche quando debbano essere per forza dichiarati (ad esempio per modificare i flag). I loro valori nello stack rappresentano l'indirizzo di ritorno dell'interrupt, cioè l'indirizzo della prima istruzione eseguita dopo la IRET: modificarli significa far compiere al sistema, al termine dell'interrupt stesso, un vero e proprio salto nel buio[248].

Ancora un'osservazione: le funzioni interrupt sono sempre void. Infatti, dal momento che le funzioni non void, prima di terminare, caricano AX (o DX:AX) con il valore da restituire, il ripristino in uscita dei valori iniziali dei registri implica l'impossibilità di restituire un valore al processo chiamante. Esempio:

```
int interrupt int_handler(int Bp,int Di,int Si,int Ds,int Es,int Dx,int Cx,
                                                                   int Bx,int Ax)
{
    Bx = 0xABCD;              /* non usa il registro, bensi' il parametro formale */
    return(1);
}
```

Il listato assembler risultante è:

```
_int_handler proc far
    push ax
    push bx
    push cx
    push dx
    push es
    push ds
    push si
    push di
    push bp
    mov bp,DGROUP
    mov ds,bp
    mov bp,sp                                        ; serve ad accedere allo stack

    mov [bp+14],0ABCDh               ; non modifica BX, bensi' il valore nello stack
    mov ax,1                ; referenzia comunque il registro, anche in presenza di
                                                                 ; parametri formali
    pop bp
```

eventuali problemi di portabilità è necessario conoscere a fondo il comportamento del compilatore utilizzato (marca e versione), di solito descritto nella documentazione con esso fornita.

[248] CS e IP, ovviamente, sono accessibili anche tramite inline assembly e gli pseudoregistri. In questo caso, però, modificarne il valore avrebbe l'effetto di far impazzire il sistema immediatamente. Infatti la modifica non riguarderebbe l'indirizzo di ritorno dell'interrupt, bensì l'indirizzo dell'istruzione da esguire subito dopo la modifica stessa.

```
    pop di
    pop si
    pop ds
    pop es
    pop dx
    pop cx
    pop bx                                  ; carica in BX il valore modificato
    pop ax                                 ; ripristina il valore inziale di AX
    iret
_int_handler endp
```

Il listato assembler evidenzia che l'istruzione `return` utilizza sempre (ovviamente) i registri e non le copie nello stack, anche qualora `AX` e `DX` siano gestibili come parametri formali.

Non si tratta, però, di un limite: si può anzi affermare che l'impossibilità di restituire un valore al processo chiamante è una caratteristica implicita delle routine di interrupt. Esse accettano parametri attraverso i registri, e sempre tramite questi restituiscono valori (anche più di uno). Inoltre, tali regole sono valide esclusivamente per gli interrupt software, che sono esplicitamente invocati da un programma (o dal DOS), il quale può quindi colloquiare con essi. Gli interrupt hardware, al contrario, non possono modificare il contenuto dei registri in quanto interrompono l'attività dei programmi "senza preavviso": questi non hanno modo di utilizzare valori eventualmente loro restituiti[249]. In effetti, le regole relative all'interfacciamento con routine (le cosiddette funzioni) mediante passaggio di parametri attraverso l'effettuazione di una copia dei medesimi nello stack, e restituzione di un unico valore in determinati registri (`AX` o `DX:AX`) sono convenzioni tipiche del linguaggio C; il concetto di interrupt, peraltro nato prima del C, è legato all'assembler.

LE FUNZIONI far

Abbiamo detto che due sono le caratteristiche fondamentali di ogni gestore di interrupt: è una funzione `far` e termina con una istruzione `IRET`. Da ciò si deduce che non è indispensabile ricorrere al tipo `interrupt` per realizzare una funzione in grado di lavorare come gestore di interrupt: proviamo a immaginare una versione più snella della `int_handler()`:

```
#pragma option -k-            // solito discorso: non vogliamo standard stack frame

void far int_handler(void)
{
    ....
    asm iret;
}
```

La nuova `int_handler()` ha il seguente listato assembler:

```
_int_handler proc far
    ....
    iret
_int_handler endp
```

La maggiore efficienza del codice è evidente. La gestione dei registri e dello stack è però lasciata interamente al programmatore, che deve provvedere a salvare e ripristinare quei valori che non debbono essere modificati. Inoltre, qualora si debba accedere a variabili globali, bisogna accertarsi che `DS`

[249] Anzi, per un interrupt hardware il modificare i registri equivarrebbe a mescolare le carte in tavola al programma interrotto.

assuma il valore corretto (quello del segmento DGROUP del programma che ha installato il gestore), operazione svolta in modo automatico, come si è visto, dalle funzioni dichiarate interrupt[250].

```
void far int_handler(void)
{
    asm {
        push ds;
        push ax;
        mov ax,DGROUP;
        mov ds,ax;                                            // carica DGROUP in ds
        pop ax;
    }
    ....
    asm {
        pop ds;
        iret;
    }
}
```

Occorre poi resistere alla tentazione di restituire un valore al processo interrotto: benché la funzione sia far e non interrupt, le considerazioni sopra espresse sull'interfacciamento con gli interrupt mantengono pienamente la loro validità. Inoltre un'istruzione return provoca l'inserimento, da parte del compilatore, di una RET, che mette fuori gioco l'indispensabile IRET: il codice

```
int far int_handler(void)
{
    ....
    return(1);
    asm iret;
}
```

diventa infatti

```
_int_handler proc far
    ....
    mov ax,1
    ret
    iret
_int_handler endp
```

con la conseguenza di produrre un gestore che, al rientro, "dimentica" sullo stack la word dei flag.

In un gestore far l'accesso ai registri può avvenire mediante lo inline assembly o gli pseudoregistri: in entrambi i casi ogni modifica ai valori in essi contenuti rimane in effetto al rientro nel processo interrotto. L'unica eccezione è rappresentata dai flag, ripristinati dalla IRET: il gestore di interrupt ha comunque a disposizione due metodi per aggirare l'ostacolo. Il primo consiste nel sostituire la IRET con una RET 2: l'effetto è quello di forzare il compilatore ad addizionare 2 al valore di SP in uscita al gestore[251], eliminando così dallo stack la word dei flag. Il secondo metodo si risolve nel dichiarare i flag come parametro formale del gestore, con una logica analoga a quella descritta per le funzioni di tipo interrupt. In questo caso, però, la funzione è di tipo far:

[250] Un'alternativa consiste nel memorizzare i dati globali in locazioni relative al valore di CS nel gestore e non al valore di DS. In pratica, occorre dichiarare una o più funzioni che servono unicamente a riservare spazio, mediante istruzioni inline assembly DB o DW o DD, per tali dati (in modo, cioè, che contengano dati e non codice eseguibile). I nomi delle funzioni, tramite opportune operazioni di cast, sono referenziati come puntatori ai dati: vedere pag. per una presentazione dettagliata dell'argomento.

[251] La sintassi RET n provoca un incremento di SP pari a n.

```
void far int_handler(int Flags)
{
    Flags |= 1;                // usa il parametro formale per settare il CarryFlag
    asm iret;
}
```

e pertanto il compilatore produce:

```
_int_handler proc far
    push bp
    mov bp,sp
    or word ptr [bp+6],1
    iret
_int_handler endp
```

La figura 12 evidenzia la struttura dello stack dopo l'ingresso nel gestore di interrupt (dichiarato `far`): nello stack, all'indirizzo (offset rispetto a `SS`) `BP+6`, c'è la word dei flag, spinta dalla chiamata ad interrupt: proprio perché la funzione è di tipo `far`, per il compilatore il primo parametro formale si trova in quella stessa locazione. Dopo quello relativo ai flag possono essere dichiarati altri parametri formali, i quali referenziano (come, del resto, nelle funzioni di tipo `interrupt`) il contenuto dello stack "sopra" i flag. E' superfluo (speriamo!) ricordare che un gestore di interrupt non viene mai invocato direttamente dal programma, ma attivato via hardware oppure mediante l'istruzione `INT`: in quest'ultimo caso il programma deve spingere sullo stack i valori che dovranno essere referenziati dal gestore come parametri formali. Vale la pena di precisare che, utilizzando l'opzione `-k-`, `BP` viene spinto sullo stack e valorizzato con `SP` solo se sono dichiarati uno o più parametri formali.

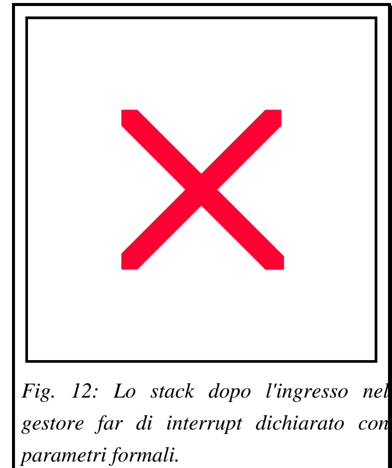

Fig. 12: Lo stack dopo l'ingresso nel gestore far di interrupt dichiarato con parametri formali.

Un'ultima osservazione: il tipo `far` della funzione risulta incoerente con il tipo `interrupt` richiesto dalla `setvect()` per il secondo parametro, rappresentante il nuovo vettore. Si rende allora necessaria un'operazione di cast.

```
void far int_handler(void)                    // definizione del gestore di interrupt
{
    ....
}
....

void install_handler(void)
{
    ....
    setvect(int_num,(void(interrupt *)(void))int_handler);
    ....
}
```

UTILIZZO DEI GESTORI ORIGINALI

Quando un gestore di interrupt viene installato, il suo indirizzo è copiato nella tavola dei vettori e sostituisce quello del gestore precedentemente attivo. La conseguenza è che quest'ultimo non viene più eseguito, a meno che il suo vettore non sia stato salvato prima dell'installazione del nuovo gestore, in modo che questo possa invocarlo, se necessario. Inoltre l'ultimo gestore installato è comunque il primo ad essere eseguito e deve quindi comportarsi in modo "responsabile". Si può riassumere l'insieme delle possibilità in un semplice schema:

MODALITÀ DI UTILIZZO DEI GESTORI ORIGINALI DI INTERRUPT

	COMPORTAMENTO	**CARATTERISTICHE**
1	Il nuovo gestore non invoca quello attivo in precedenza e, in uscita, restituisce il controllo al processo interrotto.	La funzione deve riprodurre in modo completo tutte le funzionalità indispensabili del precedente gestore, eccetto il caso in cui esso abbia proprio lo scopo di inibirle oppure occupi un vettore precedentemente non utilizzato.
2	Il nuovo gestore, in uscita, cede il controllo al gestore attivo in precedenza: quest'ultimo, a sua volta, terminato l'espletamento dei propri compiti, rientra al processo interrotto.	La funzione può delegare in tutto o in parte al gestore precedente l'espletamento delle funzionalità caratteristiche dell'interrupt. Questo approccio è utile soprattutto quando il nuovo gestore deve intervenire sullo stato del sistema (registri, flag, etc.) prima che esso venga conosciuto dalla routine originale (eventualmente per modificarne il comportamento).
3	Il nuovo gestore invoca quello attivo in precedenza come una subroutine e, ricevuto nuovamente da questo il controllo, ritorna al processo interrotto dopo avere terminato le proprie operazioni.	La funzione può delegare in tutto o in parte al gestore precedente l'espletamento delle funzionalità caratteristiche dell'interrupt. Questo approccio è seguito soprattutto quando il nuovo gestore ha necessità di conoscere i risultati prodotti dall'interrupt originale, o quando può risultare controproducente ritardarne l'esecuzione.

Nel caso 1 non vi è praticamente nulla da aggiungere a quanto già osservato.

Il caso 2, detto "concatenamento", merita invece alcuni approfondimenti. Innanzitutto va sottolineato che il nuovo gestore cede il controllo al gestore precedentemente attivo, il quale lo restituisce direttamente al processo interrotto: il nuovo gestore non ha dunque la possibilità di conoscere i risultati prodotti da quello originale, ma soltanto quella di influenzarne, se necessario, il comportamento modificando opportunamente il valore di uno o più registri.

Il controllo viene ceduto al gestore originale mediante un vero e proprio salto senza ritorno, cioè con un'istruzione `JMP`: bisogna ricorrere allo inline assembly. Vediamo un esempio di gestore dell'int 17h, interrupt BIOS per la stampante. Quando il programma lo installa, esso entra in azione ad ogni chiamata all'int 17h ed agisce come un filtro: se `AH` è nullo, cioè se è richiesto all'int 17h il servizio 0, che invia un byte in output alla stampante, viene controllato il contenuto di `AL` (il byte da stampare). Se questo è pari a `B3h` (decimale `179`, la barretta verticale), viene sostituito con 21h (decimale `33`, il punto esclamativo). Il controllo è poi ceduto al precedente gestore, con un salto (`JMP`) all'indirizzo di questo, ottenuto ad esempio mediante la `getvect()` e memorizzato nello spazio riservato dalla `GlobalData()`, in quanto esso deve trovarsi in una locazione relativa a `CS`[252].

```
#pragma option -k-                  // per gli smemorati: niente PUSH BP e MOV BP,SP
#define oldint17h  GlobalData

void GlobalData(void)                // funzione jolly: spazio per vettore originale
{
    asm db 3 dup(0);                       // 3 bytes + l'opcode della RET = 4 bytes
}

void far newint17h(void)
```

[252] Si è detto che il valore di `DS` non è noto a priori in ingresso al gestore: l'unico registro sul quale si può fare affidamento è `CS`. Vedere pag. per i dettagli.

```
{
    asm {
        cmp ah,0;
        jne ENDFUNC;
        cmp al,0xB3;
        jne ENDFUNC;
        mov al,0x21;
    }
ENDFUNC:
    asm jmp dword ptr oldint17h;
}
```

Chiaro, no? Se non effettuasse il salto alla routine originale, la `newint17h()` dovrebbe riprodurne tutte le funzionalità relative alla gestione della stampante. Se il programma fosse compilato con standard stack frame (senza la solita opzione `-k-`) sarebbe indispensabile un'istruzione in più, `POP BP`, per ripristinare lo stack:

```
void far newint17h(void)            // se non e' usata l'opzione -k- il compilatore
{                                   // mantiene la standard stack frame aggiungendo
    ....                              // PUSH BP e MOV BP,SP in testa alla funzione
    asm pop bp;                        // la POP BP ripristina lo stato dello stack
    asm jmp dword ptr oldint17h;
}
```

Senza la `POP BP`, la `IRET` del gestore originale restituirebbe il controllo all'indirizzo `IP:BP` e utilizzerebbe `CS` per ripristinare i flag: terribili guai sarebbero assicurati, anche senza considerare che la word dei "veri" flag rimarrebbe, dimenticata, nello stack.

Il metodo di concatenamento suggerito mantiene la propria validità anche con le funzioni di tipo `interrupt`; è sufficiente liberare lo stack dalle copie dei registri prima di effettuare il salto:

```
void interrupt newint17h(int Bp,int Di,int Si,int Ds,int Es,int Dx,int Cx,int Bx,
                                                                              int Ax)
{
    if(Ax < 0xFF)                                                 // vera solo se AH = 0
        if(Ax == 0xB3)                                          // vera solo se AL = B3h
            Ax = 0x21;
    asm {
        pop bp;
        pop di;
        pop si;
        pop ds;
        pop es;
        pop dx;
        pop cx;
        pop bx;
        pop ax;
        jmp dword ptr oldint17h;
    }
}
```

Si noti che `oldint17h` è, anche in questo caso, una macro che referenzia in realtà la funzione jolly contenente i dati globali: nonostante le funzioni `interrupt` provvedano alla gestione automatica di `DS`, in questo caso il valore originale del registro è già stato ripristinato dalla `POP DS`.

Non sempre il ricorso allo inline assembly è assolutamente indispensabile: la libreria del C Microsoft include la `_chain_intr()`, che richiede come parametro un vettore di interrupt ed effettua il concatenamento tra funzione di tipo `interrupt` e gestore originale. Di seguito presentiamo il listato di

una funzione analoga alla `_chain_intr()` di Microsoft, adatta però ad essere inserita nelle librerie C Borland[253].

```
/********************

    BARNINGA_Z! - 1992

    CHAINVEC.C - chainvector()

    void far cdecl chainvector(void(interrupt *oldint)(void));
    void(interrupt *oldint)(void); puntatore al gestore originale
    Restituisce: nulla

    COMPILABILE CON TURBO C++ 2.0

        bcc -O -d -c -k- -mx chainvec.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

void far cdecl chainvector(void(interrupt *oldint)(void))
{
    asm {
        add sp,6;
        mov bp,sp;
        mov ax,word ptr [oldint];
        mov bx,word ptr [oldint-2];
        add sp,8;
        mov bp,sp;
        xchg ax,word ptr [bp+16];
        xchg bx,word ptr [bp+14];
        pop bp;
        pop di;
        pop si;
        pop ds;
        pop es;
        pop dx;
        pop cx;
        ret;
    }
}
```

La chiamata alla `chainvector()` spinge sullo stack 5 word: la parte segmento e la parte offset di `oldint`, l'attuale coppia `CS:IP` e `BP`. La `chainvector()` raggiunge il proprio scopo ripristinando i valori dei registri copiati nello stack dalla funzione `interrupt` e modificando il contenuto dello stack medesimo in modo tale che la struttura di questo, prima dell'istruzione `RET`, divenga quella descritta in figura 13.

La `RET` trasferisce il controllo all'indirizzo seg:off di `oldint`, cioè al gestore originale, che viene così eseguito con lo stack contenente le 3 word (flag e `CS:IP`) salvate dalla chiamata che aveva attivato la funzione `interrupt`. In pratica, il gestore originale opera come se nulla fosse avvenuto, restituendo il controllo all'indirizzo `CS:IP` originariamente spinto sullo stack: in altre parole, al processo interrotto.

---

[253] Anche le più recenti versioni del C Borland includono una funzione analoga alla `_chain_intr()`, comunque, visto che ormai il lavoro è fatto...

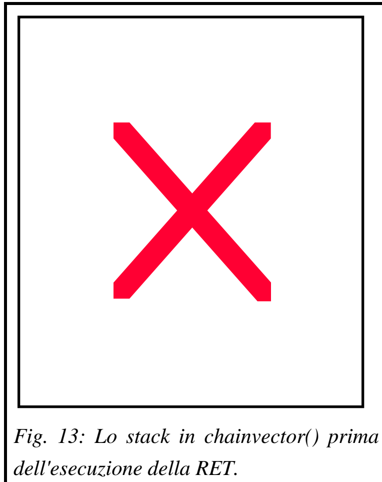

Fig. 13: Lo stack in chainvector() prima dell'esecuzione della RET.

Forse è opportuno sottolineare che la `chainvector()` può essere invocata solamente nelle funzioni dichiarate `interrupt` e che, data l'inizializzazione automatica di `DS` a `DGROUP` nelle funzioni `interrupt`, il puntatore al gestore originale può essere una normale variabile globale. Ovviamente, eventuali istruzioni inserite nel codice in posizioni successive alla chiamata a `chainvector()` non verranno mai eseguite. La `chainvector()` è dichiarata `far`, ed il suo unico parametro è un puntatore a 32 bit, pertanto il listato è valido per qualunque modello di memoria (pag. 143). Il trucco, lo ripetiamo, sta nel modificare lo stack in modo tale che esso contenga 5 word: i flag e la coppia `CS:IP` salvati dalla chiamata ad interrupt, più la coppia segmento:offset rappresentante l'indirizzo del gestore originale. A questo punto è sufficiente che la funzione `far` che effettua il concatenamento termini, eseguendo la `RET`, poiché questa non può che utilizzare come indirizzo di ritorno l'indirizzo del gestore originale. Di seguito è presentato un esempio di utilizzo, in cui `oldint17h` è una normale variabile C, dichiarata come puntatore ad interrupt, inizializzata al valore del vettore dell'int 17h mediante la `getvect()`.

```
void interrupt newint17h(int Bp,int Di,int Si,int Ds,int Es,int Dx,int Cx,int Bx,
                                                                              int Ax)
{
    if(Ax < 0xFF)                                                  // vera solo se AH = 0
        if(Ax == 0xB3)                                             // vera solo se AL = B3h
            Ax = 0x21;
    chainvector(oldint17h);
}
```

Nel caso di gestori di interrupt dichiarati `far`, è ancora possibile scrivere una funzione in grado di effettuare il concatenamento ma, mentre nelle funzioni `interrupt` la struttura dello stack è sempre quella rappresentata nella figura 11, con riferimento ai gestori `far` bisogna distinguere tra parecchie situazioni differenti, a seconda che sia utilizzata oppure no l'opzione `-k-` in compilazione, che il gestore sia definito con parametri formali o ne sia privo e che esso faccia o meno uso di varialbili locali; è inoltre indispensabile che il vettore originale sia salvato in una locazione relativa a `CS` e non a `DS`. La varietà delle situazioni che si possono di volta in volta presentare è tale da rendere preferibile, in quanto più semplice e sicuro, il ricorso all'istruzione `JMP` mediante lo inline assembly[254].

[254] A puro titolo di esempio, si propone il listato di una versione della `chainvector()` valida per gestori `far` privi di parametri formali e di variabili locali, compilati con opzione `-k-`:

```
void far chainvector(void(interrupt *oldint)(void))
{
    asm {
        pop bp;
        add sp,4;
        ret;
    }
}
```

Se il gestore far avesse un parametro formale (ad esempio: i flag) e nessuna variabile locale, la presenza di una word in più (`BP`) nello stack imporrebbe l'uso di una `chainvector()` diversa dalla precedente versione:

```
void far chainvector(void(interrupt *oldint)(void))
{
    asm {
        push ax;
```

Veniamo ora all'esame del caso 3: il gestore di interrupt utilizza la routine originale come subroutine; esso ha dunque la possibilità sia di influenzarne il comportamento modificando i registri, sia di conoscerne l'output (se prodotto) dopo avere da essa ricevuto nuovamente il controllo del sistema.

Un tipico esempio è rappresentato, solitamente, dai gestori dell'int 08h, interrupt hardware eseguito dal clock circa 18 volte al secondo[255]. L'int 08h svolge alcuni compiti, di fondamentale importanza per il buon funzionamento del sistema[256], dei quali è buona norma evitare di ritardare l'esecuzione: appare allora preferibile che il gestore, invece di portare a termine il proprio task e concatenare la routine originale, invochi dapprima quest'ultima e solo in seguito compia il proprio lavoro[257].

Il metodo che consente di invocare un interrupt come una subroutine è il seguente:

```
void (interrupt *old08h)(void);

    ....

    old08h = getvect(0x08);
    setvect(0x08,new08h);

    ....

void interrupt new08h(void)
{
    (*old08h)();
```

```
        mov ax,word ptr [bp+8];
        xchg ax,word ptr[bp+10];
        mov word ptr[bp+8],ax;
        mov ax,word ptr[bp+6];
        xchg ax,word ptr[bp+8];
        mov bp,ax;
        pop ax;
        add sp,8;
        ret;
    }
}
```

In entrambi i casi il parametro `oldint` deve essere definito in una locazione relativa a `CS`, in quanto `DS` deve comunque essere ripristinato prima della chiamata a `chainvector()`. Si aggiunga che i due casi presentati sono solamente alcuni tra quelli che possono effettivamente verificarsi. Per questi motivi si è detto che nei gestori `far` è preferibile utilizzare direttamente l'istruzione `JMP` piuttosto che implementare differenti versioni di `chainvector()` e utilizzare di volta in volta quella adatta.

[255] Per la precisione: 18,21 volte, cioè ogni 55 millisecondi.

[256] Incrementa il contatore del timer di sistema (un `long int` situato all'indirizzo `0:46C`); decrementa il contatore del sistema di spegnimento del motore dei floppy drives (un byte a `0:440`) se non è zero (quando questo raggiunge lo zero il motore viene spento e il flag di stato del motore (un byte a `0:43F`) è aggiornato); genera un int 1Ch.

[257] Piccola digressione: l'int 08h, per la verità, mette a disposizione un meccanismo analogo, rappresentato dall'int 1Ch. Questo, la cui routine di default è semplicemente una `IRET`, viene invocato dalla routine dell'int 08h al termine delle proprie operazioni. Installando un gestore per l'int 1Ch si ha la certezza che esso sia eseguito ad ogni timer tick. La differenza tra questo approccio e quello descritto (a titolo di esempio) nel paragrafo è che l'int 1Ch è eseguito ad interrupt hardware disabilitati (l'int 08h è l'interrupt hardware di massima priorità dopo il *Non Maskable Interrupt* (NMI), che gestisce situazioni di emergenza gravissima); al contrario, il nuovo gestore dell'int 08h esegue ad interrupt abilitati tutte le operazioni successive alla chiamata alla routine originale.

```
    ....
}
```

Il puntatore ad interrupt `oldint08h` viene inizializzato, mediante la `getvect()`, al valore del vettore dell'int 08h, il quale è invocato dal nuovo gestore con una semplice indirezione del puntatore[258]. Il compilatore è abbastanza intelligente da capire, vista la particolare dichiarazione del puntatore[259], che la funzione puntata deve essere invocata in maniera particolare: occorre salvare il registro dei flag sullo stack prima della `CALL`, dal momento che la funzione termina con una `IRET` anziché con una semplice `RET`.

I patiti dell'assembly possono sostituirsi al compilatore e fare tutto da soli:

```
void oldint08h(void)
{
    asm db 3 dup(0);
}

    ....

    (void(interrupt *)(void))*(long far *)oldint08h = getvect(0x08);

    ....

void interrupt newint08h(void)
{
    ....
    asm {
        pushf;
        call dword ptr oldint08h;
    }
    ....
}
```

Si nota subito che il gestore originale è attivato mediante l'istruzione `CALL`; non sarebbe possibile, ovviamente, utilizzare la `INT` perché questa eseguirebbe ancora il nuovo gestore, causando un loop senza possibilità di uscita[260]. La `PUSHF` è indispensabile: essa salva i flag sullo stack e costituisce, come detto, il "contrappeso" della `IRET` che chiude la routine di interrupt. Non va infatti dimenticato che la `CALL` è, di norma, utilizzata per invocare routine "normali", terminate da una `RET`, pertanto essa spinge sullo stack solamente l'indirizzo di ritorno (la coppia `CS:IP`), con la conseguenza che i flag devono essere gestiti a parte. Inutile aggiungere che non si deve assolutamente inserire una `POPF` dopo la `CALL`, in quanto i flag sono estratti dallo stack dalla `IRET`.

E' interessante sottolineare che l'algoritmo descritto è valido ed applicabile tanto nei gestori dichiarati `interrupt` quanto in quelli dichiarati `far`, con la sola differenza che in questi ultimi

---

[258] Il C si presta di per sé ai giochi di prestigio. Il concetto è, tutto sommato, semplice: se, per esempio, l'indirezione di un puntatore restituisce il valore contenuto nella variabile puntata, detta indirezione sostituisce, in pratica, quel valore; allora l'indirezione di un puntatore ad una funzione (cioè ad una porzione di codice eseguibile) può essere considerata equivalente al valore restituito da quella funzione (ed è quindi necessario invocare la funzione per conoscere il valore da essa restituito). Vedere pag. 93.

[259] Infatti `oldint08h` punta ad una funzione `interrupt`, non ad una funzione qualsiasi.

[260] In effetti, mentre con la `CALL` viene specificato l'indirizzo della routine da eseguire, con la `INT` viene specificato il numero dell'interrupt, il cui indirizzo viene ricavato dalla tavola dei vettori, nella quale vi è, evidentemente, quello del nuovo gestore (che sta effettuando la chiamata).

l'indirizzo del gestore originale deve trovarsi in una locazione relativa a `CS`, per gli ormai noti problemi legati alla gestione di `DS` (vedere pag. e seguenti).

DUE O TRE ESEMPI

Gestire gli interrupt conferisce al programma una notevole potenza, poiché lo mette in grado di controllare da vicino l'attività di tutto il sistema. Vi sono però delle restrizioni a quanto le routine di interrupt possono fare in determinate circostanze: per alcuni dettagli sull'argomento si rimanda al capitolo dedicato ai TSR, ed in particolare alle pagine e seguenti. In questa sede presentiamo qualche esempio pratico di gestore di interrupt, nella speranza di offrire spunti interessanti.

Inibire CTRL-C e CTRL-BREAK

L'interrupt BIOS 1Bh è generato dal rilevamento della sequenza `CTRL-BREAK` sulla tastiera. L'int 1Bh installato dal BIOS è costituito semplicemente da una `IRET`. Il DOS installa al bootstrap un proprio gestore, che valorizza un flag, controllato poi periodicamente per determinare se sia stato richiesto un BREAK. Le sequenze `CTRL-C` sono intercettate dall'int 16h, che gestisce i servizi software BIOS per la tastiera. In caso di `CTRL-C` o `CTRL-BREAK` il controllo è trasferito all'indirizzo rappresentato dal vettore dell'int 23h. Per evitare che una sequenza `CTRL-C` o `CTRL-BREAK` provochi l'uscita a DOS del programma, è necessario controllare l'interrupt 1Bh (BIOS BREAK), per impedire la valorizzazione del citato flag, e l'interrupt 16h, per mascherare le sequenze `CTRL-C`. Inoltre occorre salvare in locazioni relative a `CS` i vettori originali. Vediamo come procedere.

```
/********************

    BARNINGA_Z! - 1992

    CTLBREAK.C - funzioni per inibire CTRL-C e CTRL-BREAK

    void far int16hNoBreak(void);      gestore int 16h
    void far int1BhNoBreak(void);      gestore int 1Bh
    void oldint16h(void);              funzione fittizia: contiene il vettore
                                       originale dell'int 16h
    void oldint1Bh(void);              funzione fittizia: contiene il vettore
                                       originale dell'int 1Bh

    COMPILABILE CON TURBO C++ 2.0

        bcc -O -d -c -k- -mx ctlbreak.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

#pragma  inline
#pragma  option -k-

void oldint16h(void)               // funzione fittizia: locazione relativa a CS per
{                                       // salvare il vettore originale dell'int 16h
    asm db 3 dup (0)                      // 3 bytes + 1 byte (opcode RET) = 4 bytes
}

void oldint1Bh(void)               // funzione fittizia: locazione relativa a CS per
{                                       // salvare il vettore originale dell'int 1Bh
    asm db 3 dup (0)                      // 3 bytes + 1 byte (opcode RET) = 4 bytes
}
```

```
void far int1BhNoBreak(void)          // nuovo gestore int 1Bh: non fa proprio nulla
{
    asm iret;
}

void far int16hNoBreak(void)           // nuovo gestore int 16h: fa sparire i CTRL-C
{
    asm {
        sti;
        cmp ah,00H;                        // determina qual e' il servizio richiesto
        je SERV_0;
        cmp ah,10H;
        je SERV_0;
        cmp ah,01H;
        je SERV_1;
        cmp ah,11H;
        je SERV_1;
        jmp dword ptr oldint16h;             // altro servizio: concatena vett.orig.
    }

SERV_0:                              // richiesto servizio 00h o 10h: attendere tasto

    asm {
        push dx;                  // usa DX per incrementare AX: cosi' se il servizio
        mov dx,ax;                // chiesto e' 00h o 10h e' simulato con il servizio
        inc dh;                       // 01h o 11h rispettivamente. In pratica, se e'
    }                   // chiesto di attendere un tasto, int16hNoBreak() si limita a
                       // controllare in loop se c'e' un tasto nel buffer di tastiera
LOOP_0:
CTRL_C_0:

    asm {
        mov ax,dx;
        pushf;
        cli;
        call dword ptr oldint16h;               // subroutine: c'e' tasto in attesa ?
        jz LOOP_0;                                      // no: continua a controllare
        mov ax,dx;                        // si: usa servizio 00h o 10h per prelevarlo
        dec ah;
        pushf;
        cli;
        call dword ptr oldint16h;             // subroutine: preleva tasto da buffer
        cmp al,03H;                              // e' CTRL-C o CTRL-2 (ASCII 03h) ?
        je CTRL_C_0;                               // si: lo ignora e torna nel loop
        cmp ax,0300H;                                // no: e' ALT-003 (su keypad) ?
        je CTRL_C_0;                               // si: lo ignora e torna nel loop
        pop dx;                        // no: niente CTRL-C o simili. Ripristina DX
        iret;                        // e ritorna, restituendo il tasto al programma
    }

SERV_1:               // richiesto serv.01h o 11h: controllare se c'e' tasto in buff.

    asm {
        pushf;
        cli;
        call dword ptr oldint16h;             // subroutine: controlla se c'e' tasto
        jz EXIT_FUNC;             // no: restituisce controllo a programma chiamante
        pushf;                 // si: salva flags perche' saranno modificati dai test
        cmp al,03H;                              // e' CTRL-C o CTRL-2 (ASCII 03h) ?
        je CTRL_C_1;                            // si: prende opportuni provvedimenti
        cmp ax,0300H;                                // no: e' ALT-003 (su keypad) ?
        je CTRL_C_1;                            // si: prende opportuni provvedimenti
        popf;                        // no: niente CTRL-C o simili. Ripristina flags
```

```
        jmp EXIT_FUNC;                                                  // ed esce
    }

CTRL_C_1:               // il servizio 01h o 11h richiesto dal programma ha rilevato
                        // la presenza di un CTRL-C in attesa nel buffer di tastiera
    asm {
        mov ah,00H;                    // usa il servizio 00h per estrarre il CTRL-C
        pushf;                                                 // dal buffer di tastiera
        cli;
        call dword ptr oldint16h;             // subroutine: preleva tasto da buffer
        popf;                    // POPF controparte della PUSHF prima dei controlli
        xor ah,ah;             // dice al programma che non c'era alcun tasto pronto
    }

EXIT_FUNC:

    asm ret 2;                          // esce e preserva il nuovo valore dei flags
}
```

I commenti inseriti nel listato rendono inutile insistere sulle particolarità della `int16hNoBreak()`, la quale, tra l'altro, comprende in modo completo tutte le modalità di utilizzo dei gestori originali e di rientro al processo interrotto. Vale comunque la pena di riassumerne brevemente la logica: se il programma richiede all'int 16h il servizio 00h o 10h (estrarre un tasto dal buffer di tastiera e, se questo è vuoto, attendere l'arrivo di un tasto), la `int16hNoBreak()` entra in realtà in un loop nel quale utilizza il servizio 01h o 11h del gestore originale (controllare se nel buffer è in attesa un tasto). In caso affermativo lo preleva col servizio 00h o 10h e lo verifica: se è un `CTRL-C` (o simili) fa finta di nulla, cioè lo ignora e rientra nel ciclo; altrimenti restituisce il controllo (e il tasto) al programma chiamante. Se invece il programma richiede il servizio 01h o 11h, questo è effettivamente invocato, ma, prima di restituire la risposta, se un tasto è presente viene controllato. Se si tratta di `CTRL-C` è estratto dal buffer mediante il servizio 00h o 10h e al programma viene "risposto" che non vi è alcun tasto in attesa; altrimenti il tasto è lasciato nel buffer e restituito al programma. L'istruzione `RET 2` consente al programma di verificare l'eventuale presenza del tasto mediante il valore assunto dallo `ZeroFlag`. Se utilizzata in un programma TSR, la `int16hNoBreak()` può essere modificata come descritto a pag. per consentire ad altri TSR di assumere il controllo del sistema durante i cicli di emulazione del servizio 00h.

Accodando al listato appena commentato la banalissima `main()` listata di seguito si ottiene un programmino che conta da `1` a `1000`: durante il conteggio `CTRL-C` e `CTRL-BREAK` sono disabilitati. Provare per credere.

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

void main(void)
{
    register i;

    asm cli;
    (void(interrupt *)(void))*(long far *)oldint16h = getvect(0x16);
    (void(interrupt *)(void))*(long far *)oldint1Bh = getvect(0x1B);
    setvect(0x16,(void(interrupt *)(void))int16hNoBreak);
    setvect(0x1B,(void(interrupt *)(void))int1BhNoBreak);
    asm sti;
    for(i = 1; i <= 1000; i++)
        printf("%04d\n",i);
    asm cli;
    setvect(0x16,(void(interrupt *)(void))*(long far *)oldint16h);
    setvect(0x1B,(void(interrupt *)(void))*(long far *)oldint1Bh);
    asm sti;
}
```

Inibire CTRL-ALT-DEL

La pressione contemporanea dei tasti CTRL, ALT e DEL (o CANC) provoca un bootstrap (warm reset) della macchina. Per impedire che ciò avvenga durante l'elaborazione di un programma, è sufficiente che questo installi un gestore dell'int 09h, interrupt hardware per la tastiera, che intercetta le sequenze CTRL-ALT-DEL e le processa[261] senza che queste raggiungano il buffer di tastiera.

```
/********************

    BARNINGA_Z! - 1992

    CTLALDEL.C - funzioni per inibire CTRL-ALT-DEL

    void far int09hNoReset(void);      gestore int 09h
    void oldint09h(void);              funzione fittizia: contiene il vettore
                                       originale dell'int 09h

    COMPILABILE CON TURBO C++ 2.0

        bcc -O -d -c -k- -mx ctlaldel.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

#pragma  inline
#pragma  option -k-

void oldint09h(void)
{
    asm db 3 dup (0);
}

void far int09hNoReset(void)
{
    asm {
        push ax;                                  // salva i registri utilizzati
        push bx;
        push es;
        pushf;                   // salva i flags (saranno alterati dai confronti)
        in al,60h;                            // legge lo scan code sulla porta 60h
        cmp al,53h                                               // e' il tasto DEL ?
        jne CHAIN_OLD;                      // no: trasferisce il controllo al BIOS
        push 0;
        pop es;
        mov bx,417h;                            // ES:BX punta allo shift status byte
        mov al,es:[bx];                         // carica AL con lo shift status byte
        inc bx;                         // ES:BX punta all'extended shift status byte
        mov ah,es:[bx];            // l'extended shift status byte è caricato in AH
        test al,00000100b;             // controlla se è premuto uno dei tasti CTRL
        jnz TEST_ALT;
        test ah,00000001b;
        jz CHAIN_OLD;
    }

TEST_ALT:
```

[261] La sequenza CTRL-ALT-DEL ha il seguente effetto: il valore 1234h (flag per l'effettuazione di un warm reset) è copiato alla locazione 0:472 e viene eseguito un salto all'indirizzo FFFF:0, indirizzo standard del ROM-BIOS ove si trova una seconda istruzione di salto all'indirizzo della routine che effettua il bootstrap.

```
    asm {
        test al,00001000b;                // se uno dei tasti CTRL è premuto, allora
        jnz NO_RESET;              // controlla se è premuto anche uno dei tasti ALT
        test al,00000010b;
        jz CHAIN_OLD;
    }

NO_RESET:                    // CTRL-ALT-DEL premuto: ignora la sequenza e restituisce
                                   // il controllo direttamente al processo interrotto
    asm {                           // trattandosi di un gestore di interrupt hardware
        in al,61h;              // deve riabilitare il dispositivo gestito (tastiera)
        mov ah,al;                     // e segnalare al controllore degli interrupts
        or al,80h;                                    // che l'interrupt è terminato
        out 61h,al;                                         // riabilita la tastiera
        xchg ah,al;
        out 61h,al;
        mov al,20h;                               // segnala fine interrupt hardware
        out 20h,al;
        popf;                                                       // pulisce stack
        pop es;
        pop bx;
        pop ax;
    }
    asm iret;

CHAIN_OLD:                      // se viene concatenato il gestore originale, tutte le
                                  // operazioni di gestione hardware vengono effettuate
    asm {                                  // da questo: non rimane che pulire lo stack
        popf;
        pop es;
        pop bx;
        pop ax;
    }
    asm jmp dword ptr oldint09h;
}
```

Anche in questo caso una semplice `main()` consente di sperimentare il gestore: durante il conteggio da `1` a `1000` il reset mediante `CTRL-ALT-DEL` è disabilitato.

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

void main(void)
{
    register i;

    asm cli;
    (void(interrupt *)(void))*((long far *)oldint09h) = getvect(0x09);
    setvect(0x09,(void(interrupt *)(void))int09hNoReset);
    asm sti;
    for(i = 1; i <= 1000; i++)
        printf("%04d\n",i);
    asm cli;
    setvect(0x09,(void(interrupt *)(void))*((long far *)oldint09h));
    asm sti;
}
```

Attenzione: il programma non deve per nessun motivo essere interrotto con `CTRL-C` o `CTRL-BREAK`: il nuovo gestore rimarrebbe attivo, ma la RAM da esso occupata verrebbe disallocata e potrebbe essere sovrascritta dai programmi successivamente lanciati. L'effetto sarebbe quasi certamente il blocco del sistema, con la necessità di effettuare un cold reset. Può essere interessante sperimentare un

programma "a prova di bomba" riunendo `int09hNoReset()`, `int16hNoBreak()` e `int1BhNoBreak()` (nonché le funzioni fittizie per i vettori originali[262]) in un unico listato ed aggiungendo una `main()` che installi e disinstalli tutti e tre i nuovi gestori.

Redirigere a video l'output della stampante

In questo esempio ritroviamo l'ormai noto[263] interrupt 17h, calato, questa volta, in un caso concreto. Per dirigere sul video l'output della stampante occorre intercettare ogni byte inviato in stampa, sottrarlo all'int 17h e "consegnarlo" all'int 10h, che gestisce i servizi BIOS per il video. In particolare, si può ricorrere all'int 10h, servizio 0Eh, che scrive a video in modalità teletype (scrive un carattere e muove il cursore, interpretando i caratteri di controllo quali `CR` e `LF`: proprio come una stampante).

```
/********************

    BARNINGA_Z! - 1992

    PRNTOSCR.C - funzioni per dirigere a video l'output della stampante

    void far int17hNoPrint(void);      gestore int 17h
    void oldint17h(void);              funzione fittizia: contiene il vettore
                                       originale dell'int 17h
    void grfgcolor(void);              funzione fittizia: contiene il byte per
                                       il colore di foreground su video grafico

    COMPILABILE CON TURBO C++ 2.0

        bcc -O -d -c -k- -mx prntoscr.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

#pragma  inline
#pragma  option -k-

void oldint17h(void)
{
    asm db 3 dup (0);
}

void grfgcolor(void)                  // contiene un byte, inzializzato a 7 (bianco)
{                                    // usato per stabilire il colore di foreground a
    asm db 7;                        // video se questo e' in modo grafico. Il valore
}                                              // puo' essere modificato da programma

void far int17hNoPrint(void)
{
    asm {
        or ah,ah;                    // e' richiesto servizio 0 (stampa byte in AL) ?
        jne CHAINOLD;                               // no: concatena gestore originale
        push ax;                                                       // si: salva AL
        mov ah,0x0F;                                      // richiede modo video attuale
```

[262] A dire il vero, dal momento che non si tratta di un programma TSR, si potrebbero tranquillamente utilizzare normali puntatori a funzione `interrupt`.

[263] Lo abbiamo conosciuto a pagina 262.

```
        int 0x10;
        cmp al,0x03;                    // se 0-3 o 7 allora e' un modo testo; in BH
        jle TTYWRITE;                             // c'e' il numero di pagina attiva
        cmp al,0x07;
        je TTYWRITE;
        mov bl,byte ptr grfgcolor;               // modo grafico: attiva col.foregr.
    }

TTYWRITE:

    asm {
        pop ax;                                                         // ricarica AL
        mov ah,0x0E;                                          // richiede servizio TTY
        int 0x10;
    }

EXITFUNC:

    asm {
        mov ah,0x80;                      // simula condizione di "stampante pronta"
        iret;
    }

CHAINOLD:

    asm jmp dword ptr oldint17h;
}
```

La `int17hNoPrint()` può essere installata da un TSR: da quel momento in avanti tutto l'output diretto alla stampante è invece scritto a video. Se questo è in modalità grafica, il byte memorizzato nella funzione fittizia `grfgcolor()` è utilizzato per attivare il colore di foreground: esso è inzializzato a 7 (bianco), ma può essere modificato con un'istruzione analoga alla seguente:

```
    *((char *)grfgcolor) = NEW_COLOR;
```

ove `NEW_COLOR` è una costante manifesta definita con una direttiva `#define`. Per un esempio di calcolo degli attributi video si veda pag. 455.

I PROGRAMMI TSR

TSR è acronimo di Terminate and Stay Resident. Un TSR è pertanto un programma che, quando termina, non consente al DOS di liberare la RAM da esso occupata: al contrario, vi rimane residente; l'interprete dei comandi (generalmente `COMMAND.COM`) riprende il controllo e ricompare a video il prompt, cioè il segnale che il DOS è in attesa di un nuovo comando da eseguire. Il TSR, nel frattempo, effettua un monitoraggio costante della situazione (attraverso la gestione di una o più routine di interrupt) e, al verificarsi delle condizioni prestabilite, interrompe l'attività svolta dal DOS o dall'applicazione in corso di esecuzione per tornare ad agire in *foreground*, cioè in primo piano.

Qual è l'utilità dei TSR? In generale si può affermare che essi devono la loro ragion d'essere al fatto che il DOS è un sistema operativo *single user* e *single tasking*; esso è cioè in grado di eseguire una sola applicazione alla volta. I TSR costituiscono un parziale rimedio a questa limitazione, proprio perché essi sono in grado di nascondersi dietro le quinte ed apparire quando necessario sovrapponendosi al, o meglio interrompendo il, *foreground task*.

TIPI DI TSR

I TSR possono essere classificati in due categorie: attivi e passivi, a seconda dell'evento che ne determina il ritorno in foreground.

Un TSR passivo assume il controllo del sistema solo quando vi è una esplicita richiesta da parte di un altro programma eseguito in foreground, ad esempio attraverso una chiamata ad un interrupt software gestito dal TSR stesso.

Un TSR attivo è invece "risvegliato" da un evento esterno al programma in foreground: ad esempio la pressione di una certa combinazione di tasti o, più in generale, dal verificarsi di un predefinito interrupt hardware.

Appare evidente che un TSR, quando viene attivato, agisce nel contesto del programma del quale interrompe l'attività (ne condivide stack, handle per file aperti, e così via): i TSR passivi possono assumere che il terreno sia stato loro opportunamente preparato dal programma chiamante, e quindi la loro struttura può essere relativamente semplice. Diversa è la situazione per i TSR attivi: essi sono invocati in modo asincrono[264] e pertanto, alla loro attivazione, devono necessariamente controllare lo stato del BIOS, del DOS e del programma in foreground onde evitare di danneggiare l'ambiente in cui essi stanno per operare. La loro struttura è pertanto più complessa: quanto esposto nei prossimi paragrafi concerne in modo particolare proprio i TSR attivi, pur non perdendo validità con riferimento a quelli di tipo passivo.

LA STRUTTURA DEL TSR

Il codice di un TSR si suddivide solitamente in due segmenti. Il primo ha il compito di caricare in memoria il programma, provvedere a tutte le operazioni necessarie all'installazione del TSR e restituire il controllo al DOS (la funzione `main()` ne è un banale esempio). Questa porzione di codice non ha più alcuna utilità a partire dal momento in cui il programma è residente, pertanto la RAM da essa occupata può venire liberata a vantaggio dei programmi che verranno utilizzati in seguito: per tale motivo essa è detta parte *transiente*. Il secondo costituisce, al contrario, la parte di codice destinata a rimanere attiva in background (sottofondo) ai programmi successivamente eseguiti. E' questa la parte denominata *residente*

[264] Significa che lo stato dello hardware, del DOS e del programma in foreground non è conosciuto al momento dell'attivazione.

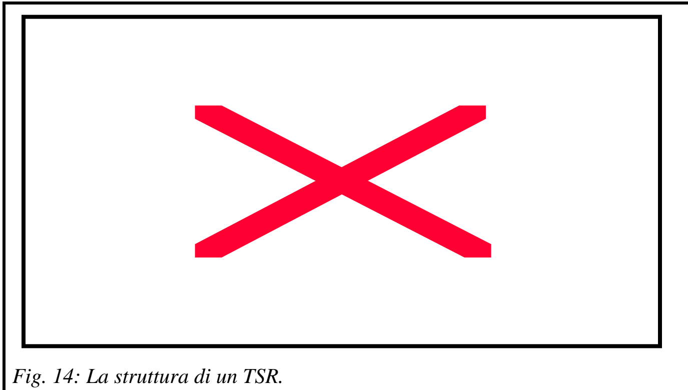

Fig. 14: La struttura di un TSR.

(figura 14), che solitamente si compone a sua volta di due categorie di routine: quelle dedicate al monitoraggio del sistema, che devono "intercettare" il segnale di attivazione del TSR, e quelle che svolgono le attività proprie del TSR medesimo, le quali possono essere le più svariate (si pensi, ad esempio, alle agende "pop-up").

La parte transiente di codice deve dunque essere in grado di determinare la quantità di memoria necessaria alla parte residente, e richiedere al DOS l'allocazione di tale quantità soltanto[265]. Il problema è reso complesso dalla necessità di allocare in modo accorto i dati necessari al TSR: sia quelli gestiti dalle routine transienti, sia quelli indispensabili al codice residente. I paragrafi che seguono analizzano in dettaglio gli argomenti sin qui accennati.

INSTALLAZIONE DEL TSR

Con il termine installazione si indicano le operazioni necessarie per terminare l'esecuzione del TSR e renderlo permanente in RAM. L'installazione viene normalmente effettuata mediante la funzione di libreria `keep()`:

```
....
keep(errlevel,resparas);
....
```

La variabile `errlevel` (di tipo `unsigned char`) contiene il valore che viene restituito dal programma al DOS[266], mentre `resparas` (di tipo `unsigned int`) contiene il numero di paragrafi (blocchi di 16 byte) che sono riservati dal DOS al programma per la sua permanenza in RAM.

INT 21H, SERV. 31H: TERMINA MA RESTA RESIDENTE IN RAM

Input	`AH`	`31h`
	`AL`	codice restituito al DOS
	`DX`	blocchi di 16 byte di RAM riservati al programma
Note		La memoria allocata mediante int 21h, funz. 48h non viene liberata.
		I file aperti non vengono chiusi.
		Il valore di `AL` può essere letto dalla successiva applicazione mediante int 21h, serv. 4Dh.

[265] Considerazioni analighe valgono con riferimento ai device driver: vedere la figura di pag. 356; vedere anche pag 363.

[266] Tale valore può essere utilizzato da un programma batch mediante l'istruzione DOS "`IF ERRORLEVEL...`" (vedere pag. 105).

INT 21H, SERV. 4DH: CODICE DI USCITA DELL'APPLICAZIONE TERMINATA

Input	`AH`	`4Dh`
Output	`AX`	codice di ritorno dell'applicazione terminata

Mentre il valore di `errlevel` è normalmente lasciato alla scelta del programmatore, in quanto esso non ha alcuna rilevanza tecnica per il buon funzionamento del TSR, il valore di `resparas` è sempre critico. Infatti si comprende, peraltro senza sforzi sovrumani, che se la porzione di RAM riservata al TSR è sottodimensionata, una parte del suo codice viene sovrascritta (e dunque distrutta) dai programmi successivamente eseguiti; in caso contrario si spreca una risorsa preziosa: si deve dunque riservare la RAM strettamente necessaria a "parcheggiare" tutto e solo quello che serve. Si consideri la figura 14: se la RAM riservata al TSR è tanto ampia da contenerne tutto il codice non si hanno problemi di alcun genere, salvo quello dello spreco. Se la regione di memoria non è sufficiente a contenere almeno il codice e i dati necessari al monitoraggio e al funzionamento delle routine residenti, le conseguenze sono imprevedibili (e, di solito, disastrose). Purtroppo non è sempre facile individuare il confine esatto tra ciò che serve e ciò che si può gettare senza problemi: è necessario qualche approfondimento.

DATI, STACK E LIBRERIE

Molto spesso nelle routine di installazione e in quelle residenti sono utilizzati i medesimi dati: le prime hanno, infatti, anche il compito di predisporre quanto serve al corretto funzionamento delle seconde. Si pensi, ad esempio, ai vettori di interrupt originali: questi sono di solito modificati dopo essere stati opportunamente salvati dalle routine transienti e proprio i valori salvati devono essere accessibili alle routine residenti per trasferire ad essi, quando necessario, il controllo del sistema (vedere pag. e seguenti).

In casi come quello descritto si ricorre, di norma, a variabili globali poiché esse sono accessibili da qualunque punto del codice, dunque non solo dalle routine transienti, ma anche da quelle residenti. Per queste ultime esiste però un limite: la RAM occupata dai dati globali che esse utilizzano, come si è visto, deve essere riservata al TSR con la funzione di libreria `keep()`. Se il linguaggio utilizzato per scrivere il TSR fosse l'assembler sarebbe sufficiente definire il segmento dati all'inizio del codice appositamente per le routine residenti; dal momento che l'assemblatore mantiene, nella traduzione in linguaggio macchina, le posizioni dei segmenti definite nel sorgente, si avrebbe la garanzia di strutturare il TSR come desiderato (vedere figura 14).

Il compilatore C ha un comportamento differente, in quanto genera il codice oggetto da sorgenti scritti in linguaggio di alto livello e consente di specificare il modello di memoria, ma non di definire i segmenti del codice: questi sono generati, in base alla struttura dello startup module (vedere pag. 105) e a criteri di ottimizzazione, dal compilatore medesimo, senza possibilità di controllo da parte del programmatore. Può dunque accadere che pur definendo le variabili globali[267] in testa al sorgente esse siano allocate dal compilatore nella parte finale del codice; perciò se le funzioni residenti sono definite prima di quelle dedicate all'installazione e di `main()`, la struttura del TSR diventa quella illustrata in figura 15.

[267] Le argomentazioni esposte sono valide per tutte le variabili globali utilizzate dalle routine residenti: non solo, dunque, quelle gestite in comune con le routine transienti.

Come si vede, la porzione di codice non necessaria alle routine residenti è quella centrale: ciò implica che deve essere allocata una quantità di RAM sufficiente a contenere tutto il codice, determinando gli sprechi di cui si è detto. In base ad un calcolo approssimativo, il codice di un programma ha un ingombro pari alla sua dimensione in byte incrementata di 256 (la dimensione del PSP; pag. 324). Va tenuto presente che per i file `.EXE` occorre sottrarre la dimensione dello header (Relocation Table) creato dal linker, in quanto esso non permane in memoria dopo il caricamento del programma. Inoltre, tra le informazioni contenute nello header vi è la quantità di memoria minima necessaria al programma, oltre al proprio ingombro, per essere caricato ed eseguito: un TSR può pertanto leggere questo dato nel proprio header per riservarsi tutta la RAM che gli è indispensabile. Ciò non significa, però, eliminare gli sprechi, dal momento che tale quantità include, ovviamente, anche la memoria necessaria alle parti di codice attive esclusivamente durante la fase di installazione: essa risponde soprattutto a criteri di sicurezza, a scapito dell'efficienza. Ecco come utilizzare lo header in questo genere di calcoli:

Fig. 15: La struttura di TSR generata dal compilatore C.

```
/********************

    BARNINGA_Z! - 1991

    RESMEM.C - resmemparas()

    unsigned cdecl resmemparas(char *progname);
    char *progname; puntatore al nome del programma
    Restituisce: il numero di paragrafi sicuramente sufficienti
                     al programma per restare residente in memoria
                 -1 in caso di errore

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx resmem.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma warn -pia

#include <stdio.h>

#define ERRCODE  -1                           // valore restituito in caso di errore
#define PAGEDIM  32                    // dimen. (in par.) di una pagina (512 bytes)
#define PSPDIM   16                               // dimensione in paragrafi del PSP
#define PARADIM  16                           // dimensione in bytes di un paragrafo

struct HEADINFO {
    unsigned signature;
    unsigned remainder;
    unsigned filepages;
    unsigned relocnum;
    unsigned headparas;
    unsigned minalloc;
};
```

```
unsigned cdecl resmemparas(char *progname)
{
    FILE *in;
    struct HEADINFO hdr;
    unsigned ret = ERRCODE;

    if(in = fopen(progname,"rb")) {
        if(fread(&hdr,sizeof(hdr),1,in) == sizeof(hdr))
            ret = ((hdr.remainder) ? hdr.filepages-1 : hdr.filepages)*PAGEDIM
                   +hdr.remainder/PARADIM+1
                   +hdr.minalloc
                   -hdr.headparas
                   +PSPDIM;
        fclose(in);
    }
    return(ret);
}
```

La funzione `resmemparas()` legge i primi 12 byte del programma in una struttura (appositamente definita) i cui campi interpretano le informazioni contenute in questa parte dello header. L'algoritmo calcola la dimensione in paragrafi del file e vi somma `1` in arrotondamento per eccesso. Al risultato ottenuto somma il numero minimo di paragrafi necessario al funzionamento del programma e la dimensione, ancora espressa in paragrafi, del PSP. Infine sottrae il numero di paragrafi componenti lo header. Dal momento che `resmemparas()` deve necessariamente accedere al file sul disco, è opportuno che essa sia tra le prime funzioni invocate, onde diminuire la probabilità che venga aperto lo sportello del drive prima che essa possa svolgere con successo il proprio compito. Il parametro `progname` può validamente essere `argv[0]` di `main()` (vedere pag. 105).

La sicurezza può poi essere salvaguardata, ancora sacrificando l'efficienza, passando alla funzione `keep()` la dimensione dell'area di memoria che il DOS ha effettivamente riservato al programma. Tale informazione è reperibile nel Memory Control Block (vedere pag. e seguenti) del programma stesso, il cui indirizzo di segmento è uguale a quello del PSP, decrementato di uno[268]:

```
    ....
    resparas = *((unsigned far *)MK_FP(_psp-1,0x03));
    ....
```

L'indirizzo di segmento del PSP è calcolato dallo startup code del C e da esso memorizzato in `_psp`, `unsigned int` globale dichiarata in `DOS.H`; `3` è l'offset, nel MCB, della word che esprime la dimensione dell'area di memoria; la macro `MK_FP()`, definita ancora in `DOS.H` (pag. 24), restituisce pertanto l'indirizzo `far` di tale word, gestita come `unsigned int` dal compilatore grazie all'operazione di cast. Alla variabile `resparas` è assegnata l'indirezione di detto indirizzo, cioè, in ultima analisi, la quantità di RAM da allocare permanentemente al programma. Si noti che è lecito passare l'espressione a `keep()` direttamente:

```
    ....
    keep(errlevel,*((unsigned far *)MK_FP(_psp-1,0x03)));
    ....
```

Questo approccio può condurre ad un pessimo utilizzo della RAM. Infatti, i due byte che si trovano all'offset `0Ch` nello header dei `.EXE` (e dunque seguono immediatamente quelli letti nel campo `minalloc` dalla `resmemparas()`) esprimono la quantità di memoria (in paragrafi) desiderata dal programma oltre al proprio ingombro, la quale è, di solito, maggiore di quella effettivamente

---

[268] Questo algoritmo, a differenza del precedente, è applicabile anche ai `.COM`.

necessaria[269]. Per quel che riguarda i .COM, invece, a causa dell'assenza di header, il DOS non è in grado di conoscere a priori la quantità di memoria necessaria al programma e pertanto ne riserva ad esso quanta più è possibile; non è infrequente che il MCB del programma sia così l'ultimo presente nella RAM e controlli un'area comprendente tutta la memoria disponibile.

Ottimizzazione dell'impiego della RAM

A pagina abbiamo presentato uno stratagemma utile per la gestione dei dati globali: esso consente di riservare loro RAM a locazioni accessibili mediante offset relativi a CS e non a DS. Si è anche precisato che i gestori di interrupt installati da un programma (magari proprio un TSR) possono in tal modo accedere facilmente ai dati globali di loro interesse, ed in particolare ai vettori dei gestori originali. Il fatto che tale artificio si traduca nel definire una o più funzioni fittizie (contenitori di dati) offre uno spunto interessante anche ai fini dell'ottimizzazione della quantità di RAM da allocare ai TSR.

Il compilatore C, all'interno di ogni segmento generato a partire dal sorgente, non altera la posizione degli elementi che lo compongono: determinando con cura la posizione della funzione fittizia si ha la possibilità di strutturare il codice del TSR come desiderato. Rivediamo in questa luce l'esempio di pag. :

```
#define integer1    (*((int *)Jolly))
#define integer2    (*(((int *)Jolly)+1))
#define new_handler ((void (interrupt *)())(*(((long *)Jolly)+1)))

#define ASM_handler Jolly+4

void Jolly(void);

void interrupt new_handler(void)
{
    ....
    asm {
        pushf;
        call dword ptr ASM_handler;
```

[269] Questo campo dello header, spesso chiamato maxalloc, esiste, presumibilmente, in previsione di future versioni multitask/multiutente del DOS. Attualmente esso è poco utilizzato: il linker gli assegna per default il valore (fittizio) di 0FFFFh, cioè 65535 paragrafi. Se si desidera che l'area di RAM allocata al programma dal DOS sia limitata all'indispensabile è sufficiente scrivere il valore 1 nel campo maxalloc. Il Microsoft Linker dispone, allo scopo, dell'opzione /CPARMAXALLOC: o /CP:. Il comando

```
link my_prog.obj /CP:1
```

raggiunge lo scopo. In TURBO C occorre agire dall'interno del codice, limitando al valore desiderato la dimensione in byte dello heap (pag. 109) attraverso la variabile globale _heaplen:

```
    ....
    _heaplen = 8000;                                        // e' un valore di esempio
    ....
```

Inoltre maxalloc può essere modificato in qualunque file .EXE con il programma EXEMOD.EXE:

```
exemod my_prog.exe /MAX 1
```

consente di ottenere il risultato voluto. Vi è infine, per gli amanti del brivido, la possibilità di intervenire sul file eseguibile con un programma in grado di effettuare l'editing esadecimale del codice compilato.

```
    }
    ....
}

....

void Jolly(void)
{
    asm dw 0;
    asm dw 1;
    asm dd 0;
}
```

Si noti che la `Jolly()` è dichiarata prima del codice appartenente al gestore di interrupt, ma definita dopo di esso. L'accorgimento di definire la funzione fittizia dopo tutte le routine residenti e prima di tutte quelle transienti consente di calcolare facilmente quanta RAM allocare al TSR: essa è data dalla differenza tra l'indirizzo di segmento della stessa `Jolly()` e l'indirizzo di segmento del PSP, più la somma, divisa per 16 (per ricondurre il tutto a numero di paragrafi), dell'offset della `Jolly()` e l'ingombro dei dati in essa definiti, più uno, a scopo di arrotondamento per eccesso. Nell'esempio riportato sopra si avrebbe:

```
    unsigned resparas;
    ....
    resparas = FP_SEG(Jolly)-_psp+1+
                              (FP_OFF(Jolly)+2*sizeof(int)+sizeof(void far *))/16;
    ....
```

E' però possibile semplificare il calcolo utilizzando il nome della prima funzione dichiarata dopo la `Jolly()` come puntatore al confine della RAM da allocare: si osservi l'esempio che segue:

```
....
void Jolly(void)
{
    ....
}

void DopoJolly()
{
    ....
}
....
```

Il valore di `resmemparas` può essere ottenuto così:

```
    ....
    resmemparas = FP_SEG(DopoJolly)+PF_OFF(DopoJolly)/16+1-_psp;
    ....
```

La semplice dichiarazione del prototipo di `Jolly()` in testa al sorgente consente di referenziarla (tramite le macro) nelle routine residenti prima che essa sia definita. Per ragioni analoghe è necessario compilare utilizzando l'opzione `-Tm2` di `TCC` (o `BCC`), che forza l'assemblatore (`TASM`) ad effettuare due passi (loop) di compilazione per risolvere i *forward reference*, cioè i riferimenti a simboli non ancora definiti[270].

[270] La dichiarazione del prototipo di `Jolly()` e l'opzione `-Tm2` non sono necessari se la `Jolly()` è definita prima di tutte le funzioni residenti che ne utilizzano il nome come puntatore ai dati residenti: in tal caso è necessario calcolare il numero di paragrafi residenti basandosi sull'indirizzo della prima funzione non residente definita nel codice. La semplificazione così introdotta rende però impossibili alcune ottimizzazioni. Si pensi al caso in cui lo

Allocazione dinamica della RAM

Anche l'allocazione dinamica della memoria può essere fonte di guai; infatti il DOS non riconosce i blocchi allocati con le funzioni di libreria appartenenti al gruppo della `malloc()` o che, comunque, non utilizzano la tecnica dei MCB[271]: la spiacevole conseguenza è che il TSR perde il "possesso" di tali blocchi non appena terminata l'installazione[272] ed essi possono essere sovrascritti dal codice o dai dati di qualunque altro programma. Ne segue che è opportuno utilizzare, nei TSR, le funzioni di libreria[273] `allocmem()`, `setblock()` e `freemem()`, le quali, a differenza delle precedenti, fanno uso dei MCB e sono pertanto in grado di interagire con il DOS. Si sottolinea che `setblock()`, a differenza di `realloc()`, non è in grado di spostare il contenuto dell'area allocata quando la RAM libera disponibile in un unico blocco a partire dall'indirizzo attuale non è sufficiente a "coprire" tutto l'ampliamento richiesto.

Il massiccio uso di tecniche di allocazione dinamica della memoria in un TSR può creare comunque problemi al DOS (in particolare è pericolosa l'alternanza di blocchi liberi e blocchi allocati nella catena dei MCB); ricorrere il più possibile ad array non è obbligatorio, ma può evitare problemi.

Con un po' di accortezza è comunque possibile, minimizzando pericoli e sprechi di RAM, allocare buffers che il TSR utilizza anche dopo il termine della fase di installazione: si osservi il listato che segue.

```
....
void Jolly(void);
....
void InstallTSRbuff(unsigned resparas,unsigned bufparas,int code)
{
    asm {
        mov ah,0x4A;
        mov bx,resparas;
        mov es,_psp;
        int 0x21;
        mov ah,0x48;
        mov bx,bufparas;
        int 0x21;
        mov Jolly,ax;
        mov ah,0x31;
        mov al,code;
        mov dx,resparas;
        int 0x21;
    }
}
....
void Jolly(void)
```

---

spazio necessario ai dati residenti non sia noto al momento della compilazione, ma sia determinato in fase di installazione: la `Jolly()` deve riservare tutto lo spazio necessario nel caso di maggiore "ingombro" possibile dei dati. Solo se essa è definita dopo tutte le funzioni residenti è possibile limitare la RAM allocata al minimo indispensabile.

[271] Tra queste ultime, per citarne una forse tra le più utilizzate, la `fopen()`. Si veda pag. per alcuni particolari interessanti.

[272] Analoghe considerazioni valgono per blocchi di memoria allocati durante le fasi di popup del TSR: essi non vengono protetti quando il controllo è ceduto all'applicazione interrotta.

[273] A condizione che sia la parte transiente ad utilizzarle. E' bene che la parte residente non faccia uso di funzioni di libreria, come chiariremo tra poco (pag. 289).

```
{
    ....
}
....
```

La `InstallTSRbuff()` esegue diverse operazioni: in primo luogo, mediante il servizio 4Ah dell'int 21h riduce la RAM allocata al TSR alle dimensioni ad esso strettamente necessarie (per un esempio di calcolo di `resparas` si veda pag.). Tramite la funzione 48h dell'int 21h essa alloca poi la RAM necessaria al buffer (si noti che, per semplicità, la funzione non include il codice necessario a rilevare il verificarsi di eventuali condizioni di errore). Infine, dopo avere salvato nello spazio riservato dalla `Jolly()` l'indirizzo di segmento del buffer, la `InstallTSRbuff()` termina il programma e lo rende residente (int 21h, servizio 31h).

In sostanza, la `InstallTSRbuff()` forza il DOS ad aggiornare opportunamente la lista dei MCB: il risultato è la creazione di un nuovo MCB, quello relativo al buffer, appartenente al TSR (o meglio al suo PSP) e situato esattamente dopo la `Jolly()`; considerando, oltre a ciò, che un MCB occupa 16 byte, risultano di immediata comprensione alcune caratteristiche della funzione presentata. Innanzi tutto essa è collocata prima della `Jolly()`, pur essendo una tipica routine transiente: questa precauzione sopprime il rischio che il DOS, modificando il contenuto della RAM, ne alteri il codice. In secondo luogo `InstallTSRbuff()` rende residente il programma: ancora per il motivo appena accennato è opportuno che la modifica dei MCB e l'allocazione del buffer siano le ultime azioni del programma nella fase di installazione. Proponiamo una versione di `InstallTSRbuff()` interamente in linguaggio C:

```
....
void Jolly(void);
....
void InstallTSRbuff(unsigned resparas,unsigned bufparas,int code)
{
    setblock(_psp,resparas);
    allocmem(bufparas,(unsigned *)(*((unsigned *)Jolly)));
    keep(code,resparas);
}
```

A pag. si dirà di alcune limitazioni riguardanti l'uso di funzioni di libreria nelle routine residenti: precisiamo che tali problemi non riguardano la `InstallTSRbuff()`, in quanto essa, come si è detto, viene eseguita solamente durante l'installazione del TSR. Va inoltre sottolineato che essa non è perfettamente equivalente alla sua omonima basata sullo inline assembly: infatti `setblock()`, `allocmem()` e `keep()` non si limitano ad invocare, rispettivamente, i servizi 4Ah, 48h e 31h dell'int 21h. In particolare la `keep()` si preoccupa di ripristinare alcuni vettori di interrupt: per i dettagli si veda pagina . L'operazione di cast

```
(unsigned *)(*((unsigned *)Jolly))
```

si spiega come segue: `Jolly` è il puntatore alla (nome della) funzione fittizia per la gestione dei dati globali e viene forzato a puntatore ad intero senza segno. L'indirezione di questo è dunque un `unsigned int`, il quale è a sua volta forzato a puntatore ad intero senza segno: `unsigned *` è, appunto, il parametro richiesto da `allocmem()`, che vi copia l'indirizzo di segmento dell'area allocata (il valore del registro `AX` dopo la chiamata all'int 21h).

I TSR e la memoria EMS

I programmi TSR possono liberamente gestire la memoria EMS (pag. 202) utilizzando i servizi dell'int 67h; vale tuttavia il principio generale per cui un TSR non deve mai scompaginare il lavoro del

processo interrotto. Va tenuto presente che un TSR (in particolare se di tipo attivo) può interromepere l'esecuzione degli altri programmi in modo asincrono (cioè in qualunque momento) senza che questi abbiano la possibilità di salvare il loro mapping context. Prima di utilizzare la memoria EMS un TSR deve quindi necessariamente provvedere "di persona" al salvataggio del mapping context attuale (che è, ovviamente, quello del processo interrotto) e solo successivamente può attivare il proprio. Prima di restituire il controllo al sistema, il TSR deve effettuare l'operazione opposta: salvare il proprio mapping context e riattivare quello del programma interrotto.

Le quattro operazioni suddette possono, in realtà, essere ridotte a due grazie alla subfunzione 02h del servizio 4Eh dell'int 67h (pag. 219), il quale è in grado di effettuare un'operazione di salvataggio del mapping context attuale e contemporaneamente attivare un secondo mapping context, salvato in precedenza.

Ecco un esempio, nell'ipotesi che `TSRmapContext()` e `InterruptedMapContext()` siano le due funzioni jolly (pag. 280) usate per memorizzare il mapping context del TSR e, rispettivamente, del processo interrotto:

```
    ....                                              // routine di ingresso del TSR
    asm push ds;
    asm mov si,seg TSRmapContext;                // DS:SI punta al buffer contenente
    asm mov ds,si;                                     // il mapping context del TSR
    asm mov si,offset TSRmapContext;           // questo mapping context e' attivato
    asm mov di,seg InterruptedMapContext;            // ES:DI punta al buffer in cui
    asm mov es,di;                           // deve essere salvato il mapping context
    asm mov di,offset InterruptedMapContext;              // del processo interrotto
    asm mov ax,4E02;
    asm int 067h;
    asm pop ds;
    asm cmp ah,0;
    asm jne ERROR;
    ....                                                       // operazioni del TSR
    asm push ds;
    asm mov si,seg InterruptedMapContext;      // DS:SI punta al mapping context del
    asm mov ds,si;                       // processo interrotto, salvato in precedenza
    asm mov si,offset InterruptedMapContext;                 // e ora da riattivare
    asm mov di,seg TSRMapContext;              // ES:DI punta al buffer in cui deve
    asm mov es,di;                               // essere salvato l'attuale mapping
    asm mov di,offset TSRMapContext;                                // context del TSR
    asm mov ax,4E02;
    asm int 067h;
    asm pop ds;
    asm cmp ah,0;
    asm je EXIT_TSR;
ERROR:
    ....                                                  // gestione errori int 67h
EXIT_TSR:
    ....                                             // operazioni di uscita dal TSR
```

Come si vede, l'implementazione non presenta difficoltà particolari. Sono necessari 2 buffers, uno dedicato al mapping context del TSR ed uno dedicato a quello del processo interrotto. In ingresso al TSR viene caricato in `DS:SI` l'indirizzo del buffer contenente il mapping context del TSR e in `ES:DI` quello del buffer dedicato al programma interrotto; in tal modo la chiamata all'int 67h determina il corretto salvataggio del mapping context attuale e il caricamento (ed attivazione) di quello del TSR. In uscita dal TSR l'operazione effettuata è identica, ma sono scambiati gli indirizzi dei due buffers (`ES:DI` per il TSR e `DS:SI` per il processo interrotto): l'int 67h salva così il mapping context del TSR nel buffer ad esso dedicato e ripristina, come necessario, quello del processo interrotto.

Rilasciare l'environment del TSR

Ancora con riferimento alla gestione della memoria, vogliamo dedicare qualche attenzione all'environment, cioè all'insieme delle variabili d'ambiente che il DOS mette a disposizione[274] di tutti i programmi al momento dell'esecuzione. Se le routine residenti del TSR non fanno uso dell'environment, questo può essere rilasciato. In altre parole è possibile disallocare la RAM ad esso riservata e renderla nuovamente disponibile per altri usi (ad esempio per ospitare le variabili d'ambiente di un programma lanciato successivamente; si tenga presente che lo spazio da esse occupato spesso non raggiunge il centinaio di byte). L'indirizzo di segmento dell'environment è la word all'offset `2Ch` nel PSP del programma; il Memory Control Block (pag.) relativo è costituito dai 16 byte immediatamente precedenti tale indirizzo. Un metodo (rozzo, tuttavia efficace) di rilasciare l'environment consiste nell'azzerare la word del MCB che contiene l'indirizzo del PSP del programma proprietario. Un sistema alternativo è quello sul quale si basa la funzione presentata di seguito:

```
/********************

    BARNINGA_Z! - 1990

    RELENV.C - releaseEnv()

    int cdecl releaseEnv(void);

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx relenv.C

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

int cdecl releaseEnv(void)
{
    asm {
        mov ax,0x6200;
        int 0x21;
        mov es,bx;
        mov bx,0x2C;
        mov es,es:[bx];
        mov ax,0x49;
        int 0x21;
        mov ah,0;
    }
    return(_AX);
}
```

La `releaseEnv()` si serve del servizio 62h dell'int 21h per conoscere l'indirizzo di segmento del PSP (vedere pag. 324) e, mediante l'offset del puntatore all'environment, carica con l'indirizzo di quest'ultimo il registro `ES`, liberando la RAM allocata con il servizio 49h dell'int 21h (vedere pag. 190). La funzione restituisce `0` se l'environment è rilasciato regolarmente; un valore diverso da `0` in caso di errore. Della `releaseEnv()` presentiamo anche una versione interamente in C.

```
/********************

    BARNINGA_Z! - 1990
```

---

[274] In realtà il DOS mette a disposizione di ogni programma una copia dell'environment originale (quello generato dall'interprete dei comandi).

```
    RELENVC.C - releaseEnvC()

    int cdecl releaseEnv(void);

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx relenvc.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

int cdecl releaseEnvC(void)
{
    return(freemem(*((unsigned far *)MK_FP(_psp,0x2C))));
}
```

La variabile `_psp` è definita in `DOS.H` e contiene l'indirizzo di segmento del PSP; il cast forza a puntatore `far` ad intero senza segno il valore restituito dalla macro `MK_FP()` (pag. 24), la cui indirezione, passata a `freemem()` come parametro, è l'indirizzo di segmento dell'environment.

Lo startup code (vedere pag. 105) del compilatore Borland, infine, mette a disposizione una comoda scorciatoia, peraltro non documentata, per conoscere l'indirizzo dell'environment: si tratta della variabile globale `_envseg`, dichiarata proprio nello startup code, che può essere utilizzata all'interno dei normali programmi, previa dichiarazione `extern`. La `releaseEnv()` potrebbe pertanto diventare:

```
#include <dos.h>

extern unsigned _envseg;

int cdecl releaseEnv2(void)
{
    return(freemem(_envseg));
}
```

Si noti che la dichiarazione della variabile `_envseg` potrebbe trovarsi anche all'interno del codice della `releaseEnv2()`, in quanto, ripetiamo, `_envseg` è definita globalmente nello startup code: si tratta semplicemente, qui, di deciderne l'ambito di visibilità.

Due parole sullo stack

Si è detto (pag.) che un TSR si attiva nel contesto del programma che in quel momento è eseguito e ne condivide pertanto le risorse, tra le quali lo stack, che, a causa delle sue particolari modalità di gestione[275], richiede, da parte delle routine residenti dei TSR, alcune precauzioni indispensabili per un buon funzionamento del sistema.

La prima, ovvia, è che i gestori di interrupt, e le routine che essi eventualmente invochino, devono utilizzare in modo opportuno i registri della CPU dedicati alla gestione dello stack (vedere pag.); essi devono inoltre estrarre da questo i dati che vi abbiano spinto in precedenza, prima di restituire il controllo alla routine chiamante. Se nelle funzioni residenti non compaiono linee di codice scritte in inline

[275] Lo stack, tanto vale ripeterlo ancora una volta, è sempre gestito secondo la modalità LIFO (Last In, First Out): l'ultimo dato o, per meglio dire, l'ultima word spinta su di esso è la prima a esserne estratta. Di qui il nome, che significa "pila".

assembly il compilatore provvede da sé ad assicurare che tutto sia gestito nel migliore dei modi[276]; in caso contrario spetta al programmatore l'onere di valutare con estrema attenzione le conseguenze dell'interazione tra codice C e assembly.

La seconda precauzione, forse meno ovvia ma altrettanto fondamentale, è che le funzioni residenti non possono permettersi di fare un uso eccessivamente pesante dello stack. Esso deve essere comunque considerato una risorsa limitata, in quanto non è possibile sapere a priori quanto spazio libero si trova nello stack del programma interrotto al momento dell'attivazione del TSR: se questo utilizza più stack di quanto il programma interrotto ne abbia disponibile, la conseguenza è, normalmente, il blocco del sistema. Per evitare un eccessivo ricorso allo stack può essere sufficiente ridurre al minimo il numero di variabili automatiche definite nelle routine residenti ed utilizzare invece variabili globali, gestite come descritto poco sopra: tale metodo è applicabile a tutte le variabili globali utilizzate dal codice residente.

E' del resto possibile (per non dire meglio!) utilizzare una funzione fittizia (pag. 172) per riservare spazio ad uno stack locale alla porzione residente del TSR. Vediamo un esempio:

```
#pragma  option -k-

#define  STACKSIZE    128                                    // 128 bytes di stack
....
void oldSS(void)                                // spazio per salvare il valore di SS
{
    asm db 0;                                  // 1 byte basta (c'e' l'opcode di RET)
}

void oldSP(void)                                // spazio per salvare il valore di SP
{
    asm db 0;                                  // 1 byte basta (c'e' l'opcode di RET)
}

void TSRstack(void)                                           // stack locale del TSR
{
    asm db STACKSIZE dup(0);
}

....
void far new1Ch(void)                                                // gestore timer
{
    asm mov word ptr oldSS,ss;
    asm mov word ptr oldSP,sp;
    asm mov sp,seg TSRstack;             // usa SP per non modificare altri registri
    asm mov ss,sp;                       // SS non puo' essere caricato direttamente
    asm mov sp,offset TSRstack;
    asm add sp,STACKSIZE;                      // SS:SP punta alla FINE di TSRstack()
    ....
    asm mov sp,word ptr oldSS;
    asm mov ss,sp;
    asm mov sp,word ptr oldSP;                              // SS:SP e' ripristinato
    asm jmp dword ptr old1Ch;
}
```

Nel listato, ridotto all'osso, compaiono 3 funzioni fittizie: `oldSS()` e `oldSP()` riservano spazio alle due word occupate da `SS` e da `SP`[277], mentre `TSRstack()` è lo stack del TSR. La coppia

[276] O, almeno, in modo tale che funzioni.

[277] I due registri, infatti, devono essere salvati senza usare lo stack, perché prima del loro salvataggio è ancora utilizzato quello del processo interrotto: in questo caso, l'obiettivo principale non è tanto quello di evitare l'uso di risorse che non appartengono al TSR, ma quello di ripristinare i valori originali in uscita dalla `new1Ch()`. Infatti, se `SS` e `SP` venissero salvati sullo stack, dovrebbero essere estratti dal medesimo mediante l'istruzione `POP`, la quale, però, non farebbe altro che prelevare una word dall'indirizzo espresso proprio dalla coppia `SS:SP`: essendo

SS:SP è salvata in oldSS() e oldSP() e caricata con l'indirizzo (seg:off) di TSRstack(); dal momento che la gestione dello stack avviene sempre "a ritroso", cioè a partire dagli indirizzi superiori verso quelli inferiori, il valore iniziale di SS:SP deve puntare all'ultimo byte occupato dalla funzione: per tale motivo ad SP è sommata la costante manifesta STACKSIZE (utilizzata anche per stabilire il numero di byte generati dalla direttiva assembly DB). Da questo punto in poi tutte le istruzioni che modificano lo stack esplicitamente (PUSH, POP, etc.) o implicitamente (CALL, etc.) utilizzano in modo trasparente lo spazio riservato da TSRstack(). In uscita da new1Ch() è necessario ripristinare i valori di SS ed SP prima della IRET (o prima di concatenare il gestore originale, come nell'esempio).

Ancora una volta, è necessario prestare attenzione ai comportamenti nascosti del compilatore: se il gestore di interrupt referenzia SI o DI, questi vengono salvati dal compilatore, in ingresso alla funzione, sullo stack (del processo interrotto) e devono pertanto essere estratti dallo stesso prima di attivare quello locale al TSR. Il codice di new1Ch() risulta allora leggermente più complesso:

```
void far new1Ch(void)                                             // gestore timer
{
    asm pop di;                                                   // PUSHed da BCC
    asm pop si;                                                   // PUSHed da BCC
    asm mov word ptr oldSS,ss;
    asm mov word ptr oldSP,sp;
    asm mov sp,seg TSRstack;              // usa SP per non modificare altri registri
    asm mov ss,sp;                        // SS non puo' essere caricato direttamente
    asm mov sp,offset TSRstack;
    asm add sp,STACKSIZE;                      // SS:SP punta alla FINE di TSRstack()
    ....
    asm mov sp,word ptr oldSS;
    asm mov ss,sp;
    asm mov sp,word ptr oldSP;                                // SS:SP e' ripristinato
    asm jmp dword ptr old1Ch;
}
```

In alternativa, se il gestore non deve modificarne i valori, SI e DI possono essere estratti dallo stack del processo interrotto prima di restituire ad esso il controllo (o prima di concatenare il gestore originale):

```
void far new1Ch(void)                                             // gestore timer
{
    asm mov word ptr oldSS,ss;
    asm mov word ptr oldSP,sp;
    asm mov sp,seg TSRstack;              // usa SP per non modificare altri registri
    asm mov ss,sp;                        // SS non puo' essere caricato direttamente
    asm mov sp,offset TSRstack;
    asm add sp,STACKSIZE;                      // SS:SP punta alla FINE di TSRstack()
    ....
    asm mov sp,word ptr oldSS;
    asm mov ss,sp;
    asm mov sp,word ptr oldSP;                                // SS:SP e' ripristinato
    asm pop di;                                                   // PUSHed da BCC
    asm pop si;                                                   // PUSHed da BCC
    asm jmp dword ptr old1Ch;
}
```

Va infine osservato, per completezza, che quello implementato nell'esempio non è un vero e proprio stack, ma piuttosto un'area riservata al TSR in modo statico: ogniqualvolta venga eseguita new1Ch() i puntatori all'area (SS:SP) sono impostati al medesimo valore (TSRstack+STACKSIZE);

---

stata quest'ultima modificata per puntare allo stack del TSR, l'estrazione avverrebbe ad un indirizzo diverso da quello al quale si trovano i due valori salvati.

una ricorsione distruggerebbe i dati dell'istanza in corso[278] (analogamente a quanto accade con gli stack interni DOS: pag. 294). Si tratta però di un'implementazione semplice ed efficace; inoltre è possibile definire uno stack privato per ogni funzione residente che ne necessiti.

Utilizzo delle funzioni di libreria

I TSR sono soggetti ad alcune limitazioni anche per quanto concerne l'uso delle funzioni di libreria; va però precisato che ciò vale esclusivamente per le routine residenti, mentre quelle transienti fruiscono di una piena libertà di comportamento.

Per generare il file eseguibile, i moduli oggetto prodotti dal compilatore devono essere consolidati con quello contenente il codice di startup (pag. 105) e con le librerie: tale operazione è svolta dal linker, il quale dapprima accoda i moduli oggetto al modulo di startup e solo al termine di questa operazione estrae dalle librerie i moduli contenenti le funzioni utilizzate dal programma e li accoda al file in costruzione.

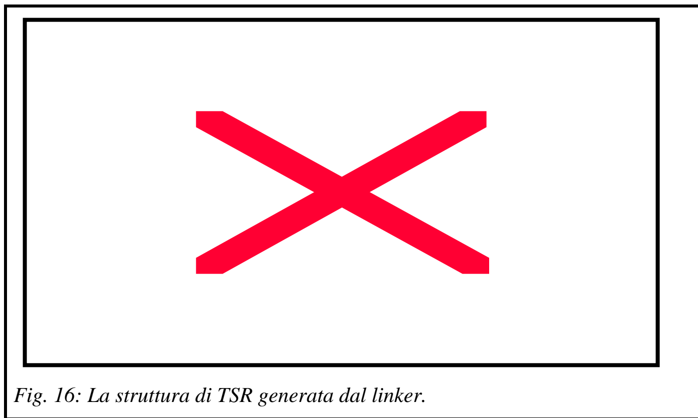

Fig. 16: La struttura di TSR generata dal linker.

La struttura del programma eseguibile risulta perciò analoga a quella in figura 16: come si vede, se le routine transienti utilizzano funzioni di libreria, l'occupazione della RAM non può essere ottimizzata, e ciò neppure nel caso in cui i dati globali siano gestiti con lo stratagemma descritto nelle pagine precedenti, in quanto anche il codice di tali funzioni deve essere residente. Il problema potrebbe essere aggirato estraendo dalle librerie i moduli relativi alle funzioni necessarie ed effettuando esplicitamente il linking dei diversi moduli che compongono il TSR, con l'accortezza di specificare per primo il nome del modulo di startup e per ultimo quello del modulo risultante dalla compilazione del sorgente[279], ma è intuibile che l'applicazione di questa tecnica richiede

---

[278] Non solo una ricorsione: è facile immaginare il caso in cui l'int 1Ch, chiamato in modo asincrono dall'int 08h (interrupt hardware ad elevata priorità) interrompa un'altra routine residente: se questa utilizza il medesimo stack locale, la frittata è fatta.

[279] Supponiamo di avere scritto il sorgente di un TSR, che per comodità battezziamo `MY_TSR.C`, le routine residenti del quale utilizzano la funzione di libreria `int86()`. Con l'opzione `"-c"` del compilatore si disattiva l'invocazione automatica del linker da parte di `BCC.EXE`; il seguente comando produce pertanto il solo file oggetto `MY_TSR.OBJ`, per il modello di memoria small (default):

```
bcc -c my_tsr
```

Dal momento che la `int86()` invoca, a sua volta, la funzione `__IOerror()` occorre estrarre entrambi i moduli dalla libreria `CS.LIB`:

```
tlib cs.lib * int86 ioerror
```

Si ottengono così `INT86.OBJ` e `IOERROR.OBJ`; poiché il modulo di startup per il modello small è `C0S.OBJ`, possiamo effettuare il linking con il comando:

```
tlink c0s int86 ioerror my_tsr,my_tsr,my_tsr,cs
```

una buona conoscenza della struttura delle librerie. Vi è, inoltre, un problema legato allo startup code: esso definisce, nel segmento dati, variabili globali utilizzate, in determinate circostanze, da alcune funzioni di libreria. Se si ottimizza la dimensione dell'area di RAM allocata al TSR in modo tale da escluderne il data segment, lo spazio occupato da tali variabili può essere utilizzato dai programmi lanciati successivamente, con tutti i rischi che ciò comporta[280].

Ma c'è di peggio. Se, da una parte, è ovvio che, per ogni funzione chiamata nel sorgente, il linker importi nell'eseguibile il modulo oggetto che la implementa, è assai meno evidente, ma purtroppo altrettanto vero, che qualcosa di analogo possa avvenire anche in corrispondenza di istruzioni che, apparentemente, nulla hanno a che fare con chiamate a funzione: è il caso, ad esempio, delle operazioni aritmetiche.

Consideriamo la funzione `opeIntegral16()`:

```
void opeIntegral16(void)
{
    int a, b, c;

    a = 2;
    b = 1818;
    c = a + b;
    c = a - b;
    c = a * b;
    c = a / b;
    c = a % b;
}
```

Come si può facilmente vedere, essa non richiama alcuna funzione di libreria: vengono definite tre variabili, sulle quali sono effettuate normali operazioni aritmetiche, applicando gli operatori utilizzabili tra dati tipo integral (pag. 12). Vediamo, ora, la traduzione in Assembler del codice C effettuata dal compilatore (opzione `-S`):

```
_opeIntegral16 proc near
    push bp
    mov bp,sp
    sub sp,6

    mov word ptr [bp-2],2
    mov word ptr [bp-4],1818

    mov ax,word ptr [bp-2]
    add ax,word ptr [bp-4]
    mov word ptr [bp-6],ax

    mov ax,word ptr [bp-2]
    sub ax,word ptr [bp-4]
    mov word ptr [bp-6],ax

    mov ax,word ptr [bp-2]
```

Il linker consolida, nell'ordine, il modulo di startup, quelli relativi alle funzioni di libreria e il codice di `MY_TSR`, producendo il file eseguibile `MY_TSR.EXE` e il map file (file contenente l'elenco degli indirizzi e delle dimensioni dei simboli pubblici che compongono il codice) `MY_TSR.MAP`: le funzioni esterne ai moduli oggetto sono ricercate nella libreria `CS.LIB`. Si noti che è indispensabile specificare al linker tutte le librerie da utilizzare.

[280] Il contenuto delle locazioni di memoria originariamente riservate a quelle variabili è, in pratica, casuale. Inoltre, se una funzione di libreria invocata dalle routine residenti tentasse di modificarlo, rischierebbe di "obliterare" il codice e/o i dati di altre applicazioni. Tra le variabili globali definite dallo startup code vi è, ad esempio, `errno` (vedere pag. 499).

```
    imul word ptr [bp-4]
    mov word ptr [bp-6],ax

    mov ax,word ptr [bp-2]
    cwd
    idiv word ptr [bp-4]
    mov word ptr [bp-6],ax

    mov ax,word ptr [bp-2]
    cwd
    idiv word ptr [bp-4]
    mov word ptr [bp-6],dx

    mov sp,bp
    pop bp
    ret
_opeIntegral16 endp
```

Tutte le operazioni sono implementate ricorrendo a semplici istruzioni Assembler (`ADD`, `SUB`, `IMUL`, `IDIV`): non è effettuata alcuna chiamata a funzione. Va però osservato che tutte le operazioni sono definite tra dati di tipo `int`, i quali (nella consueta assunzione che si compongano di 16 bit) possono essere facilmente gestiti nei registri a 16 bit del microprocessore.

Vediamo ora cosa accade se le medesime operazioni sono definite su dati a 32 bit: il sorgente C della funzione `opeIntegral32()` è identico al precedente, eccezion fatta per la dichiarazione delle variabili, questa volta di tipo `long`:

```
void opeIntegral32(void)
{
    long a, b, c;

    a = 2;
    b = 1818;
    c = a + b;
    c = a - b;
    c = a * b;
    c = a / b;
    c = a % b;
}
```

Qualcosa di insolito, però, compare nel corrispondente listato Assembler:

```
_opeIntegral32 proc near
    push bp
    mov bp,sp
    sub sp,12

    mov word ptr [bp-2],0
    mov word ptr [bp-4],2
    mov word ptr [bp-6],0
    mov word ptr [bp-8],1818

    mov ax,word ptr [bp-2]
    mov dx,word ptr [bp-4]
    add dx,word ptr [bp-8]
    adc ax,word ptr [bp-6]
    mov word ptr [bp-10],ax
    mov word ptr [bp-12],dx

    mov ax,word ptr [bp-2]
    mov dx,word ptr [bp-4]
    sub dx,word ptr [bp-8]
```

```
    sbb ax,word ptr [bp-6]
    mov word ptr [bp-10],ax
    mov word ptr [bp-12],dx

    mov cx,word ptr [bp-2]
    mov bx,word ptr [bp-4]
    mov dx,word ptr [bp-6]
    mov ax,word ptr [bp-8]
    call near ptr N_LXMUL@
    mov word ptr [bp-10],dx
    mov word ptr [bp-12],ax

    push word ptr [bp-6]
    push word ptr [bp-8]
    push word ptr [bp-2]
    push word ptr [bp-4]
    call near ptr N_LDIV@
    mov word ptr [bp-10],dx
    mov word ptr [bp-12],ax

    push word ptr [bp-6]
    push word ptr [bp-8]
    push word ptr [bp-2]
    push word ptr [bp-4]
    call near ptr N_LMOD@
    mov word ptr [bp-10],dx
    mov word ptr [bp-12],ax

    mov sp,bp
    pop bp
    ret
_opeIntegral32 endp
```

Mentre addizione e sottrazione sono, ancora una volta, implementate direttamente via Assembler (`ADC`, `SBB`), per il calcolo di moltiplicazione, divisione e resto sono utilizzate routine specifiche i cui indirizzi sono memorizzati nei puntatori `N_LMUL@`, `N_LDIV@` e, rispettivamente, `N_LMOD@`[281]. Si tratta di routine di libreria che hanno lo scopo di applicare correttamente l'aritmetica su dati che il processore non è in grado di gestire nei propri registri.

Attenzione, dunque, anche a quelle situazioni in apparenza del tutto "innocenti": è sempre opportuno documentarsi in modo approfondito sulle caratteristiche del compilatore utilizzato; inoltre, uno sguardo ai sorgenti Assembler che esso genera specificando l'opzione `-S` è spesso illuminante.

Si consideri comunque che, spesso, la soluzione è a portata di mano: se la funzione `opeIntegral32()` è compilata con l'opzione `-3`, viene generato codice specifico per processori 80386 (e superiori). In tal modo è possibile sfruttarne i registri a 32 bit, rendendo del tutto inutile il ricorso alle routine aritmetiche di libreria. Infatti, il comando

```
bcc -S -3 opeint32.c
```

origina il seguente codice Assembler:

```
.386                                // forza l'assemblatore a generare codice 80386
_opeIntegral32 proc near
    push bp
    mov bp,sp
```

[281] Il prefisso `N_` indica che il frammento di codice è stato compilato per un modello di memoria "a codice piccolo": compilando con i modelli medium, large e huge i puntatori (`far`) sarebbero `F_LMUL@`, `F_LDIV@` e `F_LMOD@`.

```
    sub sp,12

    mov dword ptr [bp-4],large 2
    mov dword ptr [bp-8],large 1818

    mov eax,dword ptr [bp-4]
    add eax,dword ptr [bp-8]
    mov dword ptr [bp-12],eax

    mov eax,dword ptr [bp-4]
    sub eax,dword ptr [bp-8]
    mov dword ptr [bp-12],eax

    mov eax,dword ptr [bp-4]
    imul eax,dword ptr [bp-8]
    mov dword ptr [bp-12],eax

    mov eax,dword ptr [bp-4]
    cdq
    idiv dword ptr [bp-8]
    mov dword ptr [bp-12],eax

    mov eax,dword ptr [bp-4]
    cdq
    idiv dword ptr [bp-8]
    mov dword ptr [bp-12],edx

    leave
    ret
_opeIntegral32 endp
```

Il prezzo da pagare, in questo caso, è l'impossibilità di utilizzare il programma su macchine dotate di CPU di categoria inferiore al 80386.

Quanto affermato circa gli integral vale, a maggior ragione, con riferimento all'aritmetica a virgola mobile. Quando il sorgente definisce operazioni aritmetiche coinvolgenti dati di tipo `float`, `double` e `long double`, il compilatore genera per default il codice per il coprocessore matematico e richiede al linker il contemporaneo consolidamento delle routine di emulazione del medesimo: l'eseguibile risultante può essere eseguito su qualsiasi macchina, con la massima efficienza. Se il TSR viene eseguito su un personal computer privo di coprocessore matematico e le funzioni residenti effettuano calcoli in virgola mobile, i problemi sono assicurati.

La situazione appare, in effetti, complessa: compilare per la generazione di codice specifico per il coprocessore, escludendo così il consolidamento delle librerie di emulazione (opzioni `-f87` e `-f287`), rende il programma ineseguibile su macchine non dotate dello hardware necessario e, d'altra parte, non evita che esso incorpori, quanto meno, le funzioni di libreria dedicate all'inizializzazione del coprocessore stesso. L'obiettivo di ottimizzazione del TSR può dirsi raggiunto solo se queste sono eseguite esclusivamente nella fase di caricamento e startup del programma: ancora una volta, l'attenta lettura della documentazione del compilatore e un po' di sperimentazione si rivelano indispensabili.

A prescindere dalle questioni legate all'efficienza del programma, vi sono casi in cui è comunque inopportuno che le routine residenti richiamino funzioni di libreria, in particolare quando queste ultime invocano, a loro volta, l'int 21h[282]: l'uso dell'int 21h al momento sbagliato da parte di un TSR può provocare il crash del sistema; infatti, a causa delle modalità di gestione da parte del DOS dei propri stack interni, esso non può essere invocato ricorsivamente da un TSR (cioè mentre un suo servizio è attivo). Sull'argomento si tornerà tra breve con maggiore dettaglio.

---

[282] Molte funzioni di libreria si servono dell'int 21h: tra esse quelle relative alla gestione dello I/O con gli streams, i files e la console (`printf()`, `fopen()`, `open()`, `getch()`, etc.).

GESTIONE DEGLI INTERRUPT

Tutti i TSR incorporano routine di gestione degli interrupt di sistema, in quanto è questo il solo mezzo che essi possono utilizzare per rimanere attivi dopo l'installazione in RAM. La gestione degli interrupt è dunque di importanza cruciale e deve essere effettuata senza perdere di vista alcuni punti fondamentali[283].

Hardware, ROM-BIOS e DOS

Il TSR deve tenere sotto controllo il sistema, per intercettare il verificarsi dell'evento che ne richiede l'attivazione (solitamente la digitazione di una determinata sequenza, detta *hotkey*, sulla tastiera), installando un gestore dell'int 09h, il quale è generato dal chip dedicato alla tastiera ogni qualvolta un tasto è premuto o rilasciato.

Il TSR, quando intercetta lo hotkey, deve essere in grado di decidere se soddisfare la richiesta di attivazione oppure attendere un momento più "propizio": una inopportuna intromissione nell'attività del BIOS o del DOS potrebbe avere conseguenze disastrose.

Gli interrupt hardware, infatti, sono pilotati da un chip dedicato, che li gestisce in base a precisi livelli di priorità, inibendo cioè l'attivazione di un interrupt mentre ne viene servito un altro avente priorità maggiore. I TSR non devono dunque attivarsi se è in corso un interrupt hardware, onde evitare l'inibizione, per un tempo indefinitamente lungo, di tutti quelli successivamente in arrivo.

Va ricordato inoltre che le routine del ROM-BIOS sono non rientranti (cioè non ricorsive[284]): un TSR non può, pertanto, invocare una funzione BIOS già in corso di esecuzione. Le routine di gestione degli interrupt BIOS incorporate nel TSR devono garantire l'impossibilità di ricorsione mediante opportuni strumenti, ad esempio l'utilizzo di flag.

Quanto detto con riferimento agli interrupt del ROM-BIOS vale, in parte, anche per le funzioni DOS: vi sono, cioè, casi in cui il TSR può interrompere l'esecuzione dell'int 21h, ma è comunque opportuno un comportamento prudente. In sintesi: il DOS non consente, a causa delle proprie modalità di gestione dello stack[285], di chiamare una funzione dell'int 21h che ne faccia uso mentre la medesima o un'altra funzione facente uso dello stack viene servita a seguito di una precedente richiesta. Se il TSR si attivasse e interrompesse l'int 21h con una chiamata allo stesso, il DOS, per rispondere alla nuova chiamata, ripristinerebbe il puntatore al proprio stack e quindi perderebbe i dati appartenenti al programma interrotto, compreso l'indirizzo di ritorno dalla chiamata originaria all'int 21h (la coppia `CS:IP` si trova anch'essa sullo stack).

Problemi di gestione dello stack possono inoltre verificarsi se il TSR viene attivato quando il DOS si trova in una condizione di "errore critico", per risolvere la quale è necessario l'intervento dell'utente (ad esempio: stampante spenta o senza carta; sportello del drive aperto, etc.).

Vediamo, in dettaglio, qualche suggerimento per la gestione degli interrupt "delicati".

---

[283] Notizie di carattere generale sugli interrupt si trovano a pag. 115.

[284] Circa la ricorsione vedere pag. 100.

[285] Vogliamo proprio essere precisi? Il DOS gestisce tre stack interni, ciascuno dei quali è costituito da un'area di memoria di circa 200 byte (non si tratta, dunque, di stack nel senso letterale del termine, bensì di buffer statici). Il primo e il secondo sono utilizzati dall'int 21h, per le funzioni 00h-0Ch e, rispettivamente, per tutte quelle restanti; il terzo è usato dall'int 24h nelle situazioni di errore critico.

I flag del DOS

Il DOS gestisce due flag, detti `InDOS flag` e `CritErr flag`, originariamente destinati ad uso "interno" e menzionati, poco e male, nella documentazione ufficiale solo a partire dalle più recenti edizioni. Ognuno di essi occupa un byte nel segmento di RAM ove è caricato il sistema operativo e vale, per default, zero: il primo è modificato quando viene servito l'int 21h e resettato al termine del servizio; il secondo è forzato a un valore non nullo quando si verifica un errore critico ed è azzerato in uscita dall'int 24h (inoltre l'int 24h azzera l'`InDOS flag`: precauzione necessaria, dal momento che l'utente potrebbe decidere di non completare l'operazione in corso). E' facile intuire che essi possono rivelarsi di grande utilità: un TSR, controllando che entrambi siano nulli, individua il momento adatto per attivarsi senza il rischio di disturbare l'attività del DOS[286]. L'indirizzo dell'`InDOS flag` è fornito dalla funzione 34h (anch'essa scarsamente documentata) dell'int 21h, come descritto a pagina .

Riportiamo il codice di una funzione che restituisce l'indirizzo dell'`InDOS flag` sotto forma di puntatore `far` a carattere; l'indirezione di tale puntatore fornisce il valore del flag.

```
/********************

    BARNINGA_Z! - 1991

    INDOSADR.C - getInDOSaddr()

    char far *cdecl getInDOSaddr(void);
    Restituisce: il puntatore all'InDOS flag

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx indosadr.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>

char far *cdecl getInDOSaddr(void)
{
```

[286] In particolare, per quanto riguarda l'`InDOS flag`, va osservato che quando esso non è nullo un servizio DOS è in corso di esecuzione, pertanto i TSR non devono utilizzare alcun servizio dell'int 21h, al fine di evitare la distruzione del contenuto dello stack del DOS. Controllando lo stato del flag, un TSR è in grado di determinare quando è possibile invocare funzioni DOS senza pericolo di compromettere lo stato del sistema. Tuttavia esiste una complicazione: l'interprete di comandi `COMMAND.COM` ed alcuni altri programmi trascorrono gran parte del tempo in attesa di input dalla tastiera mediante la funzione 0Ah (`GetString`) dell'int 21h: l'`InDOS flag`, in tale circostanza, è non-nullo. E' possibile aggirare il problema intercettando l'int 28h, chiamato in loop dai servizi 00h-0Ch, oppure dotando il TSR di un gestore dell'int 21h in grado di intercettare le chiamate alla funzione 0Ah nel modo seguente:

1) Non invocare il servizio immediatamente; eseguire invece un loop costituito da un ritardo seguito da una chiamata al servizio 0Bh dell'int 21h (`GetInputStatus`).

2) Continuare ad eseguire il loop sino a quando non venga segnalato (dal servizio 0Bh) che un tasto è pronto nel buffer della tastiera.

3) Solo a questo punto eseguire la chiamata alla funzione 0Ah: per tutto il tempo in cui viene eseguito e rieseguito il loop, il TSR può utilizzare i servizi DOS liberamente.

La descrizione dei servizi 0Ah e 0Bh dell'int 21h è riportata a pag. 313.

```
    union REGS regs;
    struct SREGS sregs;

    regs.h.ah = 0x34;
    (void)intdos(&regs,&regs);
    segread(&sregs);
    return(MK_FP(sregs.es,regs.x.bx));
}
```

Ottenere l'indirizzo del `CritErr flag` è meno semplice. A partire dalla versione 3.1 del DOS, esso si trova nel byte che precede l'`InDOS flag`: l'indirizzo è pertanto pari a quello dell'`InDOS`, decrementato di uno. Ad esempio:

```
    ....
    char far *InDOSPtr, far *CritErrPtr;

    InDOSPtr = getInDOSaddr();
    CritErrPtr = InDOSPtr-1;
    ....
```

Nelle versioni precedenti il segmento dell'indirizzo del `CritErr flag` è il medesimo di quello relativo all'`InDOS flag`, mentre l'offset deve essere ricavato dal campo "operando" dell'istruzione assembler `CMP` che si trova, nello stesso segmento DOS, nella sequenza[287] qui riportata:

```
cmp ss:[NearByte],00H
jne NearLabel
int 28H
```

In altre parole, una volta ottenuto l'indirizzo dell'`InDOS` mediante l'int 21 funzione 34h, occorre scandire il segmento che inizia all'indirizzo `ES:0000` alla ricerca degli opcode relativi alle istruzioni di cui sopra (si tratta, ovviamente, di codice compilato); il campo operando (`[NearByte]`, 2 byte) dell'istruzione `CMP SS...` è l'offset del `CritErr flag`. Sicuri di evitare un fastidio al lettore, presentiamo una funzione in grado di ricavare gli indirizzi di entrambi i flag. Dal momento che le funzioni in C restituiscono un solo valore, `getDOSflagsptrs()` fa uso di un puntatore a struttura per renderli disponibili entrambi alla routine chiamante. Essa si basa quasi interamente sullo inline assembly a scopo di efficienza e compattezza.

```
/********************

    BARNINGA_Z! - 1991

    DOSPTRS.C - getDOSfptrs()

    int cdecl getDOSfptrs(struct DOSfptrs DosFlagsPtrs *);
    struct DOSfptrs DosFlagsPtrs; punta alla struttura di tipo DOSfp
    Restituisce: 1 in caso di errore (seq. 0xCD 0x28 non trovata);
                 0 altrimenti.
```

[287] Codice del file `IBMDOS.COM` (o `MSDOS.SYS`). Chi desiderasse complicarsi la vita, potrebbe ricercare una sequenza simile, invece di decrementare l'offset dell'indirizzo dell'`InDOS flag`, anche nelle versioni di DOS dalla 3.1 in poi:

```
test ss:[NearByte],0FFH
jne  NearLabel
push ss:[NearWord]
int  28H
```

```
    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx -Tm2 dosptrs.C

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh.

    Il parametro -Tm2 evita che TASM segnali l'errore "forward
    reference needs override" (le labels che fanno da puntatori agli
    opcodes sono utilizzate prima della loro compilazione). Se la
    versione di TASM di cui si dispone non accetta tale parametro, il
    problema puo' essere aggirato spostando in testa alla funzione il
    codice fasullo che si trova tra le due rem, facendolo precedere da
    un'istruzione jmp che ne eviti l'esecuzione.

********************/
#pragma  inline
#pragma  warn -par

#define  DOS_3_31       0x030A
#define  MAX_SEARCH     0xFFFF

struct DOSfptrs {
    char far *InDOSPtr;                                 // puntatore all'InDOS flag
    char far *CritErrPtr;                              // puntatore al CritErr flag
};

int getDOSfptrs(struct DOSfptrs *DosFlagsPtrs)
{
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
    asm push ds;
    asm lds si,DosFlagsPtrs;
#else                                          // DS:SI = indirizzo della struttura
    asm mov si,DosFlagsPtrs;
#endif
    asm {
        mov ah,0x34;
        int 0x21;                     // chiede al DOS l'ind. dell'InDOS flag (ES:BX)
        mov ds:[si],bx;
        mov ds:[si+2],es;                     // copia ind. nel 1° punt. della struct
        push bx;                                // salva BX (usato da funzione 0x30)
        mov ah,0x30;
        int 0x21;                                    // chiede al DOS la versione DOS
        pop bx;
        xchg ah,al;                      // AX diventa cresecente con la versione DOS
        cmp ax,DOS_3_31;
        jb _SEARCH;                               // salta se versione inferiore a 3.10
        dec bx;                       // se no il CritErr e' il byte preced. l'InDOS
        jmp _STORE_DATA;
    }
_SEARCH:                       // se DOS < 3.10 cerca il CritErr in IBMDOS/MSDOS in RA
    asm {
        xor di,di;
        mov cx,MAX_SEARCH;                         // si cerca in un seg di 64 Kb max.
        mov ax,word ptr _DUMMY_INT28H;                        // carica AX con 0x28CD
    }
_REPEAT:
    asm {
        repne scasb;                                                   // cerca 0xCD
        jne _ERROR;                                        // ERRORE se non trovato
        cmp ah,es:[di];
        jne _REPEAT:                          // ripete se il byte succ. non e' 0x28
        mov ax,word ptr _DUMMY_CODE;                          // carica AX con 0x8036
        cmp ax,es:[di+_DUMMY_CODE-_DUMMY_INT28H-1];
        jne _REPEAT;                     // ripete se non c'e' in RAM stessa sequenza
```

```
        mov al,byte ptr (_DUMMY_CODE+2);                        // carica AL con 0x06
        cmp al,es:[di+_DUMMY_CODE-_DUMMY_INT28H+1]
        jne _REPEAT;                    // ripete se non c'e' in RAM stessa sequenza
        mov al,byte ptr _DUMMY_JMP;                             // carica AL con 0x75
        cmp al,es:[di+_DUMMY_JMP-_DUMMY_INT28H-1];
        jne _REPEAT;                    // ripete se non c'e' in RAM stessa sequenza
        mov bx,es:[di+_DUMMY_CODE-_DUMMY_INT28H+2];
    }                                                       // ind. di CritErr in BX
_STORE_DATA:
    asm {
        mov ds:[si+4],bx;
        mov ds:[si+6],es;                     // copia ind. nel 2° punt. della struc
        xor ax,ax;                                   // nessun errore: restituisce 0
        jmp _EXIT_FUNC;
    }
/*******************
 finto codice: l'ind. di CritErr e' [byte ptr NearLabel] in cmp ss:
*******************/
_DUMMY_CODE:
    asm {
_DUMMY_CODE label near;
        cmp ss:[byte ptr _DUMMY_CODE],0x00;
_DUMMY_JMP label near;
        jne _DUMMY_CODE;
_DUMMY_INT28H label near;
        int 0x28;
    }
/*******************
 fine finto cod. i cui opcodes sono usati per cercare ind. di CritErr
*******************/
_ERROR:
    asm mov ax,0x01;                                        // errore: restituisce 1
_EXIT_FUNC:
#if defined(__COMPACT__) || defined(__LARGE__) || defined(__HUGE__)
    asm pop ds;
#endif
    return(_AX);
}
```

La `getDOSfptrs()`, dopo avere salvato nel primo campo della struttura `DosFlagPtrs` l'indirizzo dell'`InDOS flag`, ottenuto, mediante la funzione 34h dell'int 21h, utilizza la funzione 30h del medesimo interrupt per conoscere la versione del DOS. Tale servizio restituisce in `AH` il numero della revisione e in `AL` il numero della versione: scambiando tra loro i valori dei due registri si ottiene in `AX` il valore esprimente versione e revisione: questo è maggiore per versione più recente. Un test effettuato sul registro `AX` consente di scegliere l'algoritmo appropriato all'individuazione dell'indirizzo del `CritErr flag`. E' interessante osservare che il codice della `getDOSfptrs()` contiene le istruzioni i cui opcode devono essere ricercati, per versioni di DOS anteriori alla 3.1, nel segmento `ES:0000`. Tale stratagemma evita di compilare a parte tali istruzioni per conoscere la sequenza di byte da ricercare. I riferimenti agli opcode sono effettuati sfruttando la capacità dell'assembler di utilizzare le labels come puntatori.

L'algoritmo utilizzato ricerca dapprima gli opcode corrispondenti all'istruzione `INT 28H` e, localizzati questi, controlla quelli relativi alle istruzioni precedenti: lo scopo è abbreviare il tempo necessario alla ricerca, in quanto l'opcode dell'istruzione `INT` appare, in `IBMDOS.COM` (e `MSDOS.SYS`), con minore frequenza rispetto a quello che rappresenta il registro `SS` nell'istruzione `CMP`.

Lasciamo al lettore il compito (leggi: grattacapo) di realizzare in linguaggio C una funzione in grado di restituire il puntatore al `CritErr flag`. Ecco alcuni suggerimenti utili per... grattarsi un po' meno il capo:

1) Versione e revisione del DOS sono disponibili nelle variabili globali (`char`) `_osmajor` e `_osminor`.

2) La sequenza di opcode da ricercare è:

   `0x36 0x80 0x06 0x?? 0x?? 0x00 0x75 0x?? 0xCD 0x28`

   ove `0x??` rappresenta un valore sconosciuto

3) La coppia di byte `0x??` ad offset 3 nella sequenza può essere gestita come unsigned integer: infatti essa è l'offset del `CritErr flag`.

4) Tenere sott'occhio la `getInDOSaddr()`.

Int 05h (BIOS): Print Screen

L'int 05h è invocato direttamente dalla routine BIOS di gestione dell'int 09h quando viene premuto il tasto `PRTSC`. Per evitare l'attivazione del TSR durante la fase di stampa del contenuto del video è sufficiente che esso controlli il valore presente nel byte che si trova all'indirizzo `0:0500`; esso è un flag che può assumere tre valori, a seconda dello stato dell'ultima operazione di Print Screen:

VALORI DEL FLAG DI STATO DELL'INT 05H

VALORE	SIGNIFICATO
`00h`	Print Screen terminato.
`01h`	Print Screen in corso.
`FFh`	Print Screen interrotto a causa di errori.

Esempio:

```
....
if(!(*((char far *)0x500)))
    do_popup();
....
```

Attenzione: il vettore originale dell'int 05h è ripristinato nascostamente dalla funzione `keep()`, che rende residente il TSR[288]. Questo, pertanto, qualora installi un gestore dell'int 05h deve evitare l'uso della `keep()`, ricorrendo direttamente all'int 21h, servizio 31h per rendersi residente.

[288] Responsabile è la `_restorezero()`, routine non documentata di servizio della `keep()`. La `_restorezero()` provvede anche a ripristinare i vettori degli int 00h, 04h e 06h, per i quali valgono dunque le affermazioni riguardanti l'int 05h.

Int 08h (Hardware): Timer

Il timer del PC (chip 8253) richiede un interrupt 18.21 volte al secondo al controllore degli interrupt (chip 8259). Questo, se gli interrupt sono abilitati, utilizza la linea IRQ0 per comunicare con il processore, che trasferisce il controllo alla routine che si trova all'indirizzo presente nella tavola dei vettori di interrupt, ad offset `20h` (`08h*04h`). Incorporando nel TSR una routine di gestione dell'int 08h è possibile dotarlo di un sistema ad azione continua per l'intercettazione dello hotkey e il controllo delle condizioni di sicurezza ai fini dell'attivazione. Va però ricordato che l'int 08h di default esegue, operazioni fondamentali per il corretto funzionamento della macchina, quali l'aggiornamento del timer di sistema (all'indirizzo `0:46C`) e lo spegnimento dei motori dei drive dopo circa due secondi in cui non siano effettuate operazioni di lettura e/o scrittura: il gestore inserito nel TSR deve effettuare queste operazioni o, quantomeno, consentire al gestore originale di occuparsene e di inviare al chip 8259 il segnale di fine interrupt. E' dunque prudente strutturare il nuovo gestore in modo che, innanzi tutto, trasferisca il controllo a quello originale e solo al rientro da questo si occupi delle elaborazioni necessarie al TSR. Ecco un esempio (semplificato):

```
void interrupt newint08h(void)
{
    asm pushf;
    asm call dword ptr oldint08h;
    ....
    ....
}
```

L'esempio assume che la variabile oldint08h contenga l'indirizzo del gestore originale; l'istruzione `PUSHF` è necessaria in quanto la `CALL` trasferisce il controllo a una routine di interrupt, la quale termina con un'istruzione `IRET` (che esegue automaticamente il caricamento del registro dei flag prelevando una word dallo stack). Successivamente alla `CALL`, il gestore può occuparsi delle operazioni necessarie al TSR, quali test e attivazione (popup). Ecco alcune importanti precauzioni:

1) Non invocare le funzioni 01h-0Ch dell'int 21h nel gestore dell'int 08h (onde evitare la distruzione dello stack da parte del DOS, provocata dalla ricorsione).

2) Verificare che l'`InDos flag` e il `CritErr flag` valgano entrambi `0`.

3) Verificare che non sia in corso un PrintScreen (int 05h).

4) Vedere pagina 266.

Int 09h (Hardware): Tastiera

Il controllore della tastiera richiede un int 09h ogni volta che un tasto è premuto o rilasciato. La routine BIOS originale aggiorna il buffer della tastiera oppure gestisce i casi particolari (`CTRL-BREAK`, `CTRL-ALT-DEL`, `PRTSC`, etc.). Se si desidera che l'attivazione del TSR avvenga a seguito della pressione di uno hotkey, esso deve agganciare il vettore dell'int 09h (cioè deve incorporare una routine che lo gestisca). Il nuovo gestore, individuato lo hotkey, può limitarsi a modificare un flag che viene successivamente controllato (vedere quanto detto circa l'int 28h, a pag.) per effettuare il popup. E' indispensabile che il nuovo gestore si incarichi di reinizializzare la tastiera (per evitare che lo hotkey sia passato al programma interrotto) e di inviare al chip 8259 il segnale di fine interrupt (per evitare che il sistema resti bloccato indefinitamente, dal momento che il chip 8259 inibisce gli interrupt sino al completamento di quello in corso). Un esempio:

```
/********************

    BARNINGA_Z! - 1991

    NEWINT09.C - newint09h()

    void interrupt newint09h(void);

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx -Tm2 newint09.C

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh.

********************/
#define HOT_KEY  0x44                                  // lo hot-key e' il tasto F10

void interrupt newint09h(void)
{
    asm {
        in al,0x60;                            // legge lo scan code del tasto premuto
        cmp al,HOT_KEY;
        je SETFLAG;
        pop bp;                                  // pulire lo stack prima di saltare!!
        pop di;                                     // ricordarsi che il compilatore C
        pop si;                                    // genera automaticamente il codice
        pop ds;                                        // necessario a salvare tutti i
        pop es;                                   // registri in entrata alle funzioni
        pop dx;                                                // dichiarate interrupt
        pop cx;
        pop bx;
        pop ax;
        jmp dword ptr oldint09h;                         // concatena routine BIOS orig.
    }
SETFLAG:
    asm {
        in al,0x61;                                   // legge lo stato della tastiera
        mov ah,al;                                                        // lo salva
        or al,0x80h;                     // setta il bit di abilitazione della tastiera
        out 0x61,al;                                           // abilita la tastiera
        mov al,ah;                                         // ricarica in AL lo stato
        out 0x61,al;                                               // e lo ripristina
        mov al,0x20;                           // prepara il segnale di fine interrupt
        out 0x20,al;                                       // e lo invia al chip 8259
        pop ax;                                // ...ricordarsi della PUSH iniziale...
    }
    popup_flag = 1;
}
```

Come si vede, `newint09h()` svolge le proprie operazioni a stretto contatto con lo hardware, leggendo e scrivendo direttamente sulle porte (vedere pag. 271); se lo scan code letto sulla porta 60h non è quello dello hotkey il controllo è ceduto al gestore originale: l'uso dell'istruzione `JMP` richiede la totale pulizia dello stack (ad eccezione di `CS`, `IP` e registro dei flag) perché la `IRET` del gestore originale non determina il rientro in `newint09h()`, bensì direttamente nella procedura interrotta, ripristinando così il registro dei flag ed utilizzando (come indirizzo di ritorno) la coppia `CS:IP` spinta sullo stack all'ingresso di `newint09h()`. Prima di terminare, `newint09h()` aggiorna la variabile (globale) `popup_flag` (è in questo punto del codice che esso potrebbe attivare il TSR); il nuovo gestore dell'int 28h e segg.) effettua il popup se `popup_flag` ha valore `1`.

Spesso lo hotkey deve essere costituito da una combinazione di tasti e di shift (`CTRL`, `ALT`, `SHIFT` etc.): il gestore dell'int 09h può conoscere lo stato degli shift ispezionando il contenuto di una coppia di byte utilizzati, a tal fine, dal sistema.

I valori degli shift status byte possono essere ottenuti, ad esempio, nel seguente modo:

```
....
char shfstatus;

shfstatus = *((char far *)0x417);
....
```

In alternativa, è possibile utilizzare gli appositi servizi dell'int 16h (vedere pag.). Inoltre, ulteriori informazioni sullo stato della tastiera possono essere ricavate dal Keyboard Status Byte, che si trova all'indirizzo `0:0496`.

SHIFT STATUS BYTE ED EXTENDED SHIFT STATUS BYTE

BIT A 1	**SIGNIFICATO DEL BIT PER SHIFT STATUS (0:0417)**	**SIGNIFICATO DEL BIT PER EXTENDED SHIFT STATUS (0:0418)**
7	Insert attivo	Insert premuto
6	Caps Lock attivo	Caps Lock premuto
5	Num Lock attivo	Num Lock premuto
4	Scroll Lock attivo	Scroll Lock premuto
3	Alt premuto	Pause attivato
2	Ctrl premuto	SysReq premuto
1	Shift Sinistro premuto	Alt sinistro premuto
0	Shift Destro premuto	Ctrl destro premuto

KEYBOARD STATUS BYTE

BIT A 1	SIGNIFICATO DEL BIT
7	Lettura ID in corso
6	Ultimo carattere = primo carattere di ID
5	Forza Num Lock se lettura ID e KBX
4	La tastiera ha 101/102 tasti
3	Alt Destro premuto
2	Ctrl Destro premuto
1	Ultimo codice = E0 Hidden Code
0	Ultimo codice = E1 Hidden Code

Int 10h (BIOS): Servizi video

E' buona norma non interrompere un servizio dell'int 10h: si pensi alle conseguenze che potrebbe avere l'attivazione del TSR proprio mentre il BIOS sta, ad esempio, cambiando la modalità video. Un TSR deve dunque gestire l'int 10h per motivi di sicurezza ed utilizzare opportunamente un flag.

```
int ok_to_popup = 1;                                        // flag sicurezza pop-up

void far newint10h(void)
{
    asm mov word ptr ok_to_popup,0x00;
    asm pushf;
    asm call dword ptr oldint10h;
    ....
    ....
    asm mov word ptr ok_to_popup,0x01;
#if __TURBOC__ > 0x0200
    asm pop bp;                          // BP PUSHed da TURBOC se versione C++
#endif
    asm iret;
}
```

La `newint10h()` chiama il gestore originale dell'interrupt con la tecnica descritta nel paragrafo dedicato all'int 08h (pag.). La routine che ha il compito di effettuare il popup (ad esempio l'int 08h stesso o l'int 28h) può effettuare un test del tipo:

```
    ....
    if(ok_to_popup)
        do_popup();
    ....
```

Si noti che `newint10h()` non è dichiarata di tipo `interrupt`, ma di tipo `far`. Ciò perché, come più volte accennato, il compilatore C genera automaticamente, per le funzioni `interrupt`, il codice necessario al salvataggio di tutti i registri in ingresso alla funzione e al loro ripristino in uscita: se `newint10h()` fosse dichiarata `interrupt` e non si provvedesse esplicitamente a sostituire nello stack i valori spinti dalla chiamata all'interrupt con quelli restituiti dal gestore originale invocato con la `CALL`, il processo chiamante non potrebbe in alcun caso conoscere lo stato del sistema conseguente alla chiamata. Attenzione, però, all'eventuale codice qui rappresentato dai puntini: se esso utilizza i registri modificati dal gestore originale, questi devono essere salvati (e ripristinati). La funzione di libreria `setvect()` "digerisce" senza problemi un puntatore a funzione `far` purché si effettui un opportuno cast a funzione `interrupt`[289]. La dichiarazione far impone, inoltre, di pulire lo stack[290] e terminare esplicitamente la funzione con una `IRET`, che sarebbe stata inserita per default dal compilatore in una funzione dichiarata `interrupt`. Circa l'int 10h vedere anche a pag. .

Int 13h (BIOS): I/O dischi

E' opportuno non interrompere un servizio di I/O su disco: vale quanto detto sopra circa l'int 10h, come dimostra l'esempio che segue.

```
int ok_to_popup = 1;                                    /* flag sicurezza pop-up */

void far newint13h(void)
{
    asm mov word ptr ok_to_popup,0x00;
    asm pushf;
    asm call dword ptr oldint13h;
    ....
    ....
    asm mov word ptr ok_to_popup,0x01;
#if __TURBOC__ > 0x0200
    asm pop bp;                           /* BP PUSHed da TURBOC se versione C++ */
#endif
    asm iret;
}
```

Int 16h (BIOS): Tastiera

L'int 16h gestisce i servizi BIOS per la tastiera. A differenza dell'int 09h esso non viene invocato da un evento asincrono quale la pressione di un tasto, bensì via software: si tratta, in altre parole, di un'interfaccia tra applicazione e tastiera. In particolare, il servizio 00h dell'int 16h sospende l'elaborazione in corso e attende che nel buffer della tastiera siano inseriti scan code e ASCII code di un tasto premuto, qualora essi non siano già presenti: se nel corso di una funzione DOS è richiesto tale servizio, un TSR non può essere attivato per tutta la durata dello stesso. L'ostacolo può essere aggirato da

---

[289] Il cast è `(void(interrupt *)())`.

[290] Si ricordi che il compilatore Borland C++ genera le istruzioni necessarie alla gestione di `BP` ed `SP` anche nelle funzioni dichiarate `void funzione(void)`, mentre ciò non avviene con il compilatore TURBO C 2.0. Utilizzando quest'ultimo è quindi indispensabile eliminare l'istruzione `POP BP` che precede la `IRET`. A complicare ulteriormente le cose giunge il fatto che il modello huge salva `DS` sullo stack in ingresso alla funzione e lo ripristina in uscita: occorre tenerne conto se si compila il TSR con detto modello di memoria.

un gestore dell'int 16h che rediriga le richieste di servizio 00h al servizio 01h (semplice controllo della eventuale presenza di dati nel buffer) e intraprenda l'azione più opportuna a seconda di quanto riportato da quest'ultimo. Un esempio di gestore dell'int 16h è riportato a pag. : per adattarlo alle esigenze dei TSR è sufficiente sostituire la parte di codice etichettata `LOOP_0` con il listato seguente.

```
LOOP_0:
CTRL_C_0:

    asm {
        mov ax,dx;                                  // controlla se un tasto è pronto
        pushf;                     // necessaria a causa dell'IRET nel gestore orig.
        cli;
        call dword ptr oldint16h;                              // usa come subroutine
        jz IDLE_LOOP;                                  // nessun tasto pronto (ZF = 1)
        mov ax,dx;                   // un tasto è pronto: occorre estrarlo dal buf.
        dec ah;
        pushf;
        cli;
        call dword ptr int16hdata;
        cmp al,03H;                                            // è CTRL C or ALT 3 ?
        je CTRL_C_0;                                                     // sì: salta
        cmp ax,0300H;                                                // no: è CTRL 2 ?
        je CTRL_C_0;                                                     // sì: salta
        pop dx;                     // no, non è una CTRL C sequence: pulisci lo stack
        iret;                                 // passa il tasto alla routine chiamante
    }

IDLE_LOOP:                        // gestisce attivita' altri TSR se il DOS e' libero

    asm {
        push ds;
        push bx;
        lds bx,InDOSflag;                         // carica in DS:BX l'indirizzo del flag
        mov al,byte ptr [bx];                                    // carica il flag in AL
        pop bx;
        pop ds;
        cmp al,00H;
        jg LOOP_0;                        // se il flag non e' 0 meglio lasciar perdere
        int 28H;                              // altrimenti si puo' interrompere il DOS
        jmp LOOP_0;                                  // e riprendere il loop in seguito
    }
```

La differenza rispetto all'esempio di pag. è evidente: all'interno del ciclo `LOOP_0` viene effettuato un test sull'`InDOSflag`, per invocare se possibile, a beneficio di altri TSR, l'int 28h.

Descriviamo brevemente alcuni dei servizi resi disponibili dall'int 16h:

INT 16H, SERV. 00H: LEGGE UN TASTO DAL BUFFER DI TASTIERA

Input	AH	00h
Output	AH	scan code
	AL	ASCII code
Note		INT 16h, FUNZ. 10h è equivalente, ma supporta la tastiera estesa e non è disponibile su tutte le macchine.

INT 16H, SERV. 01H: CONTROLLA SE È PRESENTE UN TASTO NEL BUFFER DI TASTIERA

Input	`AH`	`01h`
Output	`ZFlag`	`1` = buffer di tastiera vuoto `2` = carattere presente nel buffer: `AH` = scan code `AL` = ASCII code
Note		INT 16h, FUNZ. 11h è equivalente, ma supporta la tastiera estesa e non è disponibile su tutte le macchine.

INT 16H, SERV. 02H: STATO DEGLI SHIFT

Input	`AH`	`02h`
Output	`AL`	Byte di stato degli shift (v. pag. 302)
Note		INT 16h, FUNZ. 12h è equivalente, ma supporta la tastiera estesa e non è disponibile su tutte le macchine. Restituisce in `AH` il byte di stato esteso.

INT 16H, SERV. 05H: INSERISCE UN TASTO NEL BUFFER DELLA TASTIERA

Input	`AH`
	`CH`
	`CL`
		`05h`
		scan code
		ASCII code

Output	`AL`	`00h` = operazione riuscita `01h` = buffer di tastiera pieno

Infine, ecco alcuni ulteriori dati utili per una gestione personalizzata del buffer della tastiera[291]:

[291] Forse è opportuno spendere qualche parola sulle modalità di gestione, realizzata attraverso quattro puntatori, del buffer della tastiera. Il suo indirizzo nella RAM è dato dal puntatore all'inizio: questo e il puntatore al byte successivo alla fine ne segnano dunque i confini. Il buffer è inoltre scandito da altri due puntatori, dei quali il primo (puntatore alla testa) indica la posizione del primo tasto estratto dal buffer stesso, mentre il secondo (puntatore alla coda) indica l'indirizzo al quale è "parcheggiato" il successivo tasto premuto (cioè i suoi codici di scansione e ASCII). Quando deve essere inserito un tasto nel buffer, il puntatore alla coda viene incrementato; se il suo valore supera quello del puntatore alla fine viene "riavvolto" al valore del puntatore all'inizio. Se, a seguito di questo algoritmo, esso eguaglia il puntatore alla testa si verifica la condizione di buffer pieno: il puntatore è decrementato. Si noti che in questo caso il tasto è regolarmente inserito all'indirizzo indicato dal puntatore alla coda prima dell'incremento, ma viene ignorato: il buffer di tastiera può pertanto "trattare" un tasto in meno rispetto a quanto consentito dallo spazio allocato (normalmente 15 battute, ciascuna delle quali impegna una word dei 32 byte disponibili). Quando invece deve essere letto un tasto dal buffer, è incrementato il puntatore alla testa: se esso

PUNTATORI AL BUFFER DI TASTIERA

OGGETTO	INDIRIZZO	BYTE
Puntatore al buffer di tastiera	`0:480`	`2`
Puntatore al byte seguente la fine del buffer	`0:482`	`2`
Puntatore alla testa del buffer	`0:41A`	`2`
Puntatore alla coda del buffer	`0:41C`	`2`
Carattere immesso con `ALT`+tastierino numerico	`0:419`	`1`

Il buffer di tastiera è normalmente situato all'indirizzo `0:041E`, ma può essere rilocato via software (per un esempio si veda pag. 378 e seguenti); la sua ampiezza di default, anch'essa modificabile, è 32 byte: esso contiene però 15 tasti al massimo, in quanto l'eventuale sedicesimo forzerebbe il puntatore alla coda ad essere uguale al puntatore alla testa, condizione che invece significa "buffer vuoto". Del buffer di tastiera e dei suoi puntatori si parla anche a pag. 384.

Int 1Bh (BIOS): CTRL-BEAK

Il gestore BIOS dell'int 1Bh è costituito da una istruzione `IRET`. Il DOS lo sostituisce con una routine propria, che modifica due flag: il primo, interno al DOS medesimo e controllato periodicamente, consente a questo di intraprendere le opportune azioni (solitamente l'interruzione del programma; è la routine del `CTRL-C`) quando venga rilevato il `CTRL-BREAK` su tastiera; il secondo, situato nella low memory all'indirizzo `0:0471`, viene posto a `1` all'occorrenza del primo `CTRL-BREAK` e non più azzerato (a meno che ciò non sia fatto dall'utente) e può essere utilizzato, ad esempio, da un processo *parent* per sapere se il *child process* è terminato via `CTRL-BREAK`.

Un TSR deve gestire il vettore del `CTRL-BREAK` assumendone il controllo quando viene attivato e restituendolo al gestore originale al termine della propria attività, onde evitare di essere interrotto da una maldestra azione dell'utente. La routine di gestione può essere molto semplice; se il suo scopo è soltanto fungere da "buco nero" per le sequenze `CTRL-BREAK` essa può assumere la struttura seguente:

```
void interrupt newint1Bh(void)
{
}
```

Dichiarando `far` (e non `interrupt`) la `newint1BH()` si può evitare che essa incorpori le istruzioni per il salvataggio e il ripristino dei registri: essa dovrebbe però essere chiusa esplicitamente da una istruzione inline assembly `IRET`. Si veda inoltre quanto detto circa l'int 23h (pag.).

Va sottolineato che la sostituzione del vettore dell'int 1Bh non deve avvenire al momento dell'installazione del TSR, ma al momento della sua attivazione (per consentire alle applicazioni utilizzate

---

eguaglia il puntatore alla coda, allora il buffer è vuoto. Un'ultima precisazione: i quattro puntatori esprimono offset rispetto all'indirizzo di segmento `400h`.

successivamente di utilizzare il gestore originale) e deve essere ripristinato al termine dell'attività in foreground. Vedere pagina 268.

Int 1Ch (BIOS): Timer

L'int 1Ch, il cui gestore di default è semplicemente un'istruzione `IRET`, viene invocato dall'int 08h (timer) per dare alle applicazioni la possibilità di svolgere attività di background basate sul timer, senza gestire direttamente l'int 08h. Abbiamo suggerito sopra (vedere pag.) un sistema alternativo di gestione del timer tick, consistente nell'invocare il gestore originale all'interno di `newint08h()` e svolgere successivamente le azioni necessarie al TSR. Di qui la necessità di intercettare l'int 1Ch, onde evitare che altre applicazioni possano utilizzarlo disturbando o inibendo il TSR: Il gestore dell'int 1Ch può essere una funzione "vuota" identica a quella dell'esempio relativo all'int 1Bh; `newint08h()` invoca il vettore originale dell'int 1Ch:

```
void interrupt newint1Ch(void)
{
}

void interrupt newint08h(void)
{
    asm pushf;
    asm call dword ptr oldint08h;
    ....
    ....
    asm pushf;
    asm call dword ptr oldint1Ch;
}
```

Per quale motivo è necessario invocare esplicitamente l'int 1Ch? Questo esempio offre lo spunto per un chiarmento importante: per gestore (o vettore) originale di un interrupt non si intende necessariamente quello installato dal BIOS o dal DOS al bootstrap, ma quello che viene sostituito dalla routine incorporata nel nostro codice, in quanto quello è l'unico indirizzo che noi possiamo in ogni istante leggere (e modificare) nella tavola dei vettori. Si pensi, per esempio, al caso di più TSR caricati in sequenza, ciascuno dei quali incorpori un gestore per un medesimo interrupt: il primo aggancia il gestore installato dal BIOS (o dal DOS), il secondo quello installato dal primo, e via dicendo. Ovunque possibile, per dare spazio alle applicazioni concorrenti, è dunque opportuno che i gestori di interrupt trasferiscano il controllo alle routine da essi "sostituite", invocandole al proprio interno (`CALL`) o concatenandole (`JMP`). Il vettore dell'int 1Ch deve essere agganciato all'attivazione del TSR e ripristinato al termine dell'attività in foreground. Vedere pag. 266.

Int 21h (DOS): servizi DOS

La grande maggioranza delle funzionalità offerte dal DOS è gestita dall'int 21h, attraverso una serie di servizi accessibili invocando l'interrupt con il numero del servizio stesso nel registro `AH` (ogni servizio può, comunque, richiedere un particolare utilizzo degli altri registri; frequente è il numero di sub-funzione in `AL`); i problemi da esso posti ai TSR sono legati soprattutto alla sua non-rientranza, di cui si è detto poco sopra (pag.).

In questa sede non appare necessario dilungarsi sull'int 21h, in quanto servizi e loro caratteristiche sono descritti laddove se ne presentano le modalità e gli scopi di utilizzo.

Int 23h (DOS): CTRL-C

Il gestore dell'int 23h è invocato quando la tastiera invia la sequenza CTRL-C. La routine di default del DOS chiude tutti i file aperti, libera la memoria allocata e termina il programma: un TSR deve gestire il CTRL-C in quanto, come si intuisce facilmente, lasciar fare al DOS durante l'attività del TSR stesso scaricherebbe le conseguenze sull'applicazione interrotta. Non ci sono particolari restrizioni alle azioni che un gestore dell'int 23h può intraprendere, ma occorre tenere presenti alcune cosette:

1) All'ingresso nel gestore tutti i registri contengono i valori che contenevano durante l'esecuzione della routine interrotta.

2) Se si intende semplicemente ignorare il CTRL-C, il gestore può essere una funzione "vuota".

3) Se, al contrario, si desidera effettuare operazioni è indispensabile salvare il contenuto di tutti i registri e ripristinarlo in uscita dal gestore.

4) All'interno del gestore possono essere utilizzati tutti i servizi DOS; se, però, il gestore utilizza le funzioni 01h-0Ch dell'int 21h e durante l'esecuzione di una di esse è nuovamente premuto CTRL-C il DOS distrugge il contenuto del proprio stack (vedere pag.).

5) Se il gestore, anziché restituire il controllo alla routine interrotta, intende terminare l'esecuzione del programma, esso deve chiudersi con un'istruzione RET (e non IRET; attenzione allo stack e al codice generato per default dal compilatore).

6) Il gestore BIOS dell'int 09h, quando intercetta un CTRL-C, inserisce un ASCII code 3 nel buffer della tastiera: vedere il paragrafo dedicato all'int 16h (pag.).

Il punto 4) merita un chiarimento, poiché il problema, apparentemente fastidioso, è in realtà di facile soluzione: basta incorporare nel TSR due gestori dell'int 23h, dei quali uno "vuoto" e installare questo durante l'esecuzione di quello "complesso":

```
void interrupt safeint23h()
{
}


void interrupt newint23h()
{
    asm mov ax,seg safeint23h;
    asm mov ds,ax;
    asm mov dx,offset safeint23h;
    asm mov ah,0x25;
    asm int 0x21;
        ....
        ....
    asm mov ah,0x25;
    asm lds dx,dword ptr newint23h;
    asm int 0x21;
}
```

Anche un semplice flag può essere sufficente:

```
void interrupt newint23h()
{
    asm cmp inInt23hFlag,0;                    // se il flag e' 0 non c'e' ricorsione
```

```
    asm jne EXIT_INT;                 // stiamo gia' servendo un int 23h; pussa via!
    asm mov inInt23hFlag,1;                          // segnala che siamo nell'int 23h
        ....
        ....
    asm mov inInt23hFlag,0;      // fine dell'int 23h; pronti per eseguirne un altro
EXIT_INT:;
}
```

Il listato di `newint23h()` non evidenzia il salvataggio dei registri in ingresso e il loro ripristino in uscita perché il codice necessario è generato dal compilatore C (`newint23h()` è dichiarata `interrupt`): vedere pag. 268.

Va ancora sottolineato che il vettore dell'int 23h è copiato dal DOS in un apposito campo del PSP (vedere pag.) al caricamento del programma e da tale campo è ripristinato al termine della sua esecuzione: ciò vale anche per i programmi TSR, che terminano con la `keep()` o con l'int 21h, servizio 31h. Se il nuovo gestore dell'int 23h deve essere attivo subito dopo l'installazione del TSR (e non solo al momento della sua attivazione), questo deve copiarne il vettore nel PSP prima di terminare:

```
    *(long *)MK_FP(_psp,0x0E) = (long)newint23h;
```

E' il DOS medesimo ad attivare, involontariamente, il nuovo gestore dell'int 23h al momento della terminazione del programma, liberando questo dall'obbligo di farlo "di persona". L'indirizzo della `newint23h()` subisce un cast a `long` per semplificare l'espressione: del resto, un puntatore `far` e un `long` sono entrambi dati a 32 bit.

Int 24h (DOS): Errore critico

L'int 24h è invocato dal DOS nelle situazioni in cui si verifica un errore che non può essere corretto senza l'intervento dell'utente: ad esempio un tentativo di accesso al disco mentre lo sportello del drive è aperto. In questi casi la routine di default del DOS visualizza il messaggio "`Abort, Retry, Ignore, Fail?`" e attende la risposta. Ogni programma può installare un gestore personalizzato dell'int 24h, per evitare che il video sia "sporcato" dal messaggio standard del DOS o per limitare o estendere le possibilità di azione dell'utente; ciò è particolarmente importante per un TSR, al fine di evitare che in caso di errore critico sia la routine del programma interrotto ad assumere il controllo del sistema. Va ricordato, però, che il TSR deve installare il proprio gestore al momento dell'attivazione e ripristinare quello originale prima di restituire il controllo all'applicazione interrotta, per evitare di cambiarle le carte in tavola.

Come nel caso dell'int 23h, anche il vettore dell'int 24h è copiato dal DOS in un apposito campo del PSP (vedere pag.) al caricamento del programma e da tale campo è ripristinato al termine della sua esecuzione. Se il nuovo gestore dell'int 24h deve essere attivo subito dopo l'installazione del TSR (e non solo al momento della sua attivazione), questo deve copiarne il vettore nel PSP prima di terminare:

```
    *(long *)MK_FP(_psp,0x12) = (long)newint24h;
```

Il nuovo gestore dell'int 24h è attivato automaticamente dal DOS quando il programma termina. L'indirizzo della `newint24h()` subisce un cast a `long` per semplificare l'espressione: del resto, un puntatore `far` e un `long` sono entrambi dati a 32 bit.

Infine, durante un errore critico il DOS si trova in condizione di instabilità: è dunque opportuno non consentire l'attivazione del TSR in tali situazioni, inserendo un test sul `CritErr flag` (di cui si è detto ampiamente a pag. 295 e seguenti) nelle routine residenti dedicate al monitoring del sistema o alla intercettazione dello hotkey.

Int 28h (DOS): DOS libero

L'int 28h, scarsamente documentato, può costituire, analogamente all'int 08h, uno dei punti di entrata nel codice attivo del TSR. Il DOS lo invoca quando attende un input dalla tastiera; in altre parole, durante le funzioni 01h-0Ch dell'int 21h. Il gestore di default è semplicemente una istruzione `IRET`: il TSR deve installare il proprio, che può effettuare un controllo sul flag utilizzato da altri gestori (esempio: int 09h) per segnalare la richiesta di attivazione.

```
....
if(popup_requested)
    do_popup();
....
```

E' però importante osservare alcune precauzioni:

1) Non invocare le funzioni 01h-0Ch dell'int 21h nel gestore dell'int 28h (onde evitare la distruzione dello stack da parte del DOS, provocata dalla ricorsione). Inoltre sotto DOS 2.x non devono essere invocati servizi 50h e 51h, salvo i casi in cui il `CritErr flag` non è nullo (pag. 295).

2) Verificare che `InDos flag` e `CritErr flag` valgano `0`.

3) Verificare che non sia in corso un PrintScreen (int 05h; pag. 299).

4) Chiamare (`CALL`) o concatenare (`JMP`) al momento opportuno il gestore originale dell'int 28h (regola peraltro valida con riferimento a tutti i gestori di interrupt).

5) Tenere presente che le applicazioni che non fanno uso dei servizi 01h-0Ch dell'int 21h non possono essere interrotte dai TSR che si servono dell'int 28h quale unico popup point.

Int 2Fh (DOS): Multiplex

L'int 2Fh costituisce una via di comunicazione con i TSR: esso è ufficialmente riservato a tale scopo ed è utilizzato da alcuni programmi inclusi nel DOS (`ASSIGN`, `PRINT`, `SHARE`).

Questo interrupt è spesso utilizzato, ad esempio, per verificare se il TSR è già presente nella RAM, onde evitarne l'installazione in più istanze. Microsoft riserva a tale controllo il servizio 00h: il TSR, quando viene lanciato, invoca l'int 2Fh funzione 00h (valore in `AL`) dopo avere caricato il registro `AH` con il valore scelto quale identificatore[292]. Il gestore del servizio 00h dell'int 2Fh incorporato nel TSR deve essere in grado di riconoscere tale identificatore e di restituire in `AL` un secondo valore, che costituisce una sorta di "parola d'ordine" e indica al programma di essere già presente in RAM. Questo algoritmo si fonda sulla unicità, per ogni applicazione, del byte di identificazione e della parola d'ordine: è evidente che se due diversi programmi ne condividono i valori, solo uno dei due (quello lanciato per primo) può installarsi in RAM.

La loro unicità può essere garantita, ad esempio, assegnandoli a variabili di ambiente con il comando DOS `SET` oppure passandoli al TSR come parametro nella command line[293].

---

[292] I valori da `00h` a `7Fh` sono riservati al DOS (`PRINT` usa `AH = 01h`, `ASSIGN` usa `AH = 02h`, `SHARE` usa `AH = 10h`); le applicazioni possono dunque utilizzare uno qualsiasi dei valori restanti (`80h-FFh`).

[293] Inoltre, una word (anziché un byte) consente di esprimere un maggior numero di combinazioni: il gestore dell'int 2Fh potrebbe utilizzare altri registri per effettuare il controllo (ad esempio `BX`, o qualunque altro registro non utilizzato da altri servizi dello stesso interrupt), per diminuire la probabilità di conflitto con altre applicazioni.

Il codice necessario alla richiesta di parola d'ordine potrebbe essere analogo al seguente:

```
....
regs.h.al = 0x00;
regs.h.ah = MultiplexId;
(void)int86(0x2F,&regs,&regs);
if(regs.x.ax != AlreadyPresent)
    install();
....
```

Ed ecco una proposta di gestore dell'int 2Fh, o almeno della sua sezione iniziale:

```
void interrupt newint2Fh(int Bp,int Di,int Si,int Ds,int Es,int Dx,int Cx,int Bx,
                                                                        int Ax)
{
    if(((Ax >> 8) & 0xFF) == MultiplexId) {
        switch(Ax & 0xFF) {
            case 0x00:
                Ax = AlreadyPresent;
                return;
            ....
```

Le variabili `MultiplexId` (byte di identificazione) e `AlreadyPresent` (parola d'ordine) sono variabili globali dichiarate, rispettivamente, `char` e `int`.

Segue schema riassuntivo delle regole standard di comunicazione con routine residenti via int 2Fh:

INT 2FH: MULTIPLEX INTERRUPT

Input	AH	byte identificativo del programma chiamante
	AL	numero del servizio richiesto
	Altri registri	utilizzo libero
Output	Tutti i registri	utilizzo libero
Note		Per convenzione il servizio 0 è riservato alla verifica della presenza in RAM del TSR: l'int 2Fh deve restituire (in AL o AX) un valore prestabilito come parola d'ordine. Se l'int 2Fh non riconosce il byte di identificazione (passato in AH dal processo chiamante) deve concatenare il gestore originale dell'interrupt.

Quello presentato non è che uno dei servizi implementabili in un gestore dell'int 2Fh; va comunque sottolineato che questo rappresenta la via ufficiale di comunicazione tra il TSR ed il sistema in cui esso opera, in base ad un meccanismo piuttosto semplice. Esso prevede che il processo che deve interagire con la porzione residente del TSR (spesso è la porzione transiente del medesimo programma,

---

L'algoritmo qui descritto è, come si è detto, quello consigliato da Microsoft (e, pertanto, ufficiale), ma attualmente non esistono motivi di carattere tecnico che impediscano al programmatore di utilizzare metodi alternativi.

nuovamente lanciato al prompt del DOS dopo l'installazione) invochi l'int 2Fh con il byte di riconoscimento in AH e il sevizio richiesto in AL; l'uso degli altri registri è libero.

Ricordiamo ancora una volta che il compilatore C, nel caso di funzioni di tipo interrupt, genera automaticamente il codice assembler per effettuare in ingresso il salvataggio di tutti i registri (flag compresi) ed il loro ripristino in uscita: di qui la necessità di dichiararli quali parametri formali[294] di newint2Fh(). In tal modo, ogni riferimento a detti parametri nel corpo della funzione è in realtà un riferimento ai valori contenuti nello stack: ciò consente ad una funzione di tipo interrupt di restituire un valore al processo chiamante, proprio perché in uscita, come accennato, i valori dei registri sono "ricaricati" con il contenuto dello stack.

Il meccanismo di comunicazione costituito dall'int 2Fh si rivela utile in un altro caso notevole: la disinstallazione del TSR. Si discute diffusamente dell'argomento a pag. .

GESTIONE DELLO I/O

Tastiera

Si è detto che, normalmente, un TSR viene attivato mediante la pressione di una combinazione di tasti, detta hotkey sequence. Notizie dettagliate sulla tastiera, gli interrupt che la gestiscono ed i servizi da questi resi disponibili si trovano nei paragrafi dedicati all'int 09h e all'int 16h (pagg. e). Ulteriori esempi di gestori per l'int 09h si trovano alle pagine e .

In questa sede vale la pena di riportare la descrizione di due servizi dell'int 21h, utili per pilotare le operazioni di I/O e di popup del TSR (vedere anche pag. 295).

INT 21H, SERV. 0AH: INPUT DI UNA STRINGA MEDIANTE BUFFER

Input	AH DS:DX	0Ah indirizzo (seg:off) del buffer
Output		Nessuno; al ritorno il buffer contiene la stringa (vedere note)
Note		Il primo byte del buffer deve contenere, quando il servizio è invocato, la massima lunghezza consentita per la stringa (al massimo 254 caratteri, incluso il RETURN - ASCII 0Dh - finale). Il secondo byte contiene, in uscita, la lunghezza effettiva della stringa, escluso il RETURN terminatore. La stringa è memorizzata a pertire dal terzo byte del buffer.

[294] L'ordine dei parametri formali, lo ripetiamo, è rigido. Per il compilatore Borland esso è: BP, DI, SI, DS, ES, DX, CX, BX, AX, IP, CS, FLAGS, seguiti da eventuali parametri definiti dal programmatore. Non è necessario dichiarare tutti i parametri elencati; è però della massima importanza che la dichiarazione abbia inizio dal primo registro (BP) e includa tutti i registri referenziati nel codice della funzione, senza escludere quelli in posizione intermedia eventualmente non utilizzati (vedere pag. 251 e seguenti).

INT 21H, SERV. 0BH: CONTROLLO DELLO STATO DELL'INPUT

Input	AH	0Bh
Output	AL	FFh se un carattere è disponibile nello Standard Input (generalmente la tastiera); 00h se non vi è alcun carattere.
Note		Questo servizio può essere utilizzato, tra l'altro, per intercettare la pressione di CTRL-BREAK durante lunghi cicli di calcolo o altre elaborazioni che non attendono un input dalla tastiera, allo scopo di uscirne a richiesta dell'utente. E' superfluo aggiungere che, allo scopo, il programma (TSR o no) deve incorporare un gestore dell'int 1Bh (vedere pag. 307).

Video

Devono preoccuparsi di gestire il video tutti i TSR che, durante l'attività in foreground, ne modificano il contenuto: è indispensabile, infatti, che essi lo ripristinino prima di restituire il controllo all'applicazione interrotta. Di qui la necessità di salvare (in altre parole: copiare) il buffer video in un array di caratteri appositamente allocato, ed effettuare l'operazione opposta al momento opportuno. Le tecniche utilizzabili sono più di una: è possibile, ad esempio, ricorrere all'int 10h oppure, se lo si preferisce[295], accedere direttamente alla memoria video. La tecnica di accesso diretto al buffer video può variare a seconda dello hardware installato e della modalità video selezionata; in questa sede ci occupiamo, in particolare, di alcuni servizi dell'int 10h limitandoci, per brevità, alla gestione della modalità testo.

INT 10H, SERV. 00H: STABILISCE NUOVI MODO E PAGINA VIDEO

Input	AH	0Fh
	AL	modo video: MODO COLONNE RIGHE COLORE 00h 40 25 grigi 01h 40 25 si 02h 80 25 grigi 03h 80 25 si 07h 80 25 16 EGA

[295] Si tratta, in realtà, di una preferenza condizionata: il primo metodo offre maggiori garanzie di portabilità, mentre il secondo presenta i vantaggi di una maggiore velocità e di un codice eseguibile più compatto (soprattutto se la routine è scritta direttamente in linguaggio assembler). Le nuove schede grafiche introdotte sul mercato (EGA, VGA) hanno eliminato il problema dello "sfarfallio" (effetto neve) provocato, in alcune circostanze, dall'accesso diretto in scrittura alla memoria video con schede poco evolute (CGA).

INT 10H, SERV. 02H: SPOSTA IL CURSORE ALLE COORDINATE SPECIFICATE

Input	AH	02h
	BH	numero della pagina video: PAGINE VALIDE MODI VIDEO 0-7 00h-01h 0-3 02h-03h 0 04h-07h 0-7 0Dh 0-3 0Eh 0-1 0Fh
	DH	riga
	DL	colonna

INT 10H, SERV. 03H: LEGGE LA POSIZIONE DEL CURSORE

Input	AH	03h
	BH	numero della pagina video
Output	CH	Scan line iniziale del cursore
	CL	Scan line finale
	DH	Riga
	DL	Colonna

INT 10H, SERV. 05H: STABILISCE LA NUOVA PAGINA VIDEO

Input	AH	05h
	AL	numero della pagina video

INT 10H, SERV. 08H: LEGGE CARATTERE E ATTRIBUTO ALLA POSIZIONE DEL CURSORE

Input	AH	08h
	BH	numero della pagina video
Output	AH	attributo del carattere. Vedere pag. 455 in nota.
	AL	codice ASCII del carattere

INT 10H, SERV. 09H: SCRIVE CARATTERE E ATTRIBUTO ALLA POSIZIONE DEL CURSORE

Input	`AH`	`09h`
	`AL`	codice ASCII del carattere
	`BH`	numero della pagina video
	`BL`	attributo del carattere
	`CX`	numero di coppie car/attr da scrivere
Note		Non sposta il cursore.

INT 10H, SERV. 0EH: SCRIVE UN CARATTERE IN MODO TTY

Input	`AH`	`0Eh`
	`AL`	codice ASCII del carattere
	`BH`	numero della pagina video
Note		Il carattere è scritto alla posizione del cursore, che viene spostato a destra di una colonna, portandosi a capo se necessario.

INT 10H, SERV. 0FH: LEGGE IL MODO E LA PAGINA VIDEO ATTUALI

Input	`AH`	`0Fh`
Output	`AH`	numero di colonne
	`AL`	modo video
	`BH`	pagina video

INT 10H, SERV. 13H: SCRIVE UNA STRINGA CON ATTRIBUTO

Input	AH	13h
	AL	subfunzione: 0 = usa l'attributo in BL; non sposta il cursore 1 = usa l'attributo in BL; sposta il cursore 2 = usa gli attributi nella stringa; non sposta il cursore 3 = usa gli attributi nella stringa; sposta il cursore
	BH	pagina video
	BL	attributo colore (per servizi 0-1)
	CX	lunghezza della stringa
	DH	riga a cui scrivere la stringa
	DL	colonna a cui scrivere la stringa
	ES:BP	indirizzo della stringa da scrivere
Note		Le subfunzioni 2 e 3 interpretano la stringa come una sequenza di coppie di byte carattere/attributo: la stringa è copiata nel buffer video così come è. **Attenzione**: modificare il valore di BP implica l'impossibilità di utilizzare le variabili automatiche (in quanto ad esse il compilatore accede con indirizzi relativi proprio a BP; vedere pag. 158 e seguenti) fino al momento del suo ripristino. E' consigliabile salvarlo con una PUSH BP immediatamente prima di caricare l'indirizzo della stringa in ES:BP e ripristinarlo con una POP BP subito dopo la chiamata all'int 10h.

Per un esempio di utilizzo del servizio 13h dell'int 10h vedere pag. 450 e seguenti.

Ecco, in dettaglio, un esempio di strategia operativa, ipotizzando il caso di un TSR che lavori esclusivamente in modo testo 80x25 (se il modo video non è appropriato il TSR rinuncia[296]):

1) Controllare il modo video tramite int 10h, serv. 0Fh.

[296] Si tratta di una semplificazione piuttosto notevole, che consente, peraltro, di alleggerire (almeno in parte!) il discorso. Ci limitiamo a sottolineare che la sua rimozione comporta l'analisi di due situazioni possibili: il TSR potrebbe modificare la modalità video attraverso il servizio 00h dell'int 10h, oppure adattare il proprio comportamento alla modalità video rilevata, con l'evidente vantaggio di potere in ogni caso effettuare il pop-up. Quanto alla prima strategia, va osservato che il servizio 00h dell'int 10h cancella il video: ciò avrebbe conseguenze estetiche certamente poco desiderabili. D'altra parte, la scelta del secondo metodo complicherebbe notevolmente la vita al programmatore, in quanto richiederebbe la realizzazione di routine di output a video molto flessibili e parametriche, senza perdere di vista, naturalmente, le consuete esigenze di compattezza ed efficienza.

2) Se il registro `AL` non contiene `2`, `3` o `7` è impossibile effettuare il pop-up: uscire dalla routine segnalando (ad esempio con un bip) la situazione all'utente.

3) Altrimenti, se si prevede di modificare il contenuto di `BH`, salvarlo (esso è utilizzato nelle successive chiamate all'int 10h).

4) Eseguire l'int 10h, serv. 03h per conoscere la posizione attuale del cursore e salvare il contenuto di `DX`.

5) Portare il cursore alla posizione desiderata mediante il serv. 02h dell'int 10h (se si desidera salvare tutto il video andare in `0,0`).

6) Eseguire un loop che, ad ogni iterazione, mediante il serv. 08h dell'int 10h salvi in un apposito array di interi una coppia attributo/carattere e muova il cursore mediante il serv. 02h: se si desidera salvare tutto il video il loop deve essere eseguito 2000 volte (80 x 25).

7) Se si desidera nascondere il cursore è sufficiente portarlo a riga 26.

8) Effettuare le operazioni di output connesse all'attività di foreground del TSR.

9) Portare il cursore nell'angolo superiore sinistro dell'area di video salvata in precedenza.

10) Ripristinarne il contenuto con un loop analogo a quello descritto al punto 6).

11) Riportare il cursore alla posizione originaria.

File

Le numerose funzioni di libreria atte alla gestione dei file possono essere suddivise in due gruppi: del primo fanno parte quelle (come la `fopen()`, la `fread()`, etc.) che operano attraverso una struttura dati di tipo `FILE` associata al file (il cosiddetto stream); al secondo appartengono quelle operanti semplicemente attraverso lo handle associato al file (`open()`, `read()`, etc.: vedere pag. 115). Nella parte transiente di un TSR è possibile utilizzare qualsiasi funzione senza correre alcun tipo di rischio, mentre, al contrario, nelle routine residenti è necessaria maggiore attenzione: l'uso delle funzioni appartenenti al primo gruppo è sconsigliato, in quanto esse utilizzano tecniche di allocazione dinamica della memoria "invisibili" al DOS. Quanto detto deve essere esteso anche a quelle del secondo gruppo quando si ricorra a funzioni come, ad esempio, la `setbuf()` per gestire attraverso un buffer la operazioni di I/O. Infine, ricordiamo che, in generale, l'uso di funzioni di libreria nelle routine residenti comporta alcuni problemi, dei quali si è detto a pag. .

Meglio, allora, armarsi di santa pazienza e ricorrere direttamente ai servizi resi disponibili dall'int 21h, anche se ciò può comportare il ricorso allo inline assembly[297].

Sono, comunque, indispensabili alcune precauzioni importanti: in primo luogo va osservato che i file manipolati dalle routine residenti devono essere da queste gestiti con "cicli" completi durante l'attività in foreground. Ogni file deve, cioè, essere aperto, letto e/o scritto e, infine, richiuso prima di restituire il controllo all'applicazione interrotta. Vanno evitati nel modo più assoluto comportamenti

---

[297] D'altra parte non sarebbe la prima volta (né l'ultima...): abbiamo visto che è quasi impossibile scrivere codice residente efficiente e compatto senza l'aiuto di alcune estensioni del linguaggio, peraltro non sempre portabili verso tutti i compilatori, quale è, appunto, lo inline assembly.

pericolosi, come aprire i file in fase di installazione e lasciarli aperti a beneficio delle routine residenti, che gestiranno le operazioni di I/O senza mai chiuderli e riaprirli. Si tenga presente che il DOS è, di solito, in grado di mantenere un limitato numero di file aperti contemporaneamente: uno handle attivo ma inutilizzato rappresenta una risorsa preziosa sottratta al sistema. Inoltre, cosa ancor più importante, un TSR (salvo il caso in cui sia dotato di capacità di system monitoring particolarmente sofisticate) non può sapere che accade ai propri file per tutto il tempo in cui altre applicazioni sono attive in foreground: il tentativo (ed è solo un esempio tra i tanti possibili) di leggere dati da un file che non esiste più produrrebbe effetti analoghi a quelli di un tuffo in una piscina vuota[298].

In secondo luogo non bisogna dimenticare che molte delle opzioni accettate dalle funzioni di libreria inerenti la modalità di apertura dei file sono gestite dal DOS attraverso chiamate a servizi differenti. Di seguito, senza pretese di completezza, presentiamo alcuni schemi esplicativi.

INT 21H, SERV. 3CH: CREA UN NUOVO FILE O NE TRONCA UNO ESISTENTE

Input	`AH`	`3Ch`
	`CX`	attributo del file: `00h` = normale `01h` = sola lettura `02h` = nascosto `04h` = di sistema
	`DS:DX`	indirizzo (seg:off) del nome del file (stringa ASCIIZ)
Output	`AX`	handle per il file se `CarryFlag` = `0`, altrimenti codice dell'errore
Note		Se il file specificato non esiste, viene creato; se esiste la sua lunghezza è troncata a `0` byte ed il contenuto distrutto.

[298] In effetti, il paragone è azzardato. Chi non sa nuotare immagina certo molte spiacevoli situazioni in cui la presenza dell'acqua nella suddetta piscina non sarebbe, comunque, di conforto.

INT 21H, SERV. 3DH: APRE UN FILE ESISTENTE

Input	AH	3Dh
	AL	modalità di apertura: campi di bit BIT 0-2: 0 = sola lettura 1 = sola scrittura 2 = lettura/scrittura BIT 3: riservato (sempre 0) BIT 4-6: 0 = modo compatibilità 1 = esclusivo 2 = scrittura non permessa 3 = lettura non permessa 4 = permesse lettura e scrittura BIT 7: 0 = utilizzabile da child process 1 = non utilizzabile da child
	DS:DX	indirizzo (seg:off) del nome del file (stringa ASCIIZ)
Output	AX	handle per il file se CarryFlag = 0, altrimenti codice dell'errore.
Note		Se il file specificato non esiste, viene restituito un codice di errore; se esiste viene aperto e il puntatore è posizionato all'inizio del file. Su questo servizio si basa la funzione di libreria _open(): vedere pag. 128.

INT 21H, SERV. 3EH: CHIUDE UN FILE APERTO

Input	AH	3Eh
	BX	handle del file
Output	AX	codice di errore se CarryFlag = 1
Note		I buffer associati al file sono svuotati e la directory viene aggiornata.

INT 21H, SERV. 3FH: LEGGE DA UN FILE APERTO

Input	AH	`3Fh`
	BX	handle del file
	CX	numero di byte da leggere
	DS:DX	indirizzo (seg:off) del buffer di destinazione
Output	AX	codice di errore se `CarryFlag = 1`, altrimenti numero di byte letti.

INT 21H, SERV. 40H: SCRIVE IN UN FILE APERTO

Input	AH	`40h`
	BX	handle del file
	CX	numero di byte da scrivere
	DS:DX	indirizzo (seg:off) del buffer contenente i byte da scrivere
Output	AX	codice di errore se `CarryFlag = 1`, altrimenti numero di byte scritti.

INT 21H, SERV. 41H: CANCELLA UN FILE

Input	AH	`41h`
	DS:DX	indirizzo (seg:off) del nome del file (stringa ASCIIZ)
Output	AX	codice di errore se `CarryFlag = 1`.

INT 21H, SERV. 42H: MUOVE IL PUNTATORE ALLA POSIZIONE NEL FILE

Input	AH AL BX CX:DX	42h punto di riferimento dell'offset: 0 = da inizio file 1 = da posizione corrente 2 = da fine file handle del file spostamento da effettuare in byte (long integer)
Output	AX	codice di errore se CarryFlag = 1, altrimenti DX:AX = nuova posizione nel file (long int).
Note		E' possibile muovere il puntatore oltre la fine del file: in tal caso la lunghezza del file è aggiornata non appena è scritto almeno un byte. Muovere il puntatore ad un offset negativo rispetto all'inizio del file causa invece un errore. E' necessario effettuare almeno uno spostamento (anche se di 0 byte rispetto alla posizione corrente) tra una operazione di lettura ed una di scrittura, o tra una di scrittura ed una di lettura consecutive.

Concludiamo il paragrafo con un suggerimento: l'algoritmo utile per gestire il file in append mode, cioè per aprirlo e scrivere in coda al medesimo:

1) Aprire il file mediante int 21h, serv. 3Dh: se il CarryFlag è 0 saltare al passo 3)

2) Aprire il file mediante int 21h, serv. 3Ch: se il CarryFlag è 1 l'operazione è fallita: uscire.

3) Muovere il puntatore al file di 0 byte rispetto alla fine del file mediante int 21h, serv. 42h (AL = 2; CX = 0; DX = 0;): se CarryFlag è 1 l'operazione è fallita: uscire.

Se l'operazione 3) riporta CarryFlag = 0 è allora possibile effettuare le operazioni di scrittura, non trascurando di chiudere il file mediante int 21h, serv. 3Eh prima di restituire il controllo all'applicazione interrotta. Con la versione 3.0 del DOS è stato introdotto il servizio 5Bh, che agisce in maniera del tutto analoga al servizio 3Ch, ma con una importante differenza: fallisce se il file esiste (invece di troncarlo). Per un esempio pratico si veda a pag. .

D T A

Il DTA (Disk Transfer Address) è un buffer di 128 byte utilizzato da alcuni servizi dell'int 21h nelle operazioni di I/O con i dischi; per default esso è situato ad offset 80h nel PSP. Al momento del pop-up il TSR agisce nell'ambiente del programma interrotto e ne condivide, quindi, anche il DTA: le routine residenti, qualora utilizzino servizi basati sul DTA, devono salvare l'indirizzo del DTA dell'applicazione interrotta, comunicare al DOS quello del proprio e ripristinare l'indirizzo originale prima di cedere nuovamente il controllo all'applicazione. I servizi DOS che effettuano tali operazioni sono i seguenti:

INT 21H, SERV. 2FH: OTTIENE DAL DOS L'INDIRIZZO DEL DTA ATTUALE

Input	`AH`	`2Fh`
Output	`ES:BX`	Indirizzo (seg:off) del DTA attuale (se il servizio è richiesto dalla porzione residente di un TSR, normalmente è quello del programma interrotto).

INT 21H, SERV. 1AH: COMUNICA AL DOS L'INDIRIZZO DEL DTA

Input	`AH` `DS:DX`	`1Ah` Indirizzo (seg:off) del DTA

Infine, presentiamo l'elenco dei servizi dell'int 21h che fanno uso del DTA e dunque impongono al TSR le precauzioni di cui si è detto:

SERVIZI DELL'INT 21H UTILIZZANTI IL DTA

11h	FindFirst (Espansione wildcard) mediante FCB
12h	FindNext (Espansione wildcard) mediante FCB
14h	Lettura sequenziale mediante FCB
15h	Scrittura sequenziale mediante FCB
21h	Lettura Random mediante FCB
22h	Scrittura Random mediante FCB
27h	Lettura Random a Blocchi mediante FCB
28h	Scrittura Random a Blocchi mediante FCB
4Eh	FindFirst (Espansione wildcard)
4Fh	FindNext (Espansione wildcard)

Ad eccezione degli ultimi due in elenco, si tratta di servizi dedicati alla gestione dei file mediante File Control Block [299]: dal momento che le funzionalità da essi offerte si ritrovano nei più recenti

[299] Buffer utilizzati dalle prime versioni di DOS per tenere traccia dei file gestiti dalle applicazioni. Uno dei limiti da essi presentati è che consentono di operare unicamente su file nella directory corrente (la versione 1.0 del DOS non implementa le directory: in ogni disco ne esiste una sola, la *root*).

(e preferibili) servizi basati sulla tecnica degli handle[300], solitamente non vi è motivo per utilizzarli nella scrittura di programmi per i quali non sia richiesta la compatibilità con versioni di DOS anteriori alla 2.0.

GESTIONE DEL PSP

Il PSP (Program Segment Prefix) è un'area di 256 byte riservata dal DOS in testa al codice di ogni programma caricato ed eseguito. Essa contiene dati di vario tipo ed impiego ed "ospita" il DTA di default; senza entrare nel merito, in questa sede intendiamo trattarne alcuni aspetti che possono risultare di qualche interesse con riferimento ai TSR.

Va precisato, innanzitutto, che qualunque programma può conoscere l'indirizzo di segmento del proprio PSP utilizzando il servizio 62h dell'int 21h[301].

INT 21H, SERV. 62H: OTTIENE DAL DOS L'INDIRIZZO DEL PSP ATTUALE

Input	`AH`	`62h`
Output	`BX`	Indirizzo di segmento del PSP attuale (se il servizio è richiesto dalla porzione residente di un TSR, normalmente è quello del programma interrotto).

Tale indirizzo è il valore salvato dallo startup code (vedere pag. 105) nella variabile globale `_psp` (definita in `DOS.H`); si ricordi però che gestendo i dati globali della parte residente con lo stratagemma della `Jolly()`, detta variabile non è disponibile dopo l'installazione: il suo valore deve pertanto essere copiato nello spazio riservato ai dati dalla `Jolly()` durante la fase stessa di installazione. Le routine residenti possono invocare il servizio di cui sopra per conoscere l'indirizzo del PSP dell'applicazione attiva in quel momento (non il proprio: il TSR condivide l'ambiente dell'applicazione interrotta, PSP compreso). Salvando opportunamente il valore restituito in `BX`, le routine transienti possono servirsi della funzione 50h dell'int 21h per far conoscere al DOS il loro PSP. E' ovvio che al momento di restituire il controllo all'applicazione interrotta deve essere ripristinato l'originario indirizzo di PSP, ancora mediante il servizio 50h.

INT 21H, SERV. 50H: COMUNICA AL DOS L'INDIRIZZO DEL PSP

Input	`AH`	`50h`
	`BX`	Indirizzo di segmento del PSP

I due servizi esaminati sono disponibili a partire dalla versione 2.0 del DOS, ma solo dalla 3.0 in poi essi non fanno uso dello stack interno di sistema e possono pertanto essere richiesti anche mentre è in corso una precedente chiamata all'int 21h. Se il programma opera sotto una versione di DOS antecedente alla 3.0 il problema può essere aggirato controllando l'`InDOS flag` oppure simulando un errore critico (forzando a `1` il valore del `CritErr flag`), in modo che il DOS non utilizzi lo stack dedicato alle operazioni di I/O (vedere pag.).

---

[300] Introdotti con il DOS 2.0. Quando un'applicazione apre un file, il DOS associa ad esso un valore a 16 bit (lo handle, appunto) al quale l'applicazione fa riferimento per tutte le operazioni relative a quel file. Vedere pag. 126.

[301] Il servizio 51h (dell'int 21h) è identico, ma non documentato.

Conoscere l'indirizzo del PSP della porzione residente del TSR è di importanza fondamentale, tra l'altro, ai fini delle operazioni di disinstallazione: se ne parla a pag. 327.

La word (`unsigned int`) ad offset `02h` nel PSP esprime l'indirizzo di segmento del successivo Memory Control Block (pag. 191): esso rappresenta il limite superiore del blocco di RAM allocata alla parte residente del TSR.

La word (`unsigned int`) ad offset `2Ch` nel PSP esprime l'indirizzo di segmento dell'environment assegnato dal DOS al programma. I TSR possono servirsene, oltre che per accedere alle variabili d'ambiente, per disallocare la RAM assegnata all'environment stesso: vedere pag. .

Particolare interesse rivestono le due doubleword (puntatori `far` a funzione) ad offset `0Eh` e `12h`: esse esprimono gli indirizzi dei gestori attivi dell'int 23h (`CTRL-C/CTRL-BREAK`; pag. 309) e, rispettivamente, dell'int 24h (Errore Critico; pag. 310). Il DOS copia questi due valori nel PSP (dalla tavola dei vettori) quando il programma è invocato ed effettua la copia in direzione opposta (dal PSP alla tavola dei vettori) al termine dell'esecuzione. Ciò implica l'impossibilità, per un TSR, di installare routine permanenti di gestione dei due interrupt suddetti, salvo ricorrere ad un piccolo stratagemma: copiare autonomamente gli indirizzi delle proprie routine di gestione dell'int 23h e dell'int 24h nel PSP durante l'installazione. Ecco come fare:

```
....
*((long far *)MK_FP(_psp,0x0E)) = (long)new23h;
*((long far *)MK_FP(_psp,0x12)) = (long)new24h;
....
```

In tal modo i due gestori appartenenti al TSR rimangono attivi anche dopo la restituzione del controllo al DOS: particolare molto importante, questo, se, ad esempio, il programma residente è in realtà una libreria di funzioni[302] volte a completare e migliorare (o semplicemente a modificare) gli standard di comportamento del sistema operativo. I puntatori e gli indirizzi delle funzioni sono gestiti come dati di tipo `long` piuttosto che come puntatori `far` a funzione: in effetti, si tratta pur sempre di valori a 32 bit; il vantaggio è nella maggiore semplicità formale del listato (circa la macro `MK_FP()`, vedere pag. 24).

Il byte ad offset 80h nel PSP esprime la lunghezza della command line del programma, escluso il nome del programma ed incluso il `CR` (ASCII `0Dh`) che la chiude; la command line si trova ad offset 81h. Tali informazioni sono sovrascritte se il programma utilizza il DTA di default (vedere pag.); sull'argomento si tornerà a pag. , con riferimento alla gestione della command line.

RICAPITOLANDO...

A questo punto dovrebbe essere chiaro che l'attivazione del TSR è un momento delicato, da preparare con accortezza, così come lo sono le attività che esso deve svolgere in foreground. Vediamo allora di raccogliere le idee e di riassumere le operazioni indispensabili per evitare fastidiosi crash di sistema.

Innanzitutto occorre tenere sotto controllo lo stato del DOS, del ROM-BIOS e dello hardware: l'attivazione del TSR deve essere consentita solo se si verificano contemporaneamente alcune fondamentali condizioni:

1) L'`InDOS flag` deve essere zero.

2) Il `CritErr flag` deve essere zero.

[302] Comprendenti, ovviamente, gestori per i due interrupt citati.

3) Nessuno dei seguenti interrupt ROM-BIOS deve essere in corso di esecuzione: 05h, 09h, 10h, 13h.

4) Nessuno degli interrupt hardware deve essere in corso di esecuzione.

Chi ama vivere pericolosamente può, in qualche misura, derogare alle regole appena descritte: se delle condizioni presentate in tabella la prima non è verificata, l'attivazione del TSR è ancora possibile a patto che l'`InDOS flag` sia uguale a `1`, e l'attivazione sia effettuata dal nuovo gestore dell'int 28h. In questo caso le routine residenti devono però evitare l'utilizzo dei servizi 00h-0Ch dell'int 21h (infatti, la situazione descritta si verifica quando il DOS attende un input da tastiera proprio attraverso detti servizi; il loro uso ricorsivo avrebbe conseguenze nefaste).

Si è detto che per conoscere lo stato degli interrupt ROM-BIOS un TSR può servirsi di un flag (vedere, ad esempio, l'int 10h a pag.): tutto ciò vale, ovviamente, anche con riferimento agli interrupt hardware. Per questi ultimi, però, esiste un metodo più sofisticato, consistente nell'interrogare direttamente il loro gestore (il chip 8529A):

```
....
asm {
    mov al,0x0B;                        // valore per la richiesta di stato
    out 20H,al;                                           // interroga 8529A
    jmp $+2;                  // introduce un piccolo ritardo per attendere
    in al,20H;                                          // legge la risposta
}
....
```

I bit del registro `AL` rappresentano i diversi interrupt hardware: per ciascuno di essi il valore `1` indica che quel particolare interrupt è attivo. Ne segue che se non è eseguito alcun interrupt hardware il valore di `AL` è `0`.

Non appena attivato, il TSR deve preoccuparsi di svolgere alcune operazioni, delle quali riportiamo un elenco:

1) Salvare l'indirizzo del PSP corrente e comunicare al DOS quello del proprio.

2) Salvare l'indirizzo del DTA corrente e comunicare al DOS quello del proprio.

3) Salvare il vettore dell'int 24h e installare il proprio.

4) Salvare gli indirizzi dei vettori 23h e 1Bh e installare i propri.

5) Salvare il contenuto del video.

Solo dopo essersi opportunamente preparato il terreno il TSR può, finalmente, iniziare la propria attività in foreground: è evidente, comunque, che le operazioni 1) e 2) possono essere tralasciate se il foreground task del TSR non coinvolge PSP e DTA; analoghe considerazioni valgono a proposito dell'int 24h (il TSR non deve effettuare operazioni di I/O con dischi o stampanti, etc.) e degli int 23h e 1Bh (il TSR utilizza esclusivamente servizi DOS "insensibili" al `CTRL-BREAK/CTRL-C`, quali, ad esempio, le routine di I/O per i file[303]). Infine, è inutile salvare il contenuto del video se il TSR non lo modifica.

[303] Attenzione, però: eventuali sequenze `CTRL-BREAK` o `CTRL-C` digitate durante l'attività in foreground del TSR si "scaricano" sull'applicazione interrotta non appena le viene restituito il controllo.

Al termine dell'attività in foreground è necessario, prima di restituire il controllo all'applicazione interrotta, ripristinare il contenuto del video, i vettori 1Bh, 23h e 24h e gli indirizzi del DTA e del PSP di questa. In altre parole, occorre ripercorrere a ritroso le attività di preparazione.

DISATTIVAZIONE E DISINSTALLAZIONE

I TSR modificano in qualche misura il comportamento del sistema sottraendo alle altre applicazioni, a partire dal momento dell'installazione, la disponibilità di una porzione più o meno rilevante di RAM e, in particolare, sovrapponendosi (o addirittura sostituendosi) al DOS e al BIOS nella gestione delle routine di interrupt: quando occorra ripristinare le normali caratteristiche del sistema si rende necessario resettare la macchina oppure disinstallare (o disattivare) il programma residente.

I termini disinstallazione e disattivazione non sono sinonimi. Con il primo si indica l'interruzione completa e definitiva di ogni forma di attività del TSR, implicante il ripristino di tutti i vettori di interrupt originali, la chiusura di tutti i file da esso eventualmente gestiti e la disallocazione di tutta la RAM ad esso assegnata (codice, eventuali buffer, environment). Il secondo indica, al contrario, il permanere del TSR in memoria: qualora tutti i vettori originali siano ripristinati, esso non ha più alcuna possibilità di riattivarsi[304]. E' però possibile consentire al TSR un livello minimo di attività (ad esempio di solo monitoraggio) in modo tale che esso sia in grado, al verificarsi di una determinata condizione, di reinstallare le proprie routine di gestione degli interrupt e riprendere così in maniera completa lo svolgimento dei propri compiti[305].

Nelle pagine che seguono si analizzerà nel dettaglio il processo di disinstallazione, in quanto le operazioni necessarie alla disattivazione costituiscono un sottoinsieme di quelle ad esso correlate.

keep() ed exit()

La procedura di installazione di un TSR si conclude, generalmente, con una chiamata alla funzione di libreria `keep()`, la quale, prima di invocare il servizio 31h dell'int 21h per terminare l'esecuzione del programma e renderlo residente (si veda pag.), chiama la `_restorezero()`, la quale, definita nello startup code (pag. 105) e non documentata[306], provvede al ripristino di alcuni vettori di interrupt[307], salvati dallo startup medesimo prima della chiamata alla `main()`; i file aperti non vengono chiusi. La chiamata alla `keep()` equivale dunque a qualcosa di analogo al listato seguente:

---

[304] In realtà, a condizione che la RAM ad esso allocata non sia stata liberata, la disattivazione di un TSR potrebbe anche essere realizzata mediante la sospensione integrale di ogni sua attività: in tal caso la riattivazione (reinstallazione dei vettori) dovrebbe essere effettuata dalla parte transiente (il programma deve essere nuovamente invocato al prompt del DOS e comunicare in modo opportuno, ad esempio via int 2Fh, con la parte residente).

[305] Si pensi, per esempio, ad un programma di redirezione dell'output di altre applicazioni, progettato per visualizzare sul monitor tutto ciò che è diretto alla stampante, fino al verificarsi di un evento particolare (come la pressione di uno hotkey prestabilito) che ne interrompa l'azione: da quel momento esso dovrebbe limitarsi a monitorare la tastiera, per riprendere l'attività di redirezione quando intercetti il medesimo hotkey o un altro evento prescelto dal programmatore.

[306] Un listato di `_restorezero()` è presentato a pag. 414.

[307] Per la precisione, si tratta dei vettori 00h, 04h, 05h e 06h. Gli ultimi tre possono essere modificati dalle funzioni appartenenti al gruppo della `signal()`, se utilizzate nel programma. Ne consegue, tra l'altro, che se le routine residenti fanno uso di tali funzioni devono provvedere autonomamente a salvare e a rispristinare i vettori al momento opportuno.

```
    ....
    void _restorezero(void);
    ....
    _restorezero();
    _AH = 0x31;
    _AL = retcode;
    _DX = resparas;
    geninterrupt(0x21);
}
```

La `exit()` termina il programma senza renderlo residente (non è una novità): essa chiude tutti i file aperti e libera la memoria allocata con `malloc()` e simili; in altre parole effettua tutte le operazioni cosiddette di *cleanup*, tra le quali vi è pure la chiamata alla `_restorezero()`. Se ne trae quindi, anche in considerazione dei problemi legati all'utilizzo di funzioni di libreria nella porzione residente dei TSR (pag.), che non è buona politica procedere alla disinstallazione invocando la `exit()`. E' inoltre opportuno reprimere la tentazione di installare il programma con una chiamata diretta all'int 21h, evitando così che sia eseguita la `_restorezero()`, per poterlo poi disinstallare via `exit()`: va tenuto presente che, qualora i dati globali siano gestiti con il famigerato stratagemma della funzione jolly (gli smemorati e i distratti vedano a pag.), il segmento dati viene abbandonato al suo destino, con tutto il suo contenuto.
E' necessario procedere a basso livello, cioè a più stretto contatto con il DOS.

Suggerimenti operativi

Le tecniche di disinstallazione sono, in ultima analisi, due: la prima prevede che tutte le operazioni necessarie allo scopo siano gestite dalla porzione residente, all'interno di una o più routine di interrupt[308]; la seconda, al contario, lascia a queste il solo compito di fornire i dati necessari (vedremo quali) alla porzione transiente, che provvede alla disinstallazione vera e propria. Quest'ultimo approccio richiede che il TSR sia invocato alla riga di comando del DOS una prima volta per essere installato ed una seconda per essere disinstallato, ma è senza dubbio più "robusto" del primo, in quanto tutte le operazioni delicate (ripristino dei vettori, disallocazione della RAM) vengono svolte esternamente a routine di interrupt e il programmatore può ridurre al minimo il ricorso allo inline assembly servendosi, se lo preferisce, delle funzioni di libreria del C. Vediamo come procedere, passo dopo passo:

1) Controllare se il TSR è installato.

2) In caso affermativo richiedere alla porzione transiente l'indirizzo dei dati necessari al completamento dell'operazione.

3) Procedere al ripristino dei vettori di interrupt e alla disallocazione della memoria.

Controllo di avvenuta installazione

Il controllo della presenza del TSR in RAM può essere effettuato via int 2Fh: in proposito si rimanda a quanto detto (ed esemplificato) a pag. .

[308] D'altra parte dovrebbe essere ormai evidente che gli interrupt sono il solo mezzo che i TSR hanno a disposizione per interagire con l'ambiente.

Richiesta dell'indirizzo dei dati

Anche la richiesta dell'indirizzo dei dati può utilizzare il meccanismo di riconoscimento e risposta fornito dall'int 2Fh: la routine transiente carica `AH` con il byte di identificazione ed `AL` con il numero del servizio corrispondente, appunto, alla richiesta in questione; l'int 2Fh risponde restituendo l'indirizzo (il quale altro non è che quello della funzione jolly residente in RAM) in `AX` se si tratta di un segmento, o in `DX:AX` se è di tipo `far` (quest'ultimo caso è il più frequente). Questo indirizzo deve necessariamente essere richiesto alla porzione residente del TSR in quanto la parte transiente attiva non avrebbe altro modo per conoscerlo[309]: va ricordato che in questo caso la parte transiente e quella residente appartengono a due distinte istanze del medesimo programma (la prima installata in RAM e la seconda lanciata in un secondo tempo). Segue esempio:

```
#define UNINSTALL 0x01
....
void far new2Fh(void)
{
    ....
    if(_AL == UNINSTALL) {
        asm {
            mov ax,offset Jolly;               /* AX = offset dell'ind di Jolly() */
            mov dx,seg Jolly;               /* DX = segmento dell'ind. di Jolly() */
            iret;
        }
    }
    ....
}
```

Ancora una volta ricordiamo di prestare attenzione allo stack: prima dell'istruzione `IRET` potrebbe essere necessaria una `POP BP` (se il compilatore genera automaticamente le istruzioni `PUSH BP` e `MOV BP,SP` in apertura del codice della funzione). Presentiamo anche un esempio di routine transiente che utilizza l'int 2Fh:

```
void unistall(void)
{
    void far *ResidentJolly;

    _AH = MULTIPLEX_ID;
    _AL = UNINSTALL;
    geninterrupt(0x2F);
    asm {
        mov word ptr ResidentJolly,ax;
        mov word ptr ResidentJolly+2,dx;
    }
    ....
}
```

La variabile `ResidentJolly` è dichiarata puntatore `far` a `void`: con opportune operazioni di cast e somme di offset (analoghe a quelle descritte con riferimento alla funzione `Jolly()`, pag.) essa può agevolmente essere utilizzata come puntatore a qualsiasi tipo di dato. Quali dati? Tutti i vettori di interrupt agganciati dal TSR (usare tranquillamente `setvect()` per ripristinarli) e, naturalmente, l'indirizzo di segmento del PSP del TSR, indispensabile per disallocare la RAM.

[309] Una possibile eccezione è descritta a pag. .

Rimozione della porzione residente del TSR

Questa operazione non presenta particolari problemi. E' sufficiente passare l'indirizzo di segmento del PSP del TSR alla funzione di libreria `freemem()` per raggiungere lo scopo. Supponendo, per semplicità, che detto indirizzo sia il primo dato salvato nello spazio riservato nella `Jolly()` durante l'installazione, si può avere:

```
....
if(freemem(*((unsigned far *)ResidentJolly))
    puts("Errore: impossibile disallocare la RAM.");
....
```

Una precisazione importante: liberare con la `freemem()` la RAM allocata al TSR non significa rilasciare automaticamente quella occupata dal suo environment: la disallocazione di questa deve essere effettuata esplicitamente, salvo il caso in cui si sia già provveduto durante l'installazione. Sull'argomento si è detto fin troppo a pag. .

Precauzioni

Se il TSR che viene disinstallato è l'ultimo installato, si cancella ogni traccia della sua presenza nel sistema: i vettori di interrupt tornano ad essere quelli precedenti all'installazione e la struttura della catena dei MCB viene ripristinata. Ma cosa accade se il TSR non è l'ultimo presente in RAM? Il ripristino dei vettori implica che una chiamata a quegli interrupt trasferisca il controllo alle routine che erano attive prima dell'installazione del TSR stesso: i TSR caricati in RAM successivamente all'installazione e prima della disinstallazione di quello, e che abbiano agganciato i medesimi vettori, sono posti fuori gioco. Inoltre, la disallocazione della RAM provoca un'alternanza di blocchi liberi e blocchi assegnati nella serie dei MCB, situazione gestita con qualche difficoltà da alcune versioni di DOS (soprattutto le meno recenti).

Con riferimento alla disattivazione, si può osservare che non sussiste il problema legato alla gestione della RAM, in quanto essa non viene disallocata, mentre rimane valido quanto detto circa i vettori di interrupt, e ciò non solo nel caso in cui il TSR sia disattivato, ma anche quando venga riattivato, dopo il caricamento di altri programmi[310] (forse è meglio rileggere il capitolo dedicato agli interrupt, pag.).

Può essere quindi opportuno che un TSR, prima di procedere a disattivazione, riattivazione e disinstallazione, controlli di essere l'ultimo programma residente in RAM: solo in questo caso non vi è rischio di compromettere lo stato del sistema. La funzione che presentiamo (per il template della `struct MCB` si veda pag. 195) consente di disinstallare il TSR (lanciandolo nuovamente al DOS prompt) in condizioni di sicurezza (quasi) assoluta:

```
/********************

    BARNINGA_Z! - 1991

    LASTTSR.C - AreYouLast()

    unsigned *cdecl AreYouLast(long far *vecTable,unsigned ResPSP);
    long far *vecTable; puntatore alla copia della tavola dei vettori
                        creata in fase di installazione.
    unsigned ResPSP;    indirizzo di segmento del PSP della parte
                        transiente.
    Restituisce:        NULL se il TSR non e' l'ultimo programma
                        residente in RAM
```

[310] Va però osservato che disattivazione e riattivazione possono essere gestite senza modifiche ai vettori di interrupt: può essere sufficiente un flag opportunamente controllato in testa ai gestori di interrupt che devono essere attivi o inattivi: con un piccolo sacrificio in termini di eleganza, si evitano i problemi di cui sopra.

```
                        In caso contrario restituisce il puntatore ad
                        un array che contiene gli indirizzi di tutti i
                        blocchi che devono essere liberati in quanto
                        allocati al TSR. L'ultimo elemento dell'array
                        e' sempre un NULL.

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx lasttsr.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <stdio.h>
#include <dos.h>
#include <alloc.h>

#define  LASTBLOCK    'Z'

void _restorezero(void);

unsigned *cdecl AreYouLast(long far *vecTable,unsigned resPSP)
{
    register i;
    unsigned *blocks;
    struct MCB huge *mcb;
    extern unsigned _psp;                               // PSP della parte transiente

    _restorezero();                    // ripristina i vettori presi da startup code
    *(blocks = (unsigned *)malloc(sizeof(unsigned))) = NULL;
    for(i = 0; i < 0x22; i++)
        if(vecTable[i] != (long)getvect(i))                 // il vettore 0x22 non e'
            return(NULL);                                 // testato in quanto appare
    for(i = 0x23; i < 256; i++)                             // privo di significato
        if(vecTable[i] != (long)getvect(i))
            return(NULL);
    mcb = (struct MCB huge *)MK_FP(resPSP-1,0);               // MCB parte residente
    mcb = (struct MCB huge *)MK_FP(mcb->psp+mcb->dim,0);           // MCB successivo
    if(mcb->psp != _psp)
        return(NULL);
    mcb = (struct MCB huge *)MK_FP(getfirstmcb(),0);                // primo MCB DOS
    i = 2;
    do {
        if(mcb->psp == resPSP) {
            if(!(blocks = (unsigned *)realloc(blocks,i*sizeof(unsigned))))
                return(NULL);
            blocks[i-2] = FP_SEG(mcb)+1;           // MCB allocato alla parte resid.
            blocks[i-1] = NULL;
            ++i;
        }
    } while(mcb->pos != LASTBLOCK);
    return(blocks);
}
```

La `AreYouLast()` accetta come parametri il puntatore alla copia della tavola dei vettori creata in fase di installazione e l'indirizzo di segmento del PSP della parte residente[311].

[311] Meglio ripetere ancora una volta che si tratta di dati salvati in fase di installazione nello spazio riservato da una o più funzioni fittizie. Va ricordato che la parte del TSR residente in memoria e la parte transiente invocata in un secondo tempo, pur essendo due parti di un medesimo programma, si comportano in realtà come se fossero due programmi separati e indipendenti.

Il primo parametro è utilizzato per confrontare l'attuale tavola dei vettori con quella generata dall'installazione del TSR (si ipotizza che il TSR abbia eseguito la `_restorezero()` prima di copiare la tavola dei vettori). Se le due tavole sono identiche (a meno del vettore dell'int 22h, non significativo), la parte residente potrebbe effettivamente essere l'ultimo TSR installato, dal momento che nessuno ha modificato i vettori dopo la sua installazione: per avere un maggiore grado di sicurezza occorre verificare lo stato della catena dei MCB.

Tramite il secondo parametro, la `AreYouLast()` calcola l'indirizzo del Memory Control Block relativo al PSP della parte residente e lo assegna al puntatore a struttura MCB (vedere pag. 194). Questo è dichiarato `huge` in quanto la normalizzazione automatica[312] garantita dal compilatore consente di trascurare la parte offset del puntatore: essa vale in ogni caso `0` dal momento che un MCB è sempre allineato a un indirizzo di paragrafo (e pertanto varia solo la parte segmento). Il meccanismo del test è semplice: se il MCB successivo a quello della parte residente è il MCB della parte transiente (lo si può verificare mediante il campo `psp` della struttura), si può ragionevolmente supporre che la porzione residente sia proprio l'ultimo TSR installato, e si può allora procedere ad individuare tutti i blocchi di RAM ad essa allocati.

La variabile `blocks` è un puntatore ad `unsigned`: esso punta al primo elemento di un array di `unsigned int`, ciascuno dei quali è, a sua volta, l'indirizzo di segmento di un blocco di memoria appartenente alla porzione residente del TSR[313]; l'ultimo elemento dell'array vale sempre `NULL`. L'indirizzo del primo MCB presente in RAM è ottenuto mediante la `getfirstmcb()` (vedere pag. 194); la ricerca dei MCB si basa su un ciclo ripetuto fino ad incontrare l'ultimo blocco di RAM. L'algoritmo applicato ad ogni MCB è il seguente: se il campo `psp` è identico al parametro `resPSP`, allora il blocco appartiene alla parte transiente e si aggiorna l'array degli indirizzi dei blocchi da liberare per passare poi al successivo MCB.

La `AreYouLast()` restituisce `NULL` se la parte residente non sembra essere l'ultimo TSR installato o in caso di errore di allocazione della memoria. In caso di successo la `AreYouLast()` restituisce l'indirizzo dell'array di indirizzi da passare a `freemem()` per liberare tutti i blocchi di memoria allocati al TSR.

Si è detto che la `AreYouLast()` consente di valutare se sussistano le condizioni per disinstallare il TSR con sicurezza quasi assoluta: resta da chiarire il significato del "quasi".

Al proposito va ricordato che esistono prodotti software[314], studiati in particolare per macchine dotate di processore 80286, 80386 o superiori che consentono di rimappare agli indirizzi compresi tra `A0000h` e `FFFFFh` (cioè tra i 640 Kb e il Mb) una parte della memoria espansa installata: ciò equivale a rendere una zona addizionale di RAM, detta Upper Memory (vedere pag. 198), direttamente indirizzabile attraverso i registri della CPU (infatti una coppia di registri a 16 bit, rispettivamente segmento ed offset, è in grado di esprimere il numero `FFFF:000F` quale massimo indirizzo[315]); è pratica normale utilizzare proprio questa area di RAM per i programmi residenti (onde evitare di sottrarre spazio alle applicazioni nei 640 Kb di memoria convenzionale). Appare chiaro, a questo punto, che se il TSR da disinstallare è residente in memoria convenzionale e nella Upper Memory risiedono programmi installati successivamente (o viceversa), questi non possono essere individuati dalla `AreYouLAst()`, perché per

---

[312] I puntatori `huge` sono automaticamente normalizzati dal compilatore. Ciò significa che la loro parte offset (i 16 bit meno significativi) contiene esclusivamente lo spiazzamento all'interno del paragrafo referenziato dalla parte segmento (i 16 bit più significativi). Vedere pag. 21 e seguenti.

[313] I blocchi possono essere più di uno qualora la parte residente gestisca dei buffer.

[314] Anche il DOS, a partire dalla versione 5.0, è dotato di tale capacità. Vedere pag. 198.

[315] La parte segmento di un puntatore `far`, costruito mediante due registri a 16 bit, esprime indirizzi di paragrafo (ogni unità conta 16 byte). L'indirizzo `FFFF:000F` può dunque essere scritto `FFFFFh`, cifra equivalente al Mb. L'offset non può essere ulteriormente incrementato, poiché un offset di `10h` (`16` decimale) rappresenta in realtà un incremento di uno della parte segmento, con offset zero. Vedere anche pag. 16.

il DOS la memoria disponibile termina in ogni caso all'indirizzo `A0000h` e l'ultimo MCB è quello che controlla il blocco di RAM che termina a quell'indirizzo. In tali casi il controllo effettuato sui vettori è decisivo, e dovrebbe rendere la `AreYouLast()` a prova di bomba, con la sola eccezione di buffer allocati nella Upper Memory dal TSR stesso. A scopo di chiarezza si dà qualche cenno sulla creazione di una copia della tavola dei vettori.

La quantità di RAM necessaria alla porzione residente risulta incrementata di 1 Kb (265 vettori di 4 byte ciascuno), pertanto la funzione jolly deve riservare i 1024 byte necessari (è banale dirlo, ma comunque...). In secondo luogo il salvataggio della tavola va effettuato dopo avere agganciato tutti i vettori necessari al TSR e dopo avere ripristinato quelli modificati dallo startup code (pag. 105): occorre pertanto invocare esplicitamente la `_restorezero()`. Ciò nonostante la `keep()` può ancora essere utilizzata per terminare il programma[316]. Il salvataggio della tavola può essere realizzato mediante la `getvect()` o, per una maggiore efficienza, tramite indirezioni di puntatori o, ancora, con la funzione di libreria `_fmemcpy()`. Anche in fase di disinstallazione, come si è visto, è necessario invocare la `_restorezero()` prima di effettuare il controllo[317]; inoltre, tutti i gestori di interrupt del TSR devono essere attivati (a meno che l'eventuale routine di disattivazione e riattivazione del TSR provveda anche a modificare opportunamente la copia della tavola dei vettori o si basi semplicemente su flag). La routine di installazione potrebbe presentare la seguente parte terminale:

```
    ....
    _restorezero();
    _fmemcpy((long far *)startUpVectors,MK_FP(0,0),256*sizeof(void far *));
    _fmemcpy(((long far *)startUpVectors)+0x23,MK_FP(_psp,0x0E),
                                                2*sizeof(void far *));
    keep(code,resparas);
}
```

Il nome della funzione fittizia `startUpVectors()` è forzato a puntatore a `long` e viene gestito come array; `MK_FP(0,0)` (pag. 24) punta alla tavola dei vettori (`0:0`). Si noti che i vettori, pur essendo a rigore, puntatori a funzioni (di tipo `interrupt`), sono qui gestiti come `long int` per migliorare la leggibilità del listato, senza che ciò comprometta la correttezza tecnica del codice, trattandosi in entrambi i casi di dati a 32 bit.

E' necessario scrivere nella copia della tavola dei vettori gli indirizzi dell'int 23h e 24h, prelevandoli dal PSP del programma in fase di installazione, dal momento che il DOS li copia nella tavola dei vettori quando il programa termina (vedere pag. 309 e pag. 310).

ALCUNI ESEMPI PRATICI

Vediamo ora un esempio di programma: si tratta di un semplice TSR, che aggancia l'int 2Fh (per utilizzarlo come canale di comunicazione) e l'int 21h, mascherando il servizio 11h[318] di quest'ultimo. Per disinstallare il TSR occorre lanciarlo nuovamente con un asterisco come parametro sulla command line. E' prevista anche la possibilità di disattivazione e riattivazione, sempre tramite nuova invocazione da DOS prompt (ma con un punto esclamativo quale parametro).

---

[316] La `keep()` invoca a sua volta la `_restorezero()`, la quale copia nuovamente i vettori originali nella tavola: l'operazione è inutile e potrebbe risultare dannosa, nel caso in cui il programma TSR abbia installato nuovi vettori per gli interrupt 00h, 04h, 05h e 06h.

[317] Un listato di `_restorezero()` si trova a pag. 414.

[318] `FINDFIRST` mediante File Control Block. Modificando questo servizio si modifica, tra l'altro, il comportamento del comando `DIR`, che lo utilizza.

```
/********************

    PROVATSR.C - Barninga_Z! - 1991

        Esempio di TSR. "Azzoppa" il comando DIR, che, dopo la
        installazione, fornisce risultati erratici. Invocare con
        un punto esclamativo come parametro per ripristinare il
        comando DIR, lasciando il TSR in RAM. Per riattivare il
        TSR lanciarlo nuovamente, sempre con un '!' come parametro
        sulla riga di comando. Per disinstallarlo, lanciarlo con
        un asterisco come parametro.

    Compilabile con TURBO C++ 1.01

        tcc -Tm2 -mx provatsr.C

    NOTA: -mx rappresenta il modello di memoria: i modelli validi
          sono tiny (-mt) small (-ms), medium (-mm), compact (-mc)
          e large (-ml). Per il modello huge (-mh) occorre
          introdurre una istruzione inline assembly POP DS prima
          di ogni POP BP nei due gestori di interrupt, a causa
          del salvataggio automatico di DS generato dal compilatore
          in ingresso alla funzione.

********************/
#pragma  inline
#pragma  -k+           // il codice e' scritto per TURBO C++ 1.01 (vedere pag. 173)

#include <stdio.h>
#include <dos.h>

#define  old21h ((void(interrupt *)())(*(((long far *)ResDataPtr)+0)))
#define  old2Fh ((void(interrupt *)())(*(((long far *)ResDataPtr)+1)))
#define  Res21h ((void(interrupt *)())(*(((long far *)ResDataPtr)+2)))
#define  Res2Fh ((void(interrupt *)())(*(((long far *)ResDataPtr)+3)))
#define  ResPSP                   (*(((unsigned far *)ResDataPtr)+8))
#define  ResStatus                (*(((unsigned far *)ResDataPtr)+9))

#define  ASMold21h      GData
#define  ASMold2Fh      GData+4
#define  ASMResStatus   GData+18

#define  FCB_FFIRST     0x11               /* servizio dell'int 21h da annullare */
#define  HEY_YOU        0xAF                          /* byte di identificazione */
#define  HERE_I_AM      0xFDA3                           /* risposta: installato */
#define  OFF            0
#define  ON             1
#define  HANDSHAKE      0x00                    /* serv. int 2Fh: riconoscimento */
#define  UNINSTALL      0x01                  /* serv. int 2Fh: disinstallazione */
#define  SW_ACTIVE      0x02              /* serv. int 2Fh: attivaz./disattivaz. */
#define  UNINST_OPT     '*'                /* opzione cmd line: disinstallazione */
#define  ACTIVE_OPT     '!'                /* opzione cmd line: attiva/disattiva */

void GData(void);                                   /* prototipo funzione jolly */

void far new21h(void)                                        /* handler int 21h */
{
    asm {
        pop bp;                     /* pulisce stack - attenzione al compilatore */
        cmp ah,FCB_FFIRST;                          /* se il servizio non e' 11h */
        jne CHAIN;                           /* allora salta all'etichetta CHAIN */
        iret;                                              /* altrimenti ritorna */
    }
CHAIN:
```

```
    asm jmp dword ptr ASMold21h;                   /* concatena gestore originale */
}

void far new2Fh(void)                                         /* handler int 2Fh */
{
    asm {
        pop bp;                                                 /* pulisce stack */
        cmp ah,HEY_YOU;                      /* se non e' PROVATSR che chiama... */
        jne CHAIN;                                              /* allora salta */
        cmp al,HANDSHAKE;                  /* se non e' il test di installazione */
        jne NEXT1;                                              /* allora salta */
        mov ax,HERE_I_AM;                /* altrimenti risponde che e' presente */
        iret;
    }
NEXT1:
    asm {
        cmp al,UNINSTALL;                /* se e' richiesta la disinstallazione */
        je ANSWER;                                              /* allora salta */
        cmp al,SW_ACTIVE;            /* se non richiesta attivaz./disattivaz. */
        jne CHAIN;                                              /* allora salta */
    }
ANSWER:                /* disinstallazione e switch attivazione confluiscono qui */
    asm {
        mov ax,offset GData;                 /* restituisce in DX:AX l'indirizzo */
        mov dx,seg GData;                                    /* far di GData() */
        iret;
    }
CHAIN:
    asm jmp dword ptr ASMold2Fh;                   /* concatena gestore originale */
}

void GData(void)                                              /* funzione jolly */
{
    asm db 15 dup(0);                    /* 4+4+4+4+2+2 bytes per i dati meno 5 */
}

void releaseEnv(void)                                  /* libera l'environment */
{
    extern unsigned _envseg;               /* _envseg e' definita nello startup */

    if(freemem(_envseg))
        puts("Cannot release environment: memory error.");
}

long AreYouThere(char service)                 /* gestisce comunicazione con TSR */
{
    long RetValue;
    union REGS regs;

    regs.h.ah = HEY_YOU;
    regs.h.al = service;
    RetValue = (long)int86(0x2F,&regs,&regs);
    RetValue |= ((long)regs.x.dx) << 16;                       /* RetValue = DX:AX */
    return(RetValue);
}

void uninstall(void)                           /* gestisce disinstallazione del TSR */
{
    void far *ResDataPtr;

    ResDataPtr = (void far *)AreYouThere(UNINSTALL);
    setvect(0x21,old21h);
    setvect(0x2F,old2Fh);
    if(freemem(ResPSP))
```

```
        puts("Cannot remove from memory: memory error.");
    else
        puts("Uninstalled: vectors restored and RAM freed up.");
}

void install(void)                                          /* installa il TSR */
{
    void far *ResDataPtr;

    ResDataPtr = (void far *)GData;
    ResPSP = _psp;
    ResStatus = ON;
    Res21h = (void(interrupt *)())new21h;
    Res2Fh = (void(interrupt *)())new2Fh;
    asm cli;
    old21h = getvect(0x21);
    old2Fh = getvect(0x2F);
    setvect(0x21,Res21h);
    setvect(0x2F,Res2Fh);
    asm sti;
    releaseEnv();
    puts("Installed: 'DIR' command not reliable. Pass * or !.");
    keep(0,FP_SEG(releaseEnv)+FP_OFF(releaseEnv)/16+1-_psp);
}

void setStatus(void)                /* gestisce switch attivazione/disattivazione */
{
    void far *ResDataPtr;

    ResDataPtr = (void far *)AreYouThere(SW_ACTIVE);
    if(ResStatus) {
        setvect(0x21,old21h);
        ResStatus = OFF;
        puts("Deactivated: still present in RAM.");
    }
    else {
        setvect(0x21,Res21h);
        ResStatus = ON;
        puts("Reactivated: already present in RAM.");
    }
}

void main(int argc,char **argv)
{
    if((unsigned)AreYouThere(HANDSHAKE) == HERE_I_AM)
        if(argc > 1)                                      /* TSR gia' installato */
            switch(*argv[1]) {                           /* passato un parametro */
                case UNINST_OPT:                                   /* param = * */
                    uninstall();                                 /* disinstalla */
                    break;
                case ACTIVE_OPT:                                   /* param = ! */
                    setStatus();                             /* attiva/disattiva */
                    break;
                default:                                     /* altro parametro */
                    puts("Unknown option.");
            }
        else                                                /* nessun parametro */
            puts("Not Installed: already present in RAM.");
    else                          /* TSR non ancora installato (ignora parametri) */
        install();
}
```

I numerosi commenti inseriti nel listato eliminano la necessità di descrivere nel dettaglio la struttura e il flusso logico dell'intero programma: ci limitiamo a sottolinearne le particolarità più interessanti, precisando sin d'ora che è stato contenuto quanto più possibile il ricorso allo inline assembly, utilizzando il linguaggio C anche laddove ciò penalizza in qualche misura la compattezza e l'efficienza del codice compilato.

La porzione transiente del programma comunica con quella residente mediante la `AreYouThere()`, che invoca l'int 2Fh e restituisce, sotto forma di `long int` il valore restituito dall'interrupt nella coppia di registri `DX:AX`. Le funzioni che di volta in volta chiamano la `AreYouThere()` forzano secondo necessità, con opportune operazioni di cast, il tipo di detto valore: la `main()` ne considera solamente i 16 bit meno significativi, in quanto essi rappresentano la parola d'ordine restituita dal servizio 00h dell'int 2Fh (questo servizio non utilizza il registro `DX`). Al contrario, la `uninstall()` e la `setStatus()` effettuano un cast a `(void far *)`, per gestire correttamente, anche dal punto di vista formale, il puntatore `ResDataPtr`, il quale è, per il compilatore C, un puntatore a dati di tipo indeterminato: le macro definite in testa al codice permettono di utilizzarlo, nascondendo cast a volte complessi, per scrivere e leggere i dati globali direttamente nel buffer di 20 byte[319] ad essi riservato dalla la funzione jolly `GData()`.

Presentiamo, di seguito, un secondo esempio: si tratta di una versione semplificata del programma precedente, in quanto mancante della capacità di attivarsi e disattivarsi. Differenze sostanziali si riscontrano, però, anche nel metodo utilizzato per la disinstallazione: tutte le operazioni vengono svolte dal gestore dell'int 2Fh. Si tratta dunque di un algoritmo applicabile quando si desideri poter richiedere la disinstallazione mediante hotkey.

```
/********************

    PROV2TSR.C - Barninga_Z! - 1991

        Esempio di TSR. "Azzoppa" il comando DIR, che, dopo la
        installazione, fornisce risultati erratici. Invocare con
        un asterisco come parametro per  disinstallarlo.

    Compilabile con TURBO C++ 1.01

        tcc -Tm2 -mx prov2tsr.C

    NOTA: -mx rappresenta il modello di memoria: i modelli validi
          sono tiny (-mt) small (-ms), medium (-mm), compact (-mc)
          e large (-ml). Per il modello huge (-mh) occorre
          introdurre una istruzione inline assembly POP DS prima
          di ogni POP BP nei due gestori di interrupt, a causa
          del salvataggio automatico di DS generato dal compilatore
          in ingresso alla funzione.

********************/
#pragma inline
#pragma  -k+           // il codice e' scritto per TURBO C++ 1.01 (vedere pag. 173)

#include <stdio.h>
#include <dos.h>

#define  old21h         ((void(interrupt *)())(*(((long *)GData)+0)))
```

[319] I 15 esplicitamente definiti dall'istruzione `DB`, più i 5 di codice generato automaticamente dal compilatore (opcodes per `PUSH BP`; `MOV BP,SP`; `POP BP`). L'ordine di memorizzazione dei dati è: vettore originale dell'int 21h (doubleword); vettore originale dell'int 2Fh (doubleword); indirizzo di `new21h()` (doubleword); indirizzo di `new2Fh()` (doubleword); indirizzo di segmento del PSP del TSR (word); byte di stato (attivato/disattivato) del TSR (word).

```
#define  old2Fh         ((void(interrupt *)())(*(((long *)GData)+1)))
#define  ResPSP         (*(((unsigned *)GData)+4))

#define  ASMold21h      GData
#define  ASMold2Fh      GData+4
#define  ASMResPSP      GData+8

#define  FCB_FFIRST     0x11
#define  HEY_YOU        0xAF
#define  HERE_I_AM      0xFDA3
#define  HANDSHAKE      0x00
#define  UNINSTALL      0x01
#define  UNINST_OPT     '*'

void GData(void);

void far new21h(void)
{
    asm {
        pop bp;
        cmp ah,FCB_FFIRST;
        jne CHAIN;
        iret;
    }
CHAIN:
    asm jmp dword ptr ASMold21h;
}

void far new2Fh(void)
{
    asm {
        cmp ah,HEY_YOU;
        jne CHAIN;
        cmp al,HANDSHAKE;
        jne NEXT1;
        mov ax,HERE_I_AM;
        pop di;                  /* pulisce stack: nella disinstallazione sono */
        pop si;             /* usati SI e DI, pertanto il compilatore il salva */
        pop bp;                               /* automaticamente (insieme a BP) */
        iret;
    }
NEXT1:
    asm {
        cmp al,UNINSTALL;
        jne CHAIN;
        push ds;
        push es;
        cld;              /* operazioni su stringhe di bytes incrementano SI DI */
        cli;                                                 /* stop interrupt */
        mov ax,seg GData;                   /* carica AX con il segmento di GData */
        mov ds,ax;                                          /* e SI con l'offset */
        mov si,offset GData;                /* DS:SI punta al primo dato in GData */
        xor ax,ax;
        mov es,ax;                         /* carica ES:DI per puntare al vettore */
        mov di,0x21*4;                   /* dell'int 21h nella tavola dei vettori */
        mov cx,2;                /* poi copia due words da GData a tavola vettori */
        rep movsw;                 /* e carica DI per puntare al vettore dell'int */
        mov di,0x2F*4;                    /* 2Fh nella tavola dei vettori e copia */
        mov cx,2;                       /* altre 2 words (SI e' stato incrementato */
        rep movsw;                          /* di due grazie all'istruzione CLD) */
        mov ax,word ptr ASMResPSP;              /* carica AX con ind seg PSP TSR */
        dec ax;                   /* trova l'ind. di seg. del MCB del PSP del TSR */
        mov es,ax;              /* carica ES:DI per puntare al campo "owner" del */
        mov di,1;                                           /* MCB (ha offset 1) */
```

```
        xor ax,ax;                   /* azzera AX e lo copia nel campo "owner" del */
        stosw;                      /* MCB per disallocare la RAM assegnata al TSR */
        sti;                                             /* riabilita interrupts */
        pop es;                       /* pulisce stack: vedere sopra per SI e DI */
        pop ds;
        pop di;
        pop si;
        pop bp;
        iret;
    }
CHAIN:
    asm {
        pop di;                       /* pulisce stack: vedere sopra per SI e DI */
        pop si;
        pop bp;
        jmp dword ptr ASMold2Fh;                   /* concatena gestore originale */
    }
}

void GData(void)
{
    asm db 5 dup(0);                  /* spazio per dati (4+4+2 meno i 5 opcodes) */
}

void releaseEnv(void)
{
    extern unsigned _envseg;

    if(freemem(_envseg))
        puts("Cannot release environment: memory error.");
}

unsigned AreYouThere(char service)
{
    union REGS regs;

    regs.h.ah = HEY_YOU;
    regs.h.al = service;
    return(int86(0x2F,&regs,&regs));
}

void install(void)
{
    ResPSP = _psp;
    asm cli;
    old21h = getvect(0x21);
    old2Fh = getvect(0x2F);
    setvect(0x21,(void(interrupt *)())new21h);
    setvect(0x2F,(void(interrupt *)())new2Fh);
    asm sti;
    releaseEnv();
    puts("Installed: 'DIR' not reliable. Pass * to uninstall.");
    keep(0,FP_SEG(releaseEnv)+FP_OFF(releaseEnv)/16+1-_psp);
}

void uninstall(void)
{
    AreYouThere(UNINSTALL);
    puts("Uninstalled: vectors restored and RAM freed up.");
}

void main(int argc,char **argv)
{
    if(AreYouThere(HANDSHAKE) == HERE_I_AM)
```

```
        if(argc > 1 && argv[1][0] == UNINST_OPT)
            uninstall();
        else
            puts("Not Installed: already active in RAM.");
    else
        install();
}
```

Come si può facilmente vedere, la `new2Fh()` di `PROV2TSR.C` risulta più complessa rispetto a quella di `PROVATSR.C`: in effetti essa svolge tutte le operazioni necessarie alla disinstallazione. La coppia di registri `DS:SI` è caricata per puntare a `GData()`, cioè al primo dei dati globali (il vettore originale dell'int 21h); la coppia `ES:DI` punta invece alla tavola dei vettori, ed in particolare al vettore dell'int 21h[320]. Il vettore originale è ripristinato copiando 2 word (4 byte) dalla `GData()` residente alla tavola. Con la medesima tecnica avviene il ripristino del vettore dell'int 2Fh, dopo opportuno aggiornamento del puntatore alla tavola dei vettori. La RAM è disallocata agendo direttamente sul Memory Control Block del PSP del TSR, il cui indirizzo è ricavato decrementando di uno quello del PSP (il MCB occupa infatti i 16 byte che lo precedono). Allo scopo basta azzerare la coppia di byte (ad offset `1` nel MCB) che esprime l'indirizzo del PSP del programma "proprietario" del blocco di memoria.

E' stato evitato l'uso dei servizi che l'int 21h rende disponibili per le operazioni ora descritte: si tratta di una scelta effettuata a titolo di esempio, più che di una precauzione volta a rendere massima la sicurezza operativa del TSR (sui problemi legati all'int 21h si veda pag.), in quanto l'int 2Fh è invocato in modo sincrono dalla parte transiente, la quale "conosce" lo stato del sistema proprio perché essa stessa lo determina in quel momento. Ciò rende possibile, senza particolari rischi, l'espletamento di quelle operazioni che devono essere effettuate necessariamente via int 21h (chiusura di file, etc.). Quando la disinstallazione non sia richiesta mediante un secondo lancio del programma ma attraverso la pressione di uno hotkey, è necessario prendere alcune precauzioni. La parte transiente deve rilevare lo hotkey attraverso il gestore dell'interrupt di tastiera (int 09h o int 16h): questo si limita modificare lo stato di un flag che viene ispezionato quando il sistema è stabile, per avere la garanzia di procedere in condizioni di sicurezza. Test e disinstallazione possono essere effettuati, ad esempio, nel gestore dell'int 28h o del timer (int 08h). Presentiamo un nuovo listato della `new2Fh()`: questa versione utilizza l'int 21 in luogo degli accessi diretti alla tavola dei vettori e al MCB:

```
void far new2Fh(void)
{
    asm {
        cmp ah,HEY_YOU;
        jne CHAIN;
        cmp al,HANDSHAKE;
        jne NEXT1;
        mov ax,HERE_I_AM;
        pop bp;
        iret;
    }
NEXT1:
    asm {
        cmp al,UNINSTALL;
        jne CHAIN;
        push ds;
        push es;
        cli;
        mov ax,word ptr GData+2;                    /* carica DS e DX in modo che */
        mov ds,ax;                            /* DS:DX punti al vettore originale */
```

[320] La tavola dei vettori si trova all'indirizzo `0:0`. Dal momento che ogni vettore è un indirizzo `far`, ed occupa pertanto 4 byte, per ottenere l'offset di un vettore all'interno della tavola è sufficiente moltiplicare il numero dell'interrupt per 4.

```
        mov dx,word ptr GData;                                /* dell'int 21h */
        mov ax,0x2521;            /* serv. set interrupt vector (vettore = 21h) */
        int 0x21;
        mov ax,word ptr GData+6;                   /* carica DS e DX in modo che */
        mov ds,ax;                         /* DS:DX punti al vettore originale */
        mov dx,word ptr GData+4;                              /* dell'int 2Fh */
        mov ax,0x252F;            /* serv. set interrupt vector (vettore = 2Fh) */
        int 0x21;
        mov ax,word ptr GData+8;                  /* carica in ES l'indirizzo di */
        mov es,ax;                    /* segmento del PSP della parte residente */
        mov ah,0x49;                         /* servizio free allocated memory */
        int 0x21;
        sti;
        pop es;                                               /* pulisce stack */
        pop ds;
        pop bp;
        iret;
    }
CHAIN:
    asm {
        pop bp;
        jmp dword ptr ASMold2Fh;                   /* concatena gestore originale */
    }
}
```

Gli offset di volta in volta sommati a `GData` (il nome della funzione ne rappresenta l'indirizzo, cioè punta alla funzione stessa: vedere pag. 93) tengono conto della modalità backwords[321] di memorizzazione dei dati.

In `PROV2TSR.C` può essere utilizzata la `AreYouLast()`: essa deve essere chiamata dalla `uninstall()`, la quale solo se il valore restituitole non fosse `NULL` disinstalla il TSR ripristinando i vettori e invocando la `freemem()` per ogni elemento dell'array, eccettuato, naturalmente, il `NULL` terminale. Si noti che la `AreYouLast()` presenta caratteristiche più avanzate di quanto necessiti effettivamente a `PROV2TSR.C`, in quanto esso non gestisce buffer e dunque vi è un solo blocco di memoria allocato alla sua parte residente: quello che la contiene.

Ancora un TSR: questa volta è un programma (quasi) serio. Si tratta di uno screen saver, cioè di un programma che interviene, quando tastiera e mouse non sono sollecitati per un dato intervallo temporale, cancellando il video al fine di prevenirne una eccessiva usura; non appena è premuto un tasto o mosso il mouse, il contenuto del video è ripristinato e la sessione di lavoro può proseguire normalmente (l'operatività della macchina non viene mai bloccata).

```
/******************************************************************************

    SSS.C - Barninga Z! 1994

    Sample Screen Saver. TSR, opera validamente in modo testo 80x25.
    La stringa passata sulla command line viene visualizzata in posizioni
    random a video; se invocato con * come parametro quando residente, si
    disinstalla. Il tempo di attesa per entrare in azione e' definito in
    ticks di clock dalla costante manifesta MAXTICKS.

    Compilato sotto Borland C++ 3.1
```

[321] Backwords è un termine nato dall'assonanza con backwards (all'indietro) e significa, letteralmente, a parole rovesciate. In effetti, tutti i dati numerici sono scritti in RAM a rovescio, in modo tale, cioè, che a indirizzo inferiore corrisponda la parte meno significativa della cifra. Ad esempio, il numero `0x11FF024A` (che potrebbe rappresentare un puntatore far a `11FF:024A`) viene scritto in RAM in modo da occupare 4 byte nel seguente ordine: `4A 02 FF 11`.

```
    bcc -k- sss.c

******************************************************************************/
#pragma  inline                // usato inline assembly!

#pragma  option -k-            // evita la standard stack frame; serve nelle
                               // f() fittizie che definiscono i dati per
                               // gestire meglio lo spazio. Ovvio bisogna
                               // tenerne conto anche nelle f() eseguibili che
                               // usano l'inline assembly

#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <dir.h>
#include <time.h>              // per randomize()
#include <stdlib.h>            // per randomize() e rand()

#define  PRG            "SSS"
#define  REL            "1.0"
#define  YEAR           "94"

// costanti manifeste per la gestione del video e del timer

#define  MAXTICKS       5460        // 5 minuti, per default
#define  MAXSPEED       18          // muove banner: 1 secondo, per default
#define  MAXBANNER      40          // max. lunghezza banner; se l'utente
                                       // specifica un banner lungo MAXBANNER,
                                       // il NULL copre RET di resBanner(): OK!
#define  MAXPOS         100         // punti del percorso del banner
#define  DEFAULTBLANK   0x0720      // blank di default
#define  DEFAULTVIDEO   0xB800      // segmento video (default)
#define  DEFAULTCOLS    80          // col. video default
#define  DEFAULTROWS    25          // righe video default
#define  DEFAULTPAGE    0           // pagina video default

// costanti manifeste per i servizi dell'int 2F; 0 e' il servizio di
// riconoscimento, per evitare doppie installazioni in ram del TSR; 1 e' il
// servizio di disinstallazione del TSR dalla memoria

#define  HANDSHAKE      0x00   // servizio di scambio parola d'ordine
#define  UNINSTALL      0x01   // servizio di disinstallazione
#define  UNINST_OPT     "*"    // da passare sulla comand line per richiedere
                               // la disinstallazione del TSR
#define  HEY_YOU        0xE1   // parola d'ordine di riconoscimento
#define  HERE_I_AM      0xB105 // risposta alla parola d'ordine

typedef void DUMMY;            // tutte le f() fittizie che definiscono dati
                               // sono ovviamente void f(void); la typedef
                               // consente di evidenziare che non sono vere f()

// prototipi delle funzioni

void far new09h(void);
void far new10h(void);
void far new1Ch(void);
void far new2Fh(void);
void far new33h(void);
void _saveregs animate(void);
void _saveregs blankVideo(void);
void _saveregs restoreVideo(void);
int releaseEnv(void);
unsigned areYouThere(char service);
void initializeOffsets(void);
```

```
void install(char *banner);
void uninstall(void);
void main(int argc,char **argv);

/*--------------------------------------------------------------------------*/

// inizia qui la parte residente del TSR: dapprima si riserva spazio per i dati
// mediante alcune finte f() che devono solo "ingombrare" lo spazio necessario,
// poi sono definite tutte le funzioni che lavorano mentre il programma e'
// residente.

// Gruppo delle f() fittizie, usate come contenitori di dati. Equivalgono a
// variabili globali (il nome della funzione puo' essere usato come il nome di
// una variabile, applicando l'opportuno cast) ma si ha la garanzia che lo
// spazio e' allocato in modo statico esattamente dove si vuole: essendo esse
// definite in testa al sorgente, sappiamo che subito dopo lo startup code
// del programma ci sono i dati necessari al TSR quando e' residente in RAM.
// Grazie ad esse, le f() residenti non usano variabili globali o statiche,
// rendendo cosi' possibile limitare l'ingombro in memoria del TSR

DUMMY ticksToWait(DUMMY)       // contatore da decrementare per l'attivazione
{
    asm dw MAXTICKS;                   // max 65535, quasi 1 ora
}

DUMMY ticksForSpeed(DUMMY)     // contatore per la velocita' di animazione
{
    asm dw 0;                          // 0: 1^ banner istantaneo!
}

DUMMY resPSP(DUMMY)            // indirizzo di segmento del PSP del STR
{
    asm dw 0;
}

DUMMY old09h(DUMMY)            // vettore originale int 09h
{
    asm dd 0;
}

DUMMY old10h(DUMMY)            // vettore originale int 10h
{
    asm dd 0;
}

DUMMY old1Ch(DUMMY)            // vettore originale int 1Ch
{
    asm dd 0;
}

DUMMY old2Fh(DUMMY)            // vettore originale int 2Fh
{
    asm dd 0;
}

DUMMY old33h(DUMMY)            // vettore originale int 33h
{
    asm dd 0;
}

DUMMY videoBuf(DUMMY)          // spazio per il salvataggio del video
{
    asm dw DEFAULTCOLS*DEFAULTROWS dup(0);
}
```

```
DUMMY savedRow(DUMMY)           // riga alla quale si trovava il cursore
{
    asm db 0;
}

DUMMY savedCol(DUMMY)           // colonna alla quale si trovava il cursore
{
    asm db 0;
}

DUMMY saverActive(DUMMY)        // flag che segnala se il saver e' attivo
{
    asm db 0;
}

DUMMY restoreFlag(DUMMY)        // flag che segnala di restorare il video
{
    asm db 0;
}

DUMMY inInt10h(DUMMY)           // flag che evita ricorsione dell'int 10h
{
    asm db 0;
}

DUMMY resBanner(DUMMY)          // stringa per animazione video
{
    asm db MAXBANNER dup(0);
}

DUMMY resBannerLen(DUMMY)       // lunghezza banner: per comodita'
{
    asm dw 0;
}

DUMMY resBannerOffs(DUMMY)      // offsets per scrivere il banner
{
    asm dw MAXPOS dup(0);
}

DUMMY currBannerOff(DUMMY)      // contatore per l'array di posizioni
{
    asm dw 0;
}

// nuovo gestore dell'int 09h; non serve dichiararlo interrupt perche' non
// usa nessun registro (eccetto AX, salvato e rispristinato da noi stessi), ma
// deve comunque terminare con una iret. Gli underscores davanti ai nomi
// servono perche' il compilatore C li aggiunge a tutti i simboli mentre
// l'assembler no. L'uso dell'inline assembly consente di ottenere una funzione
// efficientissima e molto compatta.

void far new09h(void)
{
    asm mov byte ptr _restoreFlag,1;
    asm cmp byte ptr _saverActive,1;  // se e' stato premuto un tasto e lo s.s.
    asm je EAT_KEY;                   // era attivo, la battuta deve sparire!
    asm jmp dword ptr _old09h;        // se no, lasciamo al vecchio gestore!
  EAT_KEY:
    asm push ax;                      // salva l'unico registro usato
    asm in al,0x60;                   // fingiamo che interessi lo scancode...
    asm in al,0x61;                   // legge lo stato della tastiera
    asm mov ah,al;                    // lo salva
```

```
    asm or al,0x80;                  // setta il bit "enable keyboard"
    asm out 0x61,al;                 // lo scrive sulla porta di controllo
    asm xchg ah,al;                  // riprende stato originale della tast.
    asm out 0x61,al;                 // e lo riscrive
    asm mov al,0x20;                 // manda il segnale di fine Interrupt
    asm out 0x20,al;                 // al controllore dell'8259
    asm pop ax;                      // rimettiamo le cose a posto!
    asm iret;          // e' pur sempre un gestore di interrupt! ma ATTENZIONE:
                       // la IRET puo' essere usata perche' la f() e' far e non
                       // c'e' la standard stack frame (opzione -k-).
}

// il gestore dell'int 10h e' qui per sicurezza. Dal momento che l'int 1Ch
// usa l'int 10h per pasticciare col cursore, se il timer chiede un interrupt
// proprio mentre e' servito un int 10h e proprio in quel tick di timer si
// azzera il contatore di attesa per lo screen saver e viene cosi' chiamata
// blankVideo(), si ha una ricorsione dell'int 10h. Le routines BIOS non sono
// rientranti e cio' significa un crash di sistema assicurato. Questo gestore
// dell'int 10h alza un flag in ingresso, invoca il gestore originale e al
// rientro da questo resetta il flag. Detto flag deve essere testato
// nell'int 1Ch: se e' 1 bisogna lasciar perdere tutto e rinviare.

void far new10h(void)
{
    asm mov byte ptr _inInt10h,1;
    asm pushf;
    asm call dword ptr old10h;
    asm mov byte ptr _inInt10h,0;
    asm iret;
}

// anche new1Ch() e' dichiarata far, per ottenere maggiore efficienza. Quel
// poco che fa e' implementato in inline assembly; i lavori complessi sono
// curati da alcune funzioni di servizio, che possono essere implementate in
// C grazie alla dichiarazione _saveregs. La f() e' strutturata come segue:
// Se lo screen saver e' attivo si controlla se e' premuto un tasto. In caso
// affermativo si ripristinano il video, il contatore dei ticks di attesa, il
// contatore dei ticks di permanenza del banner, il flag di richiesta di
// restore e il flag di screen saver attivo. In caso negativo si esegue la
// funzione animate(). Se invece lo screen saver non e' attivo, si controlla
// se e' stato premuto un tasto. In caso affermativo si ripristinano il flag
// di tasto premuto (richiesta restore) e il contatore dei ticks di attesa. In
// caso negativo si controlla se il tempo di attesa e' scaduto (contatore ticks
// di attesa = 0). Se lo e' si esegue blankVideo() e si setta il flag di screen
// saver attivo. Se non lo e' si decrementa il contatore dei ticks di attesa.

void far new1Ch(void)
{
    asm cmp byte ptr _inInt10h,1;
    asm jne OK_TO_WORK;
    asm jmp EXIT;
  OK_TO_WORK:
    asm cmp byte ptr _saverActive,1;
    asm jne NOT_ACTIVE;
    asm cmp byte ptr _restoreFlag,1;
    asm je RESTORE_VIDEO;
    animate();
    asm jmp EXIT;
  RESTORE_VIDEO:
    restoreVideo();
    asm mov byte ptr _saverActive,0;
    asm mov word ptr _ticksForSpeed,0;
    asm jmp RESTORE_STATUS;
  NOT_ACTIVE:
```

```
    asm cmp byte ptr _restoreFlag,1;
    asm je RESTORE_STATUS;
    asm cmp word ptr ticksToWait,0;
    asm je BLANK_VIDEO;
    asm dec word ptr _ticksToWait;
    asm jmp EXIT;
  BLANK_VIDEO:
    asm mov byte ptr _saverActive,1;
    blankVideo();
    asm jmp EXIT;
  RESTORE_STATUS:
    asm mov byte ptr _restoreFlag,0;
    asm mov word ptr _ticksToWait,MAXTICKS;
  EXIT:
    asm iret;
}

// new2Fh gestisce il dialogo con il TSR. Se AH contiene HEY_YOU, l'interrupt
// e' stato chiamato dalla porzione transiente del programma stesso, dopo che
// la parte residente e' gia' stata installata, altrimenti e' un altro
// programma e non sono fatti nostri. Se AL contiene HANDSHAKE, e' la parola
// d'ordine: bisogna rispondere HERE_I_AM, cosi' la parte transiente sa che
// il TSR e' gia' installato e non tenta di reinstallarlo. Se AL contiene
// UNINSTALL, la parte transiente sta chiedendo alla parte residente di
// disinstallarsi. L'operazione e' semplice: basta ripristinare i vettori di
// interrupt e liberare la memoria allocata al TSR.
// ATTENZIONE: new2Fh() e' dichiarata far per maggiore efficienza e per poter
// restituire valori nei registri della CPU alla routine chiamante. Percio' e'
// indispensabile salvare e ripristinare, se usati, alcuni registri importanti:
// DS, ES, SI, DI. SI e DI, se usati in righe di inline assembly, sono salvati
// automaticamente dal compilatore in testa alla funzione: ecco perche' sono
// estratti dallo stack con le POP senzxa che ci siano le PUSH corrispondenti!!

void far new2Fh(void)
{
    asm cmp ah,HEY_YOU;     // riconoscimento: HEY_YOU?
    asm je SSS_CALLING;
  CHAIN:                    // no, la chiamata non viene da SSS: lasciamo stare
    asm pop di;              // pulisce stack: nella disinstallazione sono
    asm pop si;              // usati SI e DI, pertanto il compilatore il salva
    asm jmp dword ptr old2Fh;    // concatena gestore originale
  SSS_CALLING:
    asm cmp al,HANDSHAKE;      // vediamo cosa vuole SSS transiente
    asm jne NEXT1;                // SSS vuole sapere se siamo qui
    asm mov ax,HERE_I_AM;
    asm jmp EXIT;
  NEXT1:
    asm cmp al,UNINSTALL;         // o forse SSS vuole la disinstallazione
    asm jne CHAIN;
    asm push ds;          // qui si salvano solo DS e ES anche se sono usati
    asm push es;          // pure SI e DI perche' a loro pensa il compilatore
    asm cld;              // operazioni su stringhe di bytes incrementano SI DI
    asm cli;              // stop interrupts

// Disinstallazione, fase 1: ripristino dei vettori. Per ogni interrupt gestito
// dal TSR occorre caricare in DS:SI l'indirizzo della f() fittizia che ne
// contiene il vettore originale e in ES:DI l'indirizzo del vettore originale
// (che si trova nella tavola dei vettori). Poi si copiano 4 bytes dalla f()
// fittizia alla tavola dei vettori

    asm mov ax,seg old09h;
    asm mov ds,ax;
    asm mov si,offset old09h;   // DS:SI punta a old09h, cioe' al vettore orig.
    asm xor ax,ax;
```

```
    asm mov es,ax;
    asm mov di,0x09*4;          // ES:DI punta al vettore 09 nella tavola vett.
    asm mov cx,2;
    asm rep movsw;              // copia 2 words da DS:SI a ED:DI
    asm mov ax,seg old10h;
    asm mov ds,ax;
    asm mov si,offset old10h;
    asm xor ax,ax;
    asm mov es,ax;
    asm mov di,0x10*4;
    asm mov cx,2;
    asm rep movsw;
    asm mov ax,seg old1Ch;
    asm mov ds,ax;
    asm mov si,offset old1Ch;
    asm xor ax,ax;
    asm mov es,ax;
    asm mov di,0x1C*4;
    asm mov cx,2;
    asm rep movsw;
    asm mov ax,seg old2Fh;
    asm mov ds,ax;
    asm mov si,offset old2Fh;
    asm xor ax,ax;
    asm mov es,ax;
    asm mov di,0x2F*4;
    asm mov cx,2;
    asm rep movsw;
    asm mov ax,seg old33h;
    asm mov ds,ax;
    asm mov si,offset old33h;
    asm xor ax,ax;
    asm mov es,ax;
    asm mov di,0x33*4;
    asm mov cx,2;
    asm rep movsw;

// Disinstallazione, fase 2: restituzione al DOS della memoria allocata al TSR.
// Si carica in ES l'indirizzo di sgmento del Memory Control Block che il DOS
// ha costruito per il TSR. Il MCB si trova nei 16 bytes che precedeono il PSP
// del TSR, percio' il suo indirizzo e' (PSP-1):0000. Ad offset 1 nel MCB c'e'
// una word che contiene l'indirizzo di segmento del PSP del programma che
// possiede il blocco di ram: se contiene 0 il blocco e' libero. Quindi basta
// caricare 1 in DI e scrivere 0 nella word all'indirizzo ES:DI.

    asm mov ax,word ptr resPSP;
    asm dec ax;
    asm mov es,ax;
    asm mov di,1;               // ES:DI punta al campo owner del MCB del TSR
    asm xor ax,ax;
    asm stosw;                  // azzera il campo: la ram e' restituita al DOS
    asm sti;              // riabilita interrupts
    asm mov ax,HERE_I_AM;
    asm pop es;           // pulisce stack: vedere sopra per SI e DI
    asm pop ds;
  EXIT:
    asm pop di;             // pulisce stack: nella disinstallazione sono
    asm pop si;             // usati SI e DI, pertanto il compilatore il salva
    asm iret;                  // fine operazioni per SSS transiente
}

// nuovo gestore dell'int 33h; non serve dichiararlo interrupt perche' non
// usa nessun registro. E' analogo al gestore dell'int 09h, ma, invece di
// sentire la tastiera, sente il mouse.
```

```
void far new33h(void)
{
    asm mov byte ptr _restoreFlag,1;
    asm cmp byte ptr _saverActive,1;
    asm je EAT_MOUSE;
    asm jmp dword ptr _old33h;    // se SSS e' inattivo lasciamo fare al driver
  EAT_MOUSE:
    asm iret;             // se bisogna annullare il mouse rientra direttamente
}

// animate() effettua le operazioni di animazione del video mentre lo screen
// saver e' attivo. E' opportuno che NON utilizzi f() di libreria C, onde
// poter lasciare residente solo le f() appositamente definite nel sorgente.
// La dichiarazione _saveregs mette new1Ch() al riparo da brutte sorprese.

void _saveregs animate(void)
{
    register i, offset;
    int far *vPtr;

// se il contatore di permanenza del banner a video e' zero, viene riportato
// al valore inziale e si entra nell'algoritmo di gestione del banner

    if(!*(int _cs *)ticksForSpeed) {
        *(int _cs *)ticksForSpeed = MAXSPEED;

// viene utilizzato l'offset a cui e' stato scritto il banner l'ultima volta
// per settare il puntatore al video e cancellarlo. Gli offsets sono nell'array
// appositamente definito con una f() fittizia e sono referenziati con un
// indice anch'esso definito in una f() fittizia.

        offset = ((int _cs *)resBannerOffs)[*(int _cs *)currBannerOff];
        vPtr = (int far *)MK_FP(DEFAULTVIDEO,offset);
        for(i = 0; i < *(int _cs *)resBannerLen; i++)
            vPtr[i] = DEFAULTBLANK;

// se l'offset utilizzato l'ultima volta e' l'ultimo nell'array si riparte dal
// primo, altrimenti si usa il successivo.

        if(*(int _cs *)currBannerOff == MAXPOS)
            *(int _cs *)currBannerOff = 0;
        else
            ++(*(int _cs *)currBannerOff);

// l'uso delle variabili register rende piu' leggibile il codice. Il nuovo
// offset e' memroizzato nella variabile omonima

        offset = ((int _cs *)resBannerOffs)[*(int _cs *)currBannerOff];

// in i e' memorizzato lo stesso offset, aumentato della lunghezza del banner
// moltiplicata per 2. In pratica, i contiene l'offset al quale dovrebbe
// essere scritto l'ultimo carattere del banner

        i = offset+(*(int _cs *)resBannerLen)*2;

// dividendo un offset video per il numero di colonne video (raddoppiato, per
// tenere conto del byte attributo) si ottiene la riga video in cui cade
// l'offset stesso. Se le righe risultanti applicando il calcolo all'offset
// del primo e dell'ultimo carattere del banner, significa che questo verrebbe
// scritto "a cavallo" tra due righe. Per evitare l'inconveniente si
// decrementa l'offset di inizio della lunghezza del banner (per 2), ottenendo
// cosi' un offset approssimato per difetto che consente sicuramente di
// scrivere tutto il banner su una sola riga.
```

```
        if((offset/(DEFAULTCOLS*2)) != (i/(DEFAULTCOLS*2))) {
            offset -= (*(int _cs *)resBannerLen)*2;

// gia' che siamo, modifichiamo anche il valore presente nell'array, in modo
// tale che quando saranno stati utilizzati tutti gli offsets almeno una
// volta, non ci sara' piu' necessario applicare alcuna correzione.

            ((int _cs *)resBannerOffs)[*(int _cs *)currBannerOff] = offset;
        }

// si inizializza il puntatore con l'offset eventualmente corretto e si
// scrive il banner.

        vPtr = (int far *)MK_FP(DEFAULTVIDEO,offset);
        for(i = 0; i < *(int _cs *)resBannerLen; i++)
            ((char far *)vPtr)[i*2] = ((char _cs *)resBanner)[i];
    }
    else

// se il contatore non e' zero lo si decrementa, prima o poi si azzerera'...

        --(*(int _cs *)ticksForSpeed);
}

// blankVideo() copia nel buffer aaposito la videata presente sul display al
// momento dell'attivazione delloscreen saver e, contemporaneamente, scrive a
// video il carattere e l'attributo definiti come DEFAULTBLANK. La
// dichiarazione _saveregs mette new1Ch() al riparo da brutte sorprese.

void _saveregs blankVideo(void)
{
    register counter;
    int far *vPtr;

// inizializzazione del puntatore alla memoria video

    vPtr = (int far *)MK_FP(DEFAULTVIDEO,0);

// effettuazione della copia della memoria video nel buffer apposito e
// contemporanea cancellazione del video

    for(counter = 0; counter < DEFAULTCOLS*DEFAULTROWS; counter++) {
        ((int _cs *)videoBuf)[counter] = vPtr[counter];
        vPtr[counter] = DEFAULTBLANK;
    }

// e adesso facciamo sparire il cursore, salvando le sue coordinate attuali e
// spostandolo fuori dal video. E' ok metterlo sulla riga DEFAULTROWS perche'
// la numerazione BIOS delle righe va da 0 a DEFAULTROWS-1.

    _AH = 3;
    _BH = DEFAULTPAGE;
    geninterrupt(0x10);
    *(char _cs *)savedRow = _DH;
    *(char _cs *)savedCol = _DL;
    _DH = DEFAULTROWS;
    _AH = 2;
    geninterrupt(0x10);
}

// restoreVideo() ripristina la videata presente sul display al momento della
// attivazione dello screen saver. La dichiarazione _saveregs mette new1Ch() al
// riparo da brutte sorprese.
```

```
void _saveregs restoreVideo(void)
{
    register counter;
    int far *vPtr;

    vPtr = (int far *)MK_FP(DEFAULTVIDEO,0);
    for(counter = 0; counter < DEFAULTCOLS*DEFAULTROWS; counter++)
        vPtr[counter] = ((int _cs *)videoBuf)[counter];

// rimettiamo a posto il cursore, utilizzando le coordinate salvate in
// precedenza

    _DH = *(char _cs *)savedRow;
    _DL = *(char _cs *)savedCol;
    _BH = DEFAULTPAGE;
    _AH = 2;
    geninterrupt(0x10);
}

// fine della parte residente del TSR. Tutto quello che serve al TSR per
// lavorare, a patto che animate() e sue eventuali subroutines non utilizzino
// funzioni di libreria, sta al di sopra di queste righe di commento. Cio'
// consente di limitare al massimo la quantita' di memoria allocata in modo
// permanente al TSR.

/*--------------------------------------------------------------------------*/

// inizia qui la parte transiente del TSR. Tutte le f() definite a partire
// da questo punto vengono buttate via quando il programma si installa in
// memoria. Queste f() possono fare tutto quello che vogliono, usare f() di
// libreria, variabili globali e statiche, etc.

// releaseEnv() butta alle ortiche l'environment del TSR: dal momento che
// questo non lo utilizza per le proprie attivita' e' inutile lasciarlo li' a
// occupare memoria. Inoltre, dal momento che releaseEnv() e' la prima delle
// funzioni definita fuori dalla parte residente, il suo indirizzo puo' essere
// utilizzato per calcolare quanta ram lasciare residente (secondo parametro)
// della keep()). Restituisce 0 se tutto ok.

int releaseEnv(void)
{
    extern unsigned _envseg;        // variabile non documentata in Borland C++

    return(freemem(_envseg));
}

// areYouThere() e' l'interfaccia di comunicazione con l'int 2Fh, che viene
// invocato passando HEY_YOU in AH e il numero del servizio richiesto in AL.
// L'int 2Fh risponde sempre HERE_I_AM in AX per segnalare che il servizio e'
// stato espletato.

unsigned areYouThere(char service)
{
    union REGS regs;

    regs.h.ah = HEY_YOU;
    regs.h.al = service;
    return(int86(0x2F,&regs,&regs));
}

// per evitare di utilizzare funzioni di libreria nelle routines residenti, si
// inizializza un array di offsets video ai quali scrivere in sequenza il
// banner. Gli offsets sono generati in modo pseudocasuale. La divisione e
```

```
// successiva moltiplicazione applicate al valore restituito da random()
// assicurano che l'offset cosi' ottenuto sia sempre pari.

void initializeOffsets(void)
{
    register i;

    randomize();
    for(i = 0; i < MAXPOS; i++)
        ((int _cs *)resBannerOffs)[i] = (random(DEFAULTROWS*DEFAULTCOLS*2)/2)*2;
}

// install() effettua l'installazione in memoria del TSR, attivando i nuovi
// vettori di interrupt e chiedendo al DOS di riservare al programma la
// memoria necessaria. Questa e' calcolata come differenza tra l'indirizzo
// di releaseEnv() (prima f() transiente) trasformato in indirizzo di segmento
// (cioe' seg+(off/16) arrotondato per eccesso (+1) per sicurezza) e
// l'indirizzo al quale e' caricato (quello del suo PSP). L'indirizzo di
// segmento del PSP (_psp, definita in DOS.H) e' salvato nello spazio riservato
// dalla f() fittizia opportunamente definita: serve al gestore dell'int 2Fh
// per la disinstallazione del programma.

void install(char *banner)
{
    register len;

    if((len = strlen(banner)) > MAXBANNER)
        printf("%s: Could not install. Banner too long (max. %d chr.)\n",PRG,
                                                                    MAXBANNER);
    else
        if(releaseEnv())
            printf("%s: Could not install. Error releasing environment.\n",PRG);
        else {
            *(int _cs *)resPSP = _psp;
            *(int _cs *)resBannerLen = len;
            _fstrcpy((char _cs *)resBanner,(char far *)banner);
            initializeOffsets();
            asm cli;
            *(long _cs *)old09h = (long)getvect(0x09);
            *(long _cs *)old10h = (long)getvect(0x10);
            *(long _cs *)old1Ch = (long)getvect(0x1C);
            *(long _cs *)old2Fh = (long)getvect(0x2F);
            *(long _cs *)old33h = (long)getvect(0x33);
            setvect(0x09,(void(interrupt *)())new09h);
            setvect(0x10,(void(interrupt *)())new10h);
            setvect(0x1C,(void(interrupt *)())new1Ch);
            setvect(0x2F,(void(interrupt *)())new2Fh);
            setvect(0x33,(void(interrupt *)())new33h);
            asm sti;
            printf("%s: Installed. Invoke with %s to uninstall.\n",PRG,UNINST_OPT);
            keep(0,FP_SEG(releaseEnv)+FP_OFF(releaseEnv)/16+1-_psp);
        }
}

// uninstall() richiede all'int 2Fh la disinstallazione del programma. Se
// areYouThere() non risponde HERE_I_AM e' accaduto qualcosa di strano: in
// teoria cio' dovrebbe avvenire solo se il TSR non e' residente, ma dato che
// prima main() effettua questo controllo prima di invocare uninstall(), la
// probabile causa dell'errore e' che tra le due fasi (controllo e richiesta)
// qualche altro TSR abbia incasinato la memoria disattivando il nostro
// gestore di int 2Fh.

void uninstall(void)
{
```

```
    if(areYouThere(UNINSTALL) == HERE_I_AM)
        printf("%s: Uninstalled. Vectors restored and RAM freed up.\n",PRG);
    else
        printf("%s: Unidentified error.\n",PRG);
}

// main() controlla se il TSR e' residente: in tal caso l'unica azione
// possibile e' la disinstallazione. In caso contrario si puo' tentare
// l'installazione

void main(int argc,char **argv)
{
    printf("%s %s - Sample Screen Saver - Barninga Z! '%s.\n",PRG,REL,YEAR);
    if(argc != 2)
        printf("%s: Syntax error. Usage: %s bannerString | *\n",PRG,PRG);
    else
        if(areYouThere(HANDSHAKE) == HERE_I_AM)
            if(!strcmp(argv[1],UNINST_OPT))
                uninstall();
            else
                printf("%s: Could not install. Already active in RAM.\n",PRG);
        else
            install(argv[1]);
}
```

Il listato di `SSS.C` contiene commenti in quantità: non pare necessario, pertanto, dilungarsi sulle sue caratteristiche implementative; tuttavia si possono individuare alcuni spunti per migliorarne l'impianto complessivo. In primo luogo, `SSS` manca di una routine di riconoscimento della modalità video attiva, con la conseguenza che esso lavora correttamente solo se il modo video è *testo 80 % 25*: un esempio di gestione "intelligente" del buffer video si trova a pag. 529. Un secondo limite è costituito dall'impossibilità, per l'utilizzatore, di specificare un tempo di attesa diverso da quello di default (stabilito dalla costante manifesta `MAXTICKS`); infine, potrebbe essere interessante dotare `new09h()` di una routine per il riconoscimento di uno hotkey di richiesta di attivazione immediata dello screen saver (un gestore di int 09h assai più sofisticato di quanto occorra in questo caso si trova a pag. 520). Al lettore volonteroso non rimane che... darsi da fare.

I DEVICE DRIVER

Sempre più difficile: dopo avere affrontato i TSR (pag. 275) è ora il turno dei device driver. Di che si tratta? Un device driver è, come evidenzia il nome stesso, un pilota di una qualche diavoleria: insomma, un programma dedicato alla gestione di una periferica hardware.

Dal punto di vista logico i device driver sono estensioni del DOS, che li carica durante la fase di bootstrap: si tratta di un meccanismo che consente, normalmente, di personalizzare la configurazione software del personal computer incorporandovi a basso livello le routine necessarie per pilotare in modo opportuno le periferiche aggiuntive hardware (scanner, scheda fax, etc.) per le quali il DOS non sia attrezzato, ma è ovviamente possibile utilizzare la tecnica dei device driver anche per attivare gestori più sofisticati di quelli già disponibili nel sistema operativo. Proprio per questo il loro nome completo è *installable device driver*, in contrapposizione ai *resident device driver*, routine già presenti nel software di sistema[322] (video, tastiera, dischi, etc.).

ASPETTI TECNICI

Un device driver è, a tutti gli effetti, un programma TSR, ma le accennate modalità di caricamento ed utilizzo impongono (in base alle specifiche Microsoft) che esso abbia una struttura del tutto particolare, purtroppo non coincidente con quella generata dai compilatori C (vedere pag. 289): per imparare a scrivere in C un device driver, pertanto, occorre innanzitutto capire come esso è strutturato e come viene caricato ed utilizzato dal DOS[323].

Il bootstrap

Il bootstrap è la fase iniziale della sessione di lavoro della macchina. Dapprima sono eseguite le routine di autodiagnostica del BIOS, il quale provvede in seguito a cercare sui dischi del computer il loader del DOS: si tratta di una routine, memorizzata in uno dei primi settori del disco fisso (il boot sector), che carica in memoria ed esegue il primo dei due file nascosti del DOS (solitamente chiamato `IO.SYS`).

`IO.SYS` si compone di una parte residente e di una parte transiente: la prima è destinata a rimanere in memoria fino al successivo spegnimento o *reset* della macchina, mentre la seconda serve esclusivamente ad effettuare le operazioni di caricamento del sistema. La porzione transiente, infatti, tramite una routine chiamata `SYSINIT`, individua l'indirizzo del cosiddetto *top of memory* (il limite superiore della memoria convenzionale) e copia `IO.SYS` "lassù", al fine di sgombrare la parte inferiore della RAM. A questo punto in memoria vi sono due istanze di `IO.SYS`: la `SYSINIT` della seconda carica l'altro file nascosto (`MSDOS.SYS`) in modo da ricoprire la porzione transiente della prima. `SYSINIT` attiva poi tutti i device driver residenti (quelli incorporati nel DOS), legge il file

---

[322] Il concetto di device driver installabili è stato introdotto con la versione 2.0 del DOS. La versione 1.0 incorporava tutti i device driver previsti e non era possibile caricarne di nuovi.

[323] I device driver non sono mai caricati lanciandoli al prompt del dos, come avviene per i normali programmi: al contrario essi vengono installati in memoria dal sistema operativo stesso durante il bootstrap, se specificato nel file `CONFIG.SYS` mediante l'istruzione `DEVICE=`, ad esempio:

```
DEVICE=C:\UT\MOUSE.SYS
```

CONFIG.SYS per determinare quali sono gli installable device driver da caricare e provvede al loro caricamento ed inizializzazione: il device driver consente, da questo momento in poi, di accedere alla periferica mediante un nome, che per il sistema operativo equivale (dal punto di vista logico) ad un nome di file o di unità disco, come sarà meglio chiarito tra breve. L'ultima operazione effettuata da SYSINIT è l'esecuzione dell'interprete dei comandi (COMMAND.COM o il programma specificato dall'istruzione SHELL= nel file CONFIG.SYS[324]).

Tipi di device driver

Esistono due tipi di device driver: *character* e *block* device driver (driver per periferiche a carattere o a blocchi).

I primi sono adatti alla gestione di periferiche come terminali e stampanti, cioè periferiche che effettuano le loro operazioni di I/O un carattere (byte) alla volta. Il nome assegnato dal driver alla periferica può essere usato dal DOS e dalle applicazioni come un nome di file, sul quale scrivere o dal quale leggere i byte. Se il nome è identico a quello già utilizzato da un device driver residente, quest'ultimo è sostituito, nelle sue funzionalità, dall'installable device driver[325]. Il DOS può comunicare con i character device driver in due modalità differenti, a seconda che essi siano definiti come *raw* o *cooked*: la modalità raw (grezza) prevede che ogni singolo byte passi dal driver al DOS o viceversa senza alcuna modifica; in modalità cooked (letteralmente... "cucinata", ma con un po' di fantasia si potrebbe tradurre in "interpretata") il DOS memorizza i caratteri in un buffer e gestisce opportunamente i caratteri di controllo (CTRL-C, etc.) prima di passarli (al primo RETURN incontrato) all'applicazione o al device driver[326]. La modalità raw o cooked è selezionabile dalle applicazioni, mediante la subfunzione 01h del servizio 44h dell'int 21h:

[324] Una tipica istruzione SHELL= in CONFIG.SYS è:

```
SHELL=C:\DOS\COMMAND.COM C:\DOS /E:512 /P
```

[325] Alcuni device driver residenti hanno nomi piuttosto noti (vedere pag. 116): CON (tastiera/video), AUX (prima porta seriale), PRN (prima porta parallela), NUL (device nullo: una specie di buco nero). Nulla vieta di scrivere nuovi installable device driver per la loro gestione: è sufficiente, ad esempio, che un device driver si registri al DOS come AUX perchè il sistema lo utilizzi come nuovo programma di pilotaggio della porta seriale.

[326] La subfunzione 00h del servizio 44h dell'int 21h consente all'applicazione di conoscere gli attributi del driver. Essa è del tutto analoga alla 01h, ma non prevede alcun input in DX, mentre AL deve essere, evidentemente, 00h. In uscita, il registro DX contiene gli attributi del driver, come descritto a pag. 359, con la sola eccezione del bit 6, che vale 1 se si è verificata una condizione di EOF nell'ultima operazione di input dal device.

INT 21H, SERV. 44H, SUBF. 01H: MODIFICA GLI ATTRIBUTI DI UN DEVICE DRIVER

Input	AH	44h
	AL	01h
	BX	handle[327]
	DH	00h
	DL	Nuovi bit di stato: `Bit 7: 1` `5: 1` = RAW, `0` = COOKED Per i bit 0-4 e 6 si rimanda alla descrizione della device attribute word, a pagina 359.
Output	AX	Codice di errore se il `CarryFlag` è `1`. Se il `CarryFlag` è `0` la chiamata ha avuto successo

I block device driver gestiscono periferiche che effettuano l'I/O mediante blocchi di byte (esempio classico: i dischi). Contrariamente alle periferiche a carattere, accessibili esclusivamente in modo sequenziale, i block device sono dispositivi ai quali è possibile accedere in modo *random* (casuale), cioè con spostamenti arbitrari avanti o indietro rispetto alla posizione attuale nel flusso di dati. Il DOS assegna ad ogni periferica gestita dal device driver[328] un nome di disco (una lettera dell'alfabeto seguita dai due punti): un installable block device driver non può, dunque, sostituirsi a un resident device driver, ma solamente affiancarsi ad esso nella gestione di altre periferiche dello stesso tipo. Va sottolineato, infine, che i block device driver operano sempre in modalità raw.

Struttura e comportamento dei device driver

In cosa consistono le particolarità della struttura dei device driver? Va detto, innanzitutto, che non si tratta di file eseguibili[329] in senso stretto, ma di file dei quali viene caricata in memoria l'immagine binaria: in altre parole, essi vengono caricati in RAM esattamente come sono memorizzati sul disco,

---

[327] Vedremo tra breve che un character device, grazie alla "intermediazione" svolta dal driver, è accessibile via handle, in modo analogo a quanto avviene per i file (vedere pag. 126).

[328] Un block device driver può gestire più periferiche contemporaneamente, al contrario dei character device driver, che possono pilotarne una soltanto.

[329] Non sono, cioè, files `.EXE` o `.COM`. Per la precisione, il DOS è in grado, a partire dalla versione 3.0, di caricare device driver che si presentino come files `.EXE`: questi sono tali a tutti gli effetti, ma la loro struttura interna non differisce da quella della generalità dei device driver e permangono, di conseguenza, tutte le difficoltà del caso qualora si intenda scrivere in C un device driver indipendentemente dal fatto che si voglia o no ottenere, quale risultato, un file `.EXE`. Rimane da sottolineare, al riguardo, che la maggior parte dei device driver è costituita da file con estensione `.SYS`: non è un requisito di sistema, ma semplicemente una convenzione largamente condivisa e seguita.

senza la creazione di PSP (vedere pag. 324) ed environment[330], né la gestione di una eventuale Relocation Table (pag. 278) da parte del sistema operativo.

Inoltre, i primi 18 byte di ogni device driver sono riservati ad una tabella, detta *header*, contenente informazioni ad uso del DOS (sulla quale ci sofferemeremo tra poco): proprio qui incontriamo uno dei maggiori ostacoli, poiché, come è facile intuire, si tratta di una caratteristica assolutamente incompatibile con la normale modalità di compilazione dei programmi C[331].

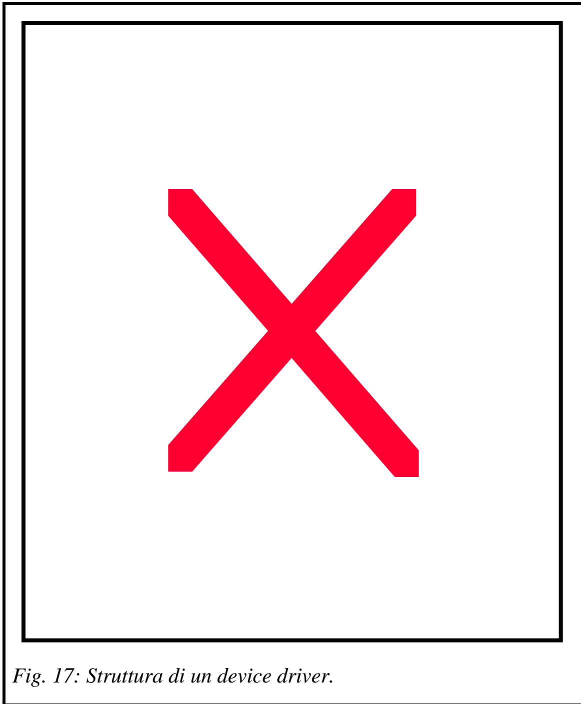

Fig. 17: Struttura di un device driver.

Ogni device driver, inoltre, incorpora una routine che deve provvedere a salvare, per utilizzi successivi, l'indirizzo del buffer attraverso il quale il DOS comunica con il driver stesso: è la cosiddetta *strategy routine* (a dire il vero non si vede che cosa ci sia di strategico in tutto ciò, ma comunque...).

Il terzo elemento caratteristico dei device driver è, infine, la *interrupt routine*: anche in questo caso il nome è poco azzeccato, perchè si tratta di una procedura che ha ben poco a che fare con i "classici" gestori di interrupt[332] (vedere, ad esempio, pag. 251); il suo compito consiste nell'individuare il servizio richiesto dal DOS al driver ed attivare la routine (interna al driver) corrispondente.

Ne consegue in modo ovvio che, per ogni servizio gestito, il driver deve incorporare una routine dedicata, più una routine generalizzata di gestione dell'errore per i casi in cui il DOS richieda un servizio non previsto: ogni driver deve però necessariamente gestire è il servizio `00h`, corrispondente alla propria inizializzazione in fase di bootstrap[333].

La figura 17 schematizza la struttura di un device driver, coerentemente con le considerazioni sin qui esposte: si vede facilmente che le analogie con i programmi TSR sono molteplici (vedere, ad esempio, la figura 14 a pag. 276).

Ne sappiamo abbastanza, a questo punto, per capire (a grandi linee) come lavorano i device driver: il "protocollo" di colloquio tra sistema operativo e driver è fisso e si articola in quattro fasi ben definite.

---

[330] La mancanza di un environment (variabili di ambiente) appare del resto ovvia, quando si pensi che, al momento del caricamento dei device driver, l'interprete dei comandi (che è incaricato della generazione dell'environment per tutti i processi) non è ancora stato lanciato.

[331] Vale la pena di ricordare che in testa ad ogni eseguibile generato dalla compilazione di un sorgente C vi è, normalmente, il codice del modulo di startup. Inoltre, se il programma è un `.EXE`, in testa al file deve esserci la Relocation Table.

[332] La *interrupt routine* si differenzia dagli interrupt, tra l'altro, in quanto è attivata dal DOS (pertanto non la chiama l'applicazione in foreground né un evento hardware asincrono) e non termina con una istruzione `IRET` (bensì con una normalissima `RETF`). Tutto ciò non significa, comunque, che un device driver non possa incorporare gestori di interrupt qualora il programmatore lo reputi necessario.

[333] Dei servizi che il DOS può richiedere ad un device driver si dirà tra poco.

- il DOS invoca la strategy routine passandole in `ES:BX` l'indirizzo (puntatore `far`) di un buffer (detto *request header*[334]) contenente i dati che occorrono per effettuare l'operazione (servizio) prevista (la struttura del buffer varia a seconda del servizio);

- la strategy routine salva l'indirizzo del buffer in una variabile (locazione di memoria) nota ed accessibile alle altre routine del driver e restituisce il controllo al sistema;

- il DOS invoca la interrupt routine[335];

- la interrupt routine accede al buffer (mediante l'indirizzo salvato in precedenza dalla strategy routine) per conoscere il numero del servizio richiesto (e relativi dati) e provvede all'esecuzione delle operazioni per esso previste, generalmente chiamando una routine dedicata, la quale, a sua volta, può chiamare altre routine del driver o interrupt di sistema: al termine, la interrupt routine o le routine dedicate scrivono nel solito buffer i risultati dell'elaborazione ed un valore avente significato di codice di errore, ed infine restituiscono il controllo al DOS.

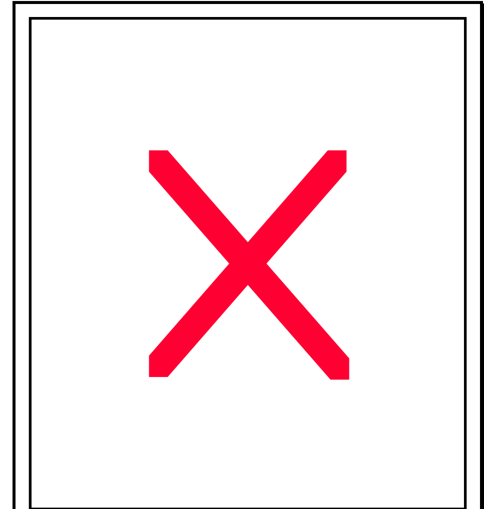

Fig. 18: Comunicazione tra applicazione e periferica via device driver.
1) richiesta I/O via int 21h
2) chiamata a strategy routine
3) ritorno al DOS
4) chiamata a interrupt routine
5) pilotaggio periferica
6) risposta della periferica
7) risposta al DOS
8) risposta all'applicazione

Conseguenza immediata di tale algoritmo è il completo isolamento del device driver dalle applicazioni che ne utilizzano i servizi: i driver, è evidente, interagiscono esclusivamente con il sistema operativo, che si pone quale interfaccia tra essi e le applicazioni. Ciò risulta ancora più palese quando si consideri che i driver rendono accessibili le periferiche loro associate attraverso un nome di file o di unità disco: le applicazioni devono ricorrere agli appositi servizi DOS[336]. Infatti, le applicazioni effettuano le operazioni di I/O richiedendo l'opportuno servizio al sistema operativo (int 21h): il DOS individua (grazie ad una tabella costruita durante il bootstrap) il device driver interessato, costruisce un request header appropriato e chiama la strategy routine. Il device driver memorizza l'indirizzo del buffer e restituisce il controllo al DOS che, immediatamente, chiama la interrupt routine, secondo lo schema analizzato: ne risulta un flusso di processi come quello rappresentato in figura 18.

Vale la pena di osservare che la suddivisione delle operazioni di interfacciamento DOS/driver tra due routine (la strategy e la interrupt) è ispirata alle esigenze di sistemi multitasking (come Unix): essa non è di alcuna utilità in sistemi monotasking (quale è il DOS), in quanto questi eseguono sempre e solo un'unica operazione di I/O alla volta.

Questo è il momento di approfondire l'analisi della struttura dello header, del buffer di comunicazione con il DOS e dei servizi che il device driver può implementare: forse non è divertente, ma è di fondamentale importanza...

---

[334] Tanto per non fare confusione...

[335] Come fa il DOS a conoscere gli indirizzi della strategy e della interrupt routine, in modo da poterle chiamare? Semplice: essi sono contenuti nella tabella di 18 byte in testa al device driver.

[336] Una parziale eccezione si ha quando il device driver incorpora routine di gestione di interrupt che le applicazioni utilizzano direttamente. Si tratta di un comportamento lecito ma, in qualche misura, anomalo e maggiormente analogo a quello dei programmi TSR che a quello caratteristico dei device driver.

Il Device Driver Header: in profondità

Il device driver header è la tabella che occupa i primi 18 byte del codice di ogni device driver. Vediamone contenuto e struttura:

STRUTTURA DEL DEVICE DRIVER HEADER

OFFSET	DIM.	CONTENUTO
`00h`	4	Puntatore `far` al device driver successivo. Deve contenere il valore `FFFFFFFFh`, in quanto viene inizializzato automaticamente dal DOS al caricamento del driver[337].
`04h`	2	*Device Attribute Word*. E' una coppia di byte i cui bit sono utilizzati per descrivere alcune caratteristiche della periferica (vedere di seguito).
`06h`	2	Puntatore `near` alla strategy routine.
`08h`	2	Puntatore `near` alla interrupt routine.
`0Ah`	8	Se la periferica che il driver gestisce è di tipo character, il campo contiene il nome logico assegnato alla periferica stessa, utilizzabile dalle applicazioni come un nome di file: se il nome è lungo meno di 8 byte, lo spazio restante deve essere riempito con spazi. Se la periferica gestita è di tipo block, il primo byte del campo contiene il numero di unità supportate; l'uso dei restanti 7 byte è riservato al DOS. Il programmatore non ha necessità di inizializzare il primo byte del campo: a ciò provvede il DOS con le informazioni restituite dal driver al termine della fase di *Init* (vedere pag. 363).

Si noti che i puntatori alla strategy e interrupt routine sono `near`: il DOS effettua però chiamate `far`, utilizzando quale parte segmento dell'indirizzo il segmento al quale il driver stesso è caricato. Ne segue che i due campi menzionati contengono, in realtà, la parte offset dell'indirizzo delle due funzioni e che queste devono essere dichiarate entrambe `far` (e terminare quindi con una istruzione `RETF`).

La tabella che segue descrive il significato dei bit della Device Attribute Word.

[337] Va detto che un unico file può contenere più di un device driver. In questo caso, in testa al codice di ogni device driver deve trovarsi un device driver header: il campo ad offset `00h` dovrà essere correttamente inizializzato a cura del programmatore con l'indirizzo del successivo driver (cioè con l'indirizzo del primo byte del successivo header) in tutti gli header eccetto l'ultimo, che contiene ancora il valore `FFFFFFFFh` (`-1L`).

STRUTTURA DELLA DEVICE ATTRIBUTE WORD

BIT	SIGNIFICATO
15	1 se si tratta di un character device driver 0 se è un block device driver
14	1 se il device driver supporta i servizi IOCTL[338] di lettura e scrittura
13	1 se la periferica è un block device (disco) formattato in modo non-IBM
12	0 (riservato)
11	1 se il drivere può gestire un block device (disco) rimuovibile (DOS 3 e succ.)
7-10	0 (riservati)
6	1 se il device driver supporta i servizi IOCTL *Generic* e *Get/Set Logical Drive*
5	0 (riservato)
4	1 se il device driver supporta la funzione DOS speciale di output veloce per la periferica CON
3	1 se il device driver è il driver per il device CLOCK
2	1 se il device driver è il driver per il device NUL
1	1 se il device driver è il driver per lo standard output (vedere pag. 116)
0	1 se il device driver è il driver per lo standard input (vedere pag. 116)

Va sottolineata l'importanza del bit 15, che indica se la periferica gestita lavora a blocchi o a caratteri (vedere pag. 354); qualora esso valga `1` (block device driver), solo i bit 6, 11 e 13-15 sono significativi: tutti gli altri devono essere impostati a `0`.

E' ancora interessante notare che le informazioni contenute nel device driver header sono utilizzate dal sistema operativo, mentre, di norma, le applicazioni non vi accedono. Vi sono però alcuni servizi IOCTL che consentono di leggere e modificare alcuni dei bit (non tutti) della attribute word[339].

---

[338] IOCTL significa Input/Output Control. Si tratta di una modalità di gestione delle periferiche mediante stringhe di controllo, che a livello DOS è supportata dal servizio 44h dell'int 21h.

[339] Sono le funzioni 00h e 01h del servizio 44h dell'int 21h (`GetDeviceInfo` e `SetDeviceInfo`). La funzione 00h consente, tra l'altro, di conoscere se ad un nome logico di file è associato un device o un vero e proprio file (per la descrizione del servizio vedere pag. 354; per un esempio si veda pag. 203).

Il Request Header e i servizi: tutti i particolari

Il request header è il buffer attraverso il quale DOS e device driver si scambiano le informazioni: il sistema operativo ne carica l'indirizzo in `ES:BX` e chiama la strategy routine perché il device driver possa conoscerlo e memorizzarlo. Il buffer contiene il numero del servizio richiesto, nonché tutti i dati necessari al driver per il suo espletamento, e si divide in due parti: la prima, detta *parte fissa*, è identica (quanto a numero, ordine e dimensione dei campi) per tutti i servizi, mentre la seconda, detta *parte variabile*, ha struttura differente a seconda del servizio richiesto (alcuni servizi presentano comunque identica parte variabile del request header, o non la utilizzano del tutto). La parte fissa del request header è strutturata come segue.

STRUTTURA DEL DEVICE DRIVER REQUEST HEADER

OFFSET	DIM.	CONTENUTO
`00h`	1	Lunghezza totale del request header. Il valore, diminuito di 13 (lunghezza della parte fissa) esprime la lunghezza della parte variabile. E' un campo utilizzato assai raramente, dal momento che la lunghezza del request header può essere desunta dal numero del servizio richiesto.
`01h`	1	Numero dell'unità (disco). E' un campo significativo solo per i block device driver e indica su quale disco (o altro block device) deve essere eseguito il servizio richiesto.
`02h`	1	*Command code*. E' il numero del servizio richiesto dal DOS al device driver.
`03h`	2	*Return Code* (*Status word*). E' il valore restituito dal driver al DOS per indicare lo stato del servizio eseguito.
`05h`	8	Utilizzo riservato al DOS.

E' importante approfondire l'analisi del Return Code: si tratta di una word (due byte) nella quale il byte più significativo è interpretato come campo di bit, ed è usato per indicare lo stato del servizio; il byte meno significativo contiene invece un valore che descrive un errore se il bit 15 della word vale 1: se questo è 0, detto valore viene ignorato dal sistema operativo. Il dettaglio dei bit e codici di errore è riportato nella tabella che segue.

STRUTTURA DELLA DEVICE DRIVER STATUS WORD

BIT	SIGNIFICATO
`15`	*Error Flag*. 1 se si è verificato un errore, 0 altrimenti.
`12-14`	Riservati.
`9`	*Busy Flag*. 1 se il driver vuole impedire al DOS di richiedere ulteriori servizi (ad esempio perché l'esecuzione del servizio non è stata ancora portata a termine), 0 altrimenti.
`8`	*Done Flag*. 1 se il servizio è stato completamente eseguito, 0 se l'operazione non è ancora stata completata.
`0-7`	*Error Code* (codice di errore). E' significativo solo se il bit 15 (*Error Flag*) è 1: `00h` Tentativo di scrittura su unità protetta. `01h` Unità sconosciuta. `02h` Unità non pronta (ad es.: sportello del disk drive aperto). `03h` Servizio non supportato. `04h` Errore di CRC. `05h` Lunghezza del Request Header errata[340]. `06h` Errore di ricerca dati sull'unità (*seek error*). `07h` Tipo di unità sconosciuto. `08h` Settore non trovato. `09h` Stampante senza carta. `0Ah` Errore di scrittura. `0Bh` Errore di lettura. `0Ch` Errore generico, non individuato (*General failure*). `0Dh` Riservato. `0Eh` Riservato. `0Fh` Cambiamento di disco non valido (a partire dal DOS 3.0)[341].

Come si è detto, la parte variabile del request header è strutturata in dipendenza dal servizio richiesto dal DOS al driver, cioè a seconda del valore che il campo *Command code* assume. Si noti che il driver restituisce valori e informazioni al DOS scrivendoli, a sua volta, nel request header (in campi della parte fissa o variabile). Di seguito è presentato l'elenco completo dei servizi che il DOS può richiedere al driver.

---

[340] Usato solo dai programmatori che hanno tempo da perdere per sviluppare una routine di controllo. Quasi tutti si fidano, a torto o a ragione, del DOS.

[341] Significa: "Non fare il furbo, rimetti il disco che c'era prima!".

ELENCO DEI SERVIZI IMPLEMENTABILI DAI DEVICE DRIVER

CODICE	SERVIZIO
00	*Init* (inizializzazione del driver).
01	*Media Check* (solo per block device driver)
02	*Build BIOS Parameter Block* (solo per block device driver)
03	*IOCTL Read*
04	*Read* (input)
05	*Nondestructive Read* (solo per character device driver)
06	*Input Status* (solo per character device driver)
07	*Flush Input Buffers* (solo per character device driver)
08	*Write* (output)
09	*Write With Verify*
10	*Output Status* (solo per character device driver)
11	*Flush Output Buffers* (solo per character device driver)
12	*IOCTL Write*
13	*Device Open*
14	*Device Close*
15	*Removable Media* (solo per block device driver)
16	*Output Until Busy* (solo per character device driver)
19	*Generic IOCTL Request*
23	*Get Logical Device* (solo per block device driver)
24	*Set Logical Device* (solo per block device driver)

I servizi 13-16 sono stati introdotti a partire dalla versione 3.0 del DOS, mentre i servizi 19 e 23-24 dalla versione 3.2. Di seguito sono analizzati nel dettaglio tutti i servizi elencati e, per ciascuno di essi, la corrispondente struttura della parte variabile del device driver request header.

Servizio 00: Init

Il servizio 0 è la routine di inizializzazione del driver, detta *Init*. Esso è richiesto dal DOS una volta sola, nella fase di caricamento del driver durante il bootstrap[342]. Il driver ha così la possibilità di effettuare tutte le operazioni necessarie alla predisposizione dell'operatività successiva: controllo dello hardware, installazione di gestori di interrupt e via dicendo. L'utilizzo del request header è il seguente:

DEVICE DRIVER, SERV. 00: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1		
02h	1	Numero del servizio richiesto (0)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		
0Dh	1		Numero di unità supportate (solo block device driver).
0Eh	4		Puntatore far al primo byte di memoria libera oltre il device driver.
12h	4	Puntatore far alla riga di comando del driver nel file CONFIG.SYS. Punta al primo byte che segue la stringa "DEVICE=".	Usato solo dai block device driver. Puntatore far all'array di puntatori ai BPB (vedere pag. 364). E' un array di word, ciascuna delle quali è un puntatore near ad un BPB. L'array contiene un elemento (word) per ogni unità logica supportata dal device driver.
16h	1	Numero della prima unità disco disponibile[343] (solo block device driver).	

[342] Ne segue che, se il codice della Init si trova in coda al sorgente e il driver non utilizza funzioni di libreria una volta terminata la fase di caricamento (cioè nelle routine che implementano gli altri servizi), l'occupazione di memoria può essere ridotta escludendo la Init stessa dalla porzione residente (vedere, per analogia, pag. 276).

Bisogna tenere presente che quando i device driver vengono caricati dal DOS, quest'ultimo non ha ancora completato la propria installazione[344] e, pertanto, non tutte le funzionalità che esso implementa sono disponibili. In particolare, Microsoft afferma che il servizio 0 dei device driver può utilizzare solo alcune delle funzioni dell'int 21h: da 01h a 0Ch (I/O di caratteri), 25h e 35h (installazione e richiesta di vettori di interrupt), 30h (richiesta della versione DOS). In realtà, esperimenti empirici hanno rivelato che altre funzioni sono attive e disponibili: particolarmente importanti risultano quelle relative all'I/O con i file (durante la fase di init è quindi possibile, ad esempio, leggere un file di configurazione specificato sulla riga di comando del device driver in `CONFIG.SYS`).

Il sistema operativo, nel richiedere il servizio 0, consente al driver di conoscere la riga di comando nel file `CONFIG.SYS`, come specificato circa il campo ad offset `12h` nel request header. La stringa, tutta in caratteri maiuscoli, termina al primo `LF` o `CR` o `EOF` (`10h` o `13h` o `1Ah`) e non deve essere modificata dal driver, che può però copiarla in una locazione di memoria privata per effettuare tutte le elaborazioni eventualmente necessarie.

Al termine della Init, il driver deve porre a `1` il bit 8 (Done Flag) della status word (campo ad offset `03h` del request header), e deve indicare al DOS quanta memoria deve essere riservata per la parte residente: particolarmente importante allo scopo risulta il campo ad offset `0Eh`, in quanto consente al driver di specificare l'indirizzo del primo byte di RAM oltre la porzione residente. Il DOS sa, in tal modo, che a quell'indirizzo inizia la memoria disponibile per le successive operazioni di bootstrap. Detto indirizzo può validamente essere, ad esempio, quello della prima funzione della parte transiente nel sorgente C (se la parte residente non usa funzioni di libreria). Se il driver rileva, durante l'inizializzazione, errori tali da renderne inutile il caricamento (ad esempio: il driver del mouse non ne trova alcuno collegato al personal computer), può scrivere nel campo in questione l'indirizzo del proprio device driver header: essendo questo l'indirizzo al quale il driver stesso è caricato in memoria, il sistema operativo non riserva neppure un byte alla parte residente, risparmiando così preziosa memoria. Per segnalare al DOS la condizione di "aborted installation" occorre anche azzerare il bit 15 della device attribute word nel device driver header (pag. 359) e restituire `0` nel campo ad offset `0Dh` del request header.

Il driver (se è un block device driver) può anche conoscere il numero assegnato alla prima delle sue unità grazie all'ultimo campo della parte variabile del request header; esso deve comunicare al DOS il numero di unità supportate (campo ad offset `0Dh`), tramite il quale il DOS assegna loro gli identificativi letterali[345], nonché gli indirizzi (campo ad offset `12h`) dei BPB che descrivono ogni unità. Il BPB (BIOS Parameter Block) è una tabella che contiene i parametri BIOS per un'unità disco e ha il formato descritto di seguito:

---

[343] A partire dal DOS 3.0.

[344] Mancano all'appello, ad esempio, l'interprete dei comandi, il master environment, la catena dei memory control block (vedere pag. 191), etc..

[345] Facciamo un esempio. Se il DOS "dice" al driver che la prima unità libera è la numero 3, ciò significa che al momento del caricamento del driver sono presenti nel sistema 3 unità disco, numerate da 0 a 2 e chiamate `A:`, `B:` e `C:`. Se, al termine della Init, il driver "risponde" di supportare 2 unità, queste verranno numerate 3 e 4 e saranno loro attribuiti gli identificativi `D:` ed `E:`.

STRUTTURA DEL BPB

OFF	DIM	CAMPO
`00h`	2	Lunghezza del settore in byte
`02h`	1	Numero di settori per cluster (unità di allocazione)
`03h`	2	Numero di settori riservati (a partire dal settore 0)
`05h`	1	Numero di FAT (File Allocation Table) presenti sul disco
`06h`	2	Numero massimo di elementi nella directory root
`08h`	2	Numero totale di settori
`0Ah`	1	Media descriptor byte (o Media ID byte): `F0h`: floppy 3.5" (2 lati, 18 sett./traccia) `F8h`: hard disk `F9h`: floppy 5.25" (2 lati, 15 sett./traccia) `F9h`: floppy 3.5" (2 lati, 9 sett./traccia) `FCh`: floppy 5.25" (1 lato, 9 sett./traccia) `FDh`: floppy 5.25" (2 lati, 9 sett./traccia) `FEh`: floppy 5.25" (1 lati, 8 sett./traccia) `FFh`: floppy 5.25" (2 lati, 8 sett./traccia)
`0Bh`	2	Numero di settori in una FAT
`0Dh`	2	Numero di settori per traccia (dal DOS 3.0 in poi)
`0Fh`	2	Numero di testine (dal DOS 3.0 in poi)
`11h`	2	Numero di settori nascosti (dal DOS 3.0 in poi)
`13h`	2	Word più significativa del numero di settori nascosti, a partire dal DOS 3.2 (in cui il numero settori nascosti diviene un dato a 32 bit)
`15`	4	Se la word ad offset `08h` è `0`, questo campo contiene il numero totale di settori espresso come dato a 32 bit, onde consentire il superamento del limite di 32Mb alla dimensione delle partizioni dei dischi fissi (dal dos 3.2 in poi).

Servizio 01: Media Check

Il servizio 1 è specifico dei block device driver ed è richiesto dal DOS durante operazioni di I/O diverse dalle semplici lettura o scrittura. In pratica, il sistema chiede al driver di verificare se il disco sul quale le operazioni sono in corso è stato sostituito: il driver può rispondere "SI", "NO" oppure "NON SO". Nel primo caso, il DOS non riutilizza il contenuto dei buffer[346] relativi a quella unità, richiede un

[346] Sì, sono quelli generati con l'istruzione `BUFFERS=` in `CONFIG.SYS`.

servizio 2 (descritto di seguito) e rilegge la FAT e la root directory. Se il driver risponde "NO", allora il sistema effettua l'oprazione di I/O richiesta dall'applicazione considerando valido il contenuto dei buffer. Nel caso di risposta dubitativa, il DOS opera in modo differente a seconda dello stato dei buffer: se questi contengono dati in attesa di essere scritti, il sistema li scrive (anche a rischio di danneggiare il contenuto del disco nel caso in cui esso sia stato effettivamente sostituito); in caso contrario procede come per la risposta "SI".

Il request header è strutturato come segue:

DEVICE DRIVER, SERV. 01: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1	Numero di unità disco (0 = A:)	
02h	1	Numero del servizio richiesto (1)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		
0Dh	1	Media ID byte (pag. 365)	
0Eh	1		Media Change Code: -1: Il disco e' stato sostituito 0: Forse sì, forse no, boh? 1: Il disco non è stato sostituito
0Fh	4		A partire dal DOS 3.0: puntatore far ad una stringa ASCIIZ (terminata con un NULL) contenente l'etichetta di volume del disco[347] se il Media Change Code è -1 e si è verificato un errore 0Fh (vedere pag. 361).

Il device driver può utilizzare il campo ad offset 1Fh se il bit 11 della device attribute word (pag. 359) vale 1.

Servizio 02: Build BPB

Anche questo servizio è tipico dei block device driver. Il DOS lo richiede dopo un servizio 1 (testè descritto) in due casi: se il device driver di unità restituisce un Media Change Code -1 (disco sostituito), oppure restituisce 0 (disco forse sostituito) e non vi sono buffer contenenti dati in attesa di essere scritti. La chiamata al servizio 2 consente al DOS di conoscere le caratteristiche del nuovo disco presente nell'unità e "legalizza" la sostituzione avvenuta. Il request header è utilizzato come segue:

[347] Se il disco non ha etichetta di volume è consentito restituire un puntatore far alla stringa "NO NAME".

DEVICE DRIVER, SERV. 02: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
`00h`	1	Lunghezza del request header	
`01h`	1	Numero di unità disco (`0` = `A:`)	
`02h`	1	Numero del servizio richiesto (`2`)	
`03h`	2		Stato dell'operazione (pag. 361)
`05h`	8		
`0Dh`	1	Media ID byte (pag. 365)	
`0Eh`	1	Puntatore `far` ad un buffer contenente il primo settore della FAT del disco.	
`12h`	4		Puntatore `far` al BPB (pag. 364) del disco presente nell'unità.

Il buffer il cui indirizzo è ricevuto dal driver nel campo ad offset `0Eh` non deve essere modificato dal driver stesso se l'unità gestita è formattata secondo lo standard IBM; in caso contrario il buffer può essere utilizzato come area di lavoro.

Servizio 03: IOCTL Read

Il servizio 3 deve essere supportato dal driver solo se il bit 14 della device attribute word (pag. 359) è `1`. Esso è solitamente utilizzato dalle applicazioni per ottenere dal device driver informazioni sullo stato o sulla configurazione della periferica, senza trasferimento di dati, attraverso le apposite subfunzioni dell'int 21h, funzione 44h. La struttura del request header è descritta nella tabella che segue.

Va infine osservato che le applicazioni possono leggere e scrivere dati da un character device solo dopo averlo "aperto", utilizzando il nome logico come se si trattasse di un vero e proprio file: per un esempio si veda pag. 456.

Vedere anche il servizio 12, pag. 372.

DEVICE DRIVER, SERV. 03: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
`00h`	1	Lunghezza del request header	
`01h`	1	Numero di unità disco (`0 = A:`); usato solo dai block device driver	
`02h`	1	Numero del servizio richiesto (`3`)	
`03h`	2		Stato dell'operazione (pag. 361)
`05h`	8		
`0Dh`	1	Media ID byte (pag. 365); usato solo dai block device driver	
`0Eh`	4	Puntatore `far` ad un buffer utilizzato per il trasferimento delle stringhe di controllo.	
`12h`	2	Numero di byte disponibili nel buffer	Numero di byte utilizzati nel buffer

Servizio 04: Read (Input)

Il servizio 4 consente il trasferimento di dati dalla periferica ad un buffer in memoria. Il request header è strutturato come dalla tabella che segue.

Dal momento che (per i block device driver) il DOS gestisce i settori dei dischi come settori logici numerati progressivamente a partire da 0, il device driver, per effettuare l'operazione di lettura deve trasformare (se necessario) il dato ricevuto nel campo ad offset `14h` in un numero di settore fisico, esprimibile mediante le coordinate "tridimensionali" BIOS: lato/traccia/settore[348]. A partire dal DOS 3.0, i block device driver possono servirsi dei dati gestiti mediante i servizi 13 (pag. 372) e 14 (pag. 373) per determinare se il disco sia stato sostituito "imprudentemente". Vedere anche i servizi 8 a pag. 371 e 9 a pag. 371.

[348] Il che non è proprio immediato. Va tenuto presente, infatti, che il primo settore gestito dal DOS è il Boot Sector, che ha numero logico 0: sui floppy disk esso è anche il primo settore fisico del disco, cioè il primo settore della prima traccia (o cilindro) del primo lato (o testina), e ha coordinata BIOS (lato/traccia/settore) 0,0,1. Sui dischi fissi, invece, esso è il primo settore della prima traccia del secondo lato (coordinata BIOS 0,1,1). Conoscendo il tipo di disco, il numero dei settori per traccia, il numero di tracce per lato ed il numero di lati (un hard disk ha, di solito, più di due lati, essendo costituito da più dischi montati su di un perno) con cui il disco è formattato è possibile convertire il numero logico DOS in coordinata BIOS e viceversa.

DEVICE DRIVER, SERV. 04: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1	Numero di unità disco (0 = A:); usato solo dai block device driver	
02h	1	Numero del servizio richiesto (4)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		
0Dh	1	Media ID byte (pag. 365); usato solo dai block device driver	
0Eh	4	Puntatore far ad un buffer utilizzato per il trasferimento dei dati	
12h	2	Numero di byte (character device driver) o settori (block device driver) di cui è richiesto il trasferimento	Numero di byte (character device driver) o settori (block device driver) di cui è effettuato il trasferimento
14h	2	Numero del settore logico di partenza; usato solo dai block device driver	
16h	4		A partire dal DOS 3.0: puntatore far ad una stringa ASCIIZ (terminata con un NULL) contenente l'etichetta di volume del disco se si è verificato un errore 0Fh (vedere pag. 361).

Va infine osservato che le applicazioni possono leggere e scrivere dati da un character device solo dopo averlo "aperto", utilizzando il nome logico come se si trattasse di un vero e proprio file: per un esempio si veda pag. 456.

Servizio 05: Nondestructive Read

Il servizio 5 è supportato solo dai character device driver ed è utilizzato dal DOS per ispezionare il prossimo byte presente nel flusso di dati proveniente dalla periferica (nel caso della tastiera, lo scopo specifico è individuare eventuali sequenze CTRL-C). La struttura del request header è descritta nella tabella che segue.

Se nel flusso dati vi è effettivamente un carattere in attesa di essere letto, il driver deve restituirlo nel campo ad offset 0Dh; esso deve inoltre porre a 0 il Busy Flag Bit nella word di stato dell'operazione. Se non vi è alcun carattere in attesa, il driver deve unicamente porre a 1 detto bit.

DEVICE DRIVER, SERV. 05: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1		
02h	1	Numero del servizio richiesto (5)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		
0Dh	1		Prossimo byte nel flusso di dati proveniente dalla periferica

Le applicazioni possono leggere e scrivere dati da un character device solo dopo averlo "aperto", utilizzando il nome logico come se si trattasse di un vero e proprio file: per un esempio si veda pag. 456.

Servizio 06: Input Status

Il servizio 6 è supportato solamente dai character device driver. Esso è utilizzato dal DOS per verificare se vi sono caratteri in attesa di essere letti nel flusso di dati proveniente dalla periferica ed utilizza la sola parte fissa del request header.

DEVICE DRIVER, SERV. 06: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1		
02h	1	Numero del servizio richiesto (6)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		

Il driver deve porre a 1 il Busy Flag Bit nella Status Word se non vi è alcun carattere in attesa; in caso contrario deve porre detto bit a 0. Se il device non dispone di un buffer (hardware, o implementato via software dal driver stesso) detto bit deve essere sempre 0. Questo servizio è il supporto di basso livello alla funzione 06h dell'int 21h (Check Input Status). Vedere anche il servizio 10.

Servizio 07: Flush Input Buffers

Il servizio 7 è supportato solo dai character device driver. E' utilizzato dal DOS per richiedere al driver di eliminare tutti i caratteri in attesa di lettura presenti nei buffer associati alla periferica, rendendo questi ultimi disponibili per nuove operazioni di lettura. E' utilizzata solo la parte fissa del request header:

DEVICE DRIVER, SERV. 07: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1		
02h	1	Numero del servizio richiesto (7)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		

Vedere anche il servizio 11 a pag. 371.

Servizio 08: Write (Output)

Il servizio 8 consente il trasferimento di dati da un buffer in memoria alla periferica. Circa la struttura del request header e le particolarità operative, si veda il servizio 4 (Read), a pag. 368, con l'avvertenza che il Command Code è, ovviamente, 8 invece di 4. Vedere anche il servizio 9.

Va ancora osservato che le applicazioni possono leggere e scrivere dati da un character device solo dopo averlo "aperto", utilizzando il nome logico come se si trattasse di un vero e proprio file: per un esempio si veda pag. 456.

Servizio 09: Write With Verify

Il servizio 9 è analogo al servizio 8, testè descritto, ma dopo l'operazione di scrittura il driver deve effettuare una operazione di lettura dei dati appena scritti per verificarne la correttezza. Il command code è 9. Vedere anche il servizio 4 a pag. 368.

Servizio 10: Output Status

Il servizio 10 è supportato solamente dai character device driver. Esso è utilizzato dal DOS per verificare se un'operazione di scrittura da parte del driver sia in corso. Come per il servizio 6, è utilizzata solo la parte fissa del request header (ma il Command Code è 10); il driver deve porre a 1 il Busy Flag Bit nella Status Word (pag. 361) se una operazione di scrittura è in corso. In caso contrario (condizione di *idle driver*) il bit deve essere azzerato. Questo servizio è il supporto di basso livello alla funzione 07h dell'int 21h (Check Output Status).

Servizio 11: Flush Output Buffers

Il servizio 11 è supportato solo dai character device driver. E' utilizzato dal DOS per richiedere al driver di completare tutte le operazioni di scrittura in corso, trasferendo fisicamente alla periferica tutti i dati presenti nei buffer ad essa associati e rendendo questi ultimi nuovamente disponibili per nuove operazioni di scrittura. Come per il servizio 7 (pag. 371), è utilizzata solo la parte fissa del request header, ma il Command Code è, ovviamente, 11.

Servizio 12: IOCTL Write

Il servizio 12 deve essere supportato dal driver solo se il bit 14 della device attribute word (pag. 359) è 1. Esso è solitamente utilizzato dalle applicazioni per inviare al device driver direttive di configurazione della periferica, senza trasferimento di dati, attraverso le apposite subfunzioni dell'int 21h,

funzione 44h. La struttura del request header è identica a quella presentata con riferimento al servizio 3 (pag. 367), con le sole differenze che il Command Code vale 12 e che il campo ad offset 0Eh del request header passato dal DOS al driver indica il numero di byte utilizzati nel buffer; il medesimo campo restituito dal driver esprime il numero di byte che esso ha effettivamente inviato alla periferica.

Va infine osservato che le applicazioni possono leggere e scrivere su un character device solo dopo averlo "aperto", utilizzando il nome logico come se si trattasse di un vero e proprio file: per un esempio si veda pag. 456.

Servizio 13: Device Open

Il servizio 13 deve essere supportato dai driver che hanno il bit 11 della device attribute word (pag. 359) posto a 1 ed è richiesto dal DOS a partire dalla varsione 3.0. E' utilizzata solo la parte fissa del request header:

DEVICE DRIVER, SERV. 13: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1	Numero dell'unità (solo per i block device driver)	
02h	1	Numero del servizio richiesto (13)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		

Il driver non restituisce al DOS alcun risultato: scopo principale del servizio è consentire al driver di gestire un contatore dei file aperti sulla periferica supportata: questo deve essere incrementato dal servizio in esame e decrementato dal servizio 14 (Device Close; commentato di seguito). Il contatore deve inoltre essere forzato a 0 dal driver quando sia intercettata la sostituzione del disco nell'unità (vedere i servizi 1 e 2 a pag. 365 e seguenti).

Il servizio 13 è richiesto dal DOS in corrispondenza di tutte le chiamate da parte di applicazioni alle funzioni dell'int 21h che gestiscono l'apertura o la creazione di file e l'apertura di un character device per input o output. Può essere utilizzato, dai character device driver, per inviare alle periferiche stringhe di inizializzazione, eventualmente ricevute dalle applicazioni (via DOS) attraverso il servizio 12, testè commentato.

L'apertura di un character device driver viene effettuata come una apertura di file utilizzando il nome logico del device.

Servizio 14: Device Close

Il servizio 14 è la controparte del servizio 13, testè descritto; pertanto anch'esso deve essere supportato dai driver che hanno il bit 11 della device attribute word (pag. 359) posto a 1. Come per il servizio 13, è utilizzata solo la parte fissa del request header (ma il Command Code è 14) e il driver non restituisce al DOS alcun risultato: scopo principale del servizio è la gestione, in "collaborazione" con il servizio 13, di un contatore dei file aperti sulla periferica.

Il servizio 14 è richiesto dal DOS in corrispondenza di tutte le chiamate da parte di applicazioni alle funzioni dell'int 21h che gestiscono la chiusura di file e la chiusura di un character device al termine di operazioni di input o output. Può essere utilizzato, dai character device driver, per inviare alle

periferiche stringhe di reset o reinizializzazione, eventualmente ricevute dalle applicazioni (via DOS) attraverso il servizio 12 (pag. 372).

Servizio 15: Removable Media

Il servizio 15 è supportato dai block device driver che hanno il bit 11 della device attribute word (pag. 359) posto a `1` ed è richiesto dal DOS a partire dalla versione 3.0. Esso costituisce il supporto, a basso livello, della subfunzione 08h della funzione 44h dell'int 21h (`CheckIfBlockDeviceIsRemovable`). E' utilizzata solo la parte fissa del request header:

DEVICE DRIVER, SERV. 15: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
`00h`	1	Lunghezza del request header	
`01h`	1	Numero dell'unità	
`02h`	1	Numero del servizio richiesto (`15`)	
`03h`	2		Stato dell'operazione (pag. 361)
`05h`	8		

Il driver deve porre a `1` il Busy Flag Bit (bit 11) della Status Word se il disco non è rimuovibile; in caso contrario detto bit deve essere azzerato.

Servizio 16: Output Until Busy

Il servizio 16 è supportato esclusivamente dai character device driver che hanno il bit 13 della device attribute word posto a `1` ed è richiesto dal DOS a partire dalla versione 3.0. Il suo scopo principale è permettere l'implementazione di una routine ottimizzata di output per il pilotaggio in background di periferiche (ad esempio stampanti con spooler). Il request header è strutturato come da tabella:

DEVICE DRIVER, SERV. 16: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
`00h`	1	Lunghezza del request header	
`01h`	1		
`02h`	1	Numero del servizio richiesto (`16`)	
`03h`	2		Stato dell'operazione (pag. 361)
`05h`	8		
`0Dh`	1		
`0Eh`	4	Puntatore `far` ad un buffer utilizzato per il trasferimento dei dati	
`12h`	2	Numero di byte di cui è richiesta la scrittura verso la periferica	Numero di byte effettivamente trasferiti

Il driver trasferisce i dati alla periferica, in modo continuo, sino al loro esaurimento e restituisce al sistema operativo il numero di byte effettivamente trasferiti[349].

Servizio 19: Generic IOCTL Request

Il servizio 19 è definito a partire dalla versione 3.2 del DOS, che lo richiede solo se il bit 6 della device attribute word è `1`. Esso costituisce il supporto di basso livello per la subfunzione 0Ch della funzione 44h dell'int 21h e ha quale principale finalità fornire supporto alla realizzazione di una interfaccia IOCTL con le caratteristiche desiderate dal programmatore[350]. Il request header è utilizzato come segue:

[349] Numero di byte ricevuti e numero di byte trasferiti possono validamente differire, ad esempio qualora il driver interpreti l'output da inviare alla periferica sostituendo determinate sequenze di byte con altre.

[350] Ad esempio per gestire la formattazione di dischi non-DOS, etc..

DEVICE DRIVER, SERV. 19: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
00h	1	Lunghezza del request header	
01h	1	Numero di unità disco (0 = A:); usato solo dai block device driver	
02h	1	Numero del servizio richiesto (19)	
03h	2		Stato dell'operazione (pag. 361)
05h	8		
0Dh	1	*Category Code* (Major Code)	
0Eh	1	*Function Code* (Minor Code)	
0Fh	2	Contenuto del registro SI	
11h	2	Contenuto del registro DI	
13h	4	Puntatore *far* ad un buffer (Generic IOCTL Data Packet) contenente i dati necessari al servizio	

L'interfaccia IOCTL supportata dal servizio 19 è di tipo aperto, disponibile, cioè, per future implementazioni. Tutte le sue caratteristiche e funzionalità, in ogni caso, devono essere definite dal programmatore. Il driver riceve dal DOS Category Code e Function Code, che possono essere utilizzati, rispettivamente, per identificare un tipo di driver e selezionare il sottoservizio desiderato. L'utilizzo dei valori di SI e DI copiati nel request header è libero. Infine, anche la dimensione, il formato ed il contenuto del buffer, il cui indirizzo è memorizzato nell'ultimo campo del request header, sono definiti dal programmatore a seconda della modalità scelta per l'implementazione del servizio. Già a partire dalla versione 3.3 del DOS si sono diffusi alcuni utilizzi standard del servizio, con valori predefiniti per Category Code, Function Code e formato del buffer; ciononstante, nessuno di essi è ufficialmente documentato.

Analoghe considerazioni valgono per eventuali valori restituiti dal driver al DOS: può essere utilizzato qualsiasi campo del request header (eccetto il campo di status) o lo stesso buffer.

Per un esempio di utilizzo del servizio 19, si veda pag. 450.

Servizio 23: Get Logical Device

Il servizio 23 è supportato dai block device driver nella cui device attribute word il bit 6 è posto a 1 ed è richiesto dal DOS a partire dalla versione 3.2. Esso costituisce il supporto di basso livello per la sottofunzione 0Eh della funzione 44h dell'int 21h (GetLogicalDriveMap), il cui scopo è determinare se alla medesima unità disco sia assegnata più di una lettera. La struttura del request header è la seguente:

DEVICE DRIVER, SERV. 23: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
`00h`	1	Lunghezza del request header	
`01h`	1	Numero dell'unità	Codice indicativo dell'ultima lettera utilizzata per referenziare l'unità (`1=A:`, `2=B:`, etc.).
`02h`	1	Numero del servizio richiesto (`23`)	
`03h`	2		Stato dell'operazione (pag. 361)
`05h`	8		

Se il driver supporta una sola unità disco deve restituire `0` nel campo ad offset `01h` nel request header. Vedere anche il servizio 24, commentato di seguito.

Servizio 24: Set Logical Device

Il servizio 24 è supportato dai block device driver nella cui device attribute word il bit 6 è posto a `1` ed è richiesto dal DOS a partire dalla versione 3.2. Esso costituisce il supporto di basso livello per la sottofunzione 0Fh della funzione 44h dell'int 21h (*SetLogicalDriveMap*) ed è la controparte del servizio 23, testè descritto. Il suo scopo è far conoscere al driver la successiva lettera con cui è referenziata l'unità disco. La struttura del request header è la seguente:

DEVICE DRIVER, SERV. 24: USO DEL REQUEST HEADER

OFF	DIM	REQUEST HEADER RICEVUTO	REQUEST HEADER RESTITUITO
`00h`	1	Lunghezza del request header	
`01h`	1	Numero dell'unità	
`02h`	1	Numero del servizio richiesto (`24`)	
`03h`	2		Stato dell'operazione (pag. 361)
`05h`	8		

Il numero di unità passato dal DOS al driver è relativo, con base `0`, alla prima unità supportata dal driver stesso[351].

[351] Modo criptico di dire che se, ad esempio, il driver supporta due unità disco il numero `0` indica che la successiva operazione coinvolge la prima di esse.

I DEVICE DRIVER E IL C

Sin qui la teoria: in effetti di C si è parlato poco, o per nulla. D'altra parte, il linguaggio utilizzato "per eccellenza" nello scrivere i device driver, a causa della loro rigidità strutturale e della necessità di disporre di software compatto e molto efficiente[352], è l'assembler. Proviamo a riassumere i principali problemi che si presentano al povero programmatore C:

1) I primi 18 byte del file sono occupati dal device driver header: non è perciò possibile compilare e sottoporre a linking il sorgente secondo le normali modalità C.

2) Lo startup module non può essere utilizzato.

3) La restante parte del sorgente deve essere strutturata come nel caso di un programma TSR (pag. 276), con le funzioni transienti nella parte terminale. E' inoltre opportuno utilizzare il solito trucchetto delle funzioni jolly (pag. 170) per gestire i dati globali utilizzati anche dalla parte residente.

4) Se l'operatività del driver è "pesante", è necessario dotarlo di uno stack locale, onde evitare di rompere le uova nel paniere al DOS: vedere pag. 286.

5) Mentre la strategy routine non pone particolari problemi, la interrupt routine deve essere realizzata tenendo presenti alcuni accorgimenti: innanzitutto deve essere dichiarata `far`[353] (come del resto la strategy), inoltre deve salvare tutti i registri della CPU (compresi i flag) e deve analizzare il Command Code per invocare l'opportuna funzione dedicata. Infine, in uscita, deve ripristinare correttamente i registri e gestire la restituzione di una corretta status word (pag. 361) al DOS tramite l'apposito campo del request header.

6) In fase di inizializzazione il driver ha a disposizione tutti i servizi BIOS, ma non tutti quelli DOS. In particolare non può allocare memoria. Durante la normale operatività i servizi DOS sono, in teoria, tutti disponibili, ma la logica con cui operano alcune delle funzioni di libreria C le rende comunque inutilizzabili[354]. Mancano, comunque, environment e PSP: l'assenza del Program Segment Prefix determina il caricamento dell'immagine binaria del file ad offset `00h` rispetto al Code Segment, cioè al valore di `CS`[355].

---

[352] Benché il DOS sia in grado, a partire dalla versione 3.0, di caricare device driver sotto forma di programmi `.EXE`, il formato binario puro rimane l'unico compatibile con tutte le versioni di sistema operativo. Inoltre, molte delle difficoltà insite nell'utilizzo del C per scrivere un device driver rimangono anche quando lo si realizzi sotto forma di `.EXE`.

[353] Non venga in mente a qualcuno di dichiararla `interrupt`. Il DOS invoca la interrupt routine eseguendo una `CALL FAR`; la `IRET` che chiude ogni funzione dichiarata `interrupt` provocherebbe l'estrazione dallo stack di una word di troppo, con le solite disastrose conseguenze.

[354] Esempietto: allocazione dinamica di RAM mediante `farmalloc()` e compagnia. Le funzioni di libreria per la gestione di memoria far lavorano estendendo, tramite il servizio DOS `SETBLOCK` (int 21h,4Ah) il blocco di RAM allocato al programma: una chiamata a `farmalloc()` è destinata a fallire miseramente in ogni caso, perché il blocco di memoria assegnato al driver è statico (ha comunque altri blocchi allocati al di sopra di sé; se non altro quello dell'interprete dei comandi).

7) L'utilizzo di funzioni di libreria è comunque sconsigliato in tutte le routine residenti, per i problemi già analizzati con riferimento ai TSR (pag. 289).

8) Per ogni servizio, anche se non supportato dal driver, deve essere implementata una routine dedicata: questa deve, quanto meno, invocare a sua volta una generica funzione di gestione dell'errore.

9) Il DOS assume che le routine di gestione dei servizi si comportino in completo accordo con le specifiche ad essi relative[356]: programmatore avvisato...

Ce n'è abbastanza per divertirsi e trascorrere qualche[357] notte insonne. Tuttavia vale la pena di provarci: qualcosa di interessante si può certamente fare.

Un timido tentativo

Il nostro primo device driver è molto semplice: esso non fa altro che emettere un messaggio durante il caricamento ed installare un buffer per la tastiera, lasciando residente solo quest'ultimo. E' bastato fare finta di scrivere un TSR, con l'accortezza di piazzare la funzione fittizia che riserva lo spazio per il device driver header prima di ogni altra funzione. Diamo un'occhiata al listato: i commenti sono inseriti all'interno dello stesso, onde evitare frequenti richiami, che renderebbero il testo meno leggibile.

```
/****************************************************************************

    KBDBUF.C - Barninga Z! - 01/05/94

    Device driver che installa un buffer di tastiera dell'ampiezza voluta dal
    programmatore (costante manifesta BUFDIM).

    Compilato sotto Borland C++ 3.1:

    bcc -c -mt -k- -rd kbdbuf.c
    tlink -t -c kbdbuf.obj,kbdbuf.sys

****************************************************************************/
#pragma  inline
#pragma  option -k-       // Come gia' sperimentato nei TSR e' opportuno evitare
                          // la generazione della standard stack frame laddove
                          // non serve: vedere pag. 173

#include <dos.h>                  // MK_FP(), FP_SEG(), FP_OFF()

#define  BIT_15           32768U  // 1000000000000000b

#define  BUFDIM           64      // Words (tasti) nel kbd buffer; modificare questo
```

[355] Ciò è esplicitato, a livello di sorgente, dalla direttiva assembler `ORG 00H`. Per i normali programmi eseguibili l'offset è, normalmente, `100h` e corrisponde al primo byte che segue il PSP (che, a sua volta, occupa proprio 256 byte, 100 esadecimale).

[356] La scelta della logica di implementazione è, ovviamente, libera. Si tenga però presente che anche i servizi definibili dal programmatore (si veda, ad esempio, il servizio 19 a pag. 374) devono rispettare rigorosamente le regole definite per l'interfacciamento con il sistema (struttura ed utilizzo del request header).

[357] Qualche? Beh, cerchiamo di essere ottimisti...

```
                                 // valore secondo l'ampiezza desiderata per il
                                 // buffer di tastiera

#define  BIOSDSEG         0x40   // segmento dati BIOS (tutti i dati BIOS sono
                                 // memorizzati a partire dall'indirizzo 0040:0000

// Macro definite per referenziare con semplicita' i puntatori con cui viene
// gestito il buffer di tastiera. Per i dettagli vedere pag. 304 e seguenti

#define  kbdStartBiosPtr  *(int far *)0x480       // macro per kbd buf start ptr
#define  kbdEndBiosPtr    *(int far *)0x482       // macro per kbd buf end ptr
#define  kbdHeadBiosPtr   *(int far *)0x41A       // macro per kbd buf head ptr
#define  kbdTailBiosPtr   *(int far *)0x41C       // macro per kbd buf tail ptr

// Le costanti manifeste che seguono riguardano i servizi e i codici di errore

#define  INIT             0                       // servizio 0: inizializzazione
#define  SRV_OK           0                       // servizio completato OK
#define  E_NOTSUPP        3                       // errore: serv. non implementato
#define  E_GENERIC        12                      // errore

// Macro per accesso al Request Header

#define  ReqHdrPtr        ((ReqHdr far *)*(long far *)reqHdrAddr)

// alcune typedef

typedef  void  DUMMY;                             // incrementa la leggibilita'
typedef  unsigned char  BYTE;                     // incrementa la leggibilita'

// Template di struttura per la gestione della PARTE VARIABILE del Request Header
// del servizio 0 (INIT). Vedere pag. 363

typedef struct {                             // struct per INIT Request Data
    BYTE     unit;                                // unita' (solo block device)
    void far *endOfResCode;                       // seg:off fine codice resid.
    void far *extParmPtr;                         // ptr alla command line
    BYTE     firstUnit;                           // lettera 1^ unita' (block)
} InitHdr;                                    // tipo InitHdr = struct...

// Template di struttura per la gestione del Request Header del servizio 0 (INIT)
// La parte variabile e' uno dei suoi campi

typedef  struct {                             // struct per Request Header
    BYTE    reqHdrLen;                            // lunghezza totale (Hdr+var)
    BYTE    unit;                                 // unita' (solo block device)
    BYTE    command;                              // comando richiesto
    int     status;                               // stato in uscita
    char    reserved[8];                          // riservato al DOS
    InitHdr initData;                             // dati per servizio 0 INIT
} ReqHdr;                                     // tipo ReqHdr = struct...

// Prototipi delle funzioni. Le funzioni DUMMY sono quelle fittizie, definite per
// riservare spazio ai dati residenti (vedere pag. 170)

DUMMY devDrvHdr(DUMMY);
DUMMY reqHdrAddr(DUMMY);
DUMMY kbdBuffer(DUMMY);

DUMMY helloStr(DUMMY);
DUMMY errorStr(DUMMY);
DUMMY okStr(DUMMY);

void far strategyRtn(void);
```

```
void _saveregs far interruptRtn(void);
int initDrvSrv(void);
void putstring(char far *string);

// La direttiva ORG 0 informa l'assemblatore che il programma verra' caricato in
// memoria senza PSP e senza Relocation Table

asm org 0;                                 // e' un Device Driver

// devDrvHdr() e' la funzione fittizia che definisce il Device Driver Header
// Vedere pag. 358 per i dettagli

DUMMY devDrvHdr(DUMMY)                     // Device Driver Header
{
    asm dd -1;                             // indirizzo driver successivo:
                                           // sempre -1L

    asm dw 1000000000000000b;              // attribute word: il bit 15 e' 1
                                           // perche' e' un character dev drv

    asm dw offset strategyRtn;             // offset della Strategy Routine
    asm dw offset interruptRtn;            // offset della Interrupt Routine
    asm db 'KBD     ';                     // nome logico del device: la stringa
                                           // deve essere lunga 8 caratteri
}

// reqHdrAddr() riserva spazio per l'indirizzo far del request header passato dal
// DOS in ES:BX alla strategy routine (strategyRtn())

DUMMY reqHdrAddr(DUMMY)                    // spazio per dati
{
    asm dd 0;                              // indirizzo del Request Header
}

// kbdBuffer() riserva spazio per il buffer di tastiera. E' ampio BUFDIM words
// e ogni word corrisponde ad un tasto (1 byte per scan code e 1 per ascii code)

DUMMY kbdBuffer(DUMMY)                     // spazio per dati
{
    asm dw BUFDIM dup(0);                  // keyboard buffer
}

// strategyRtn() e' la strategy routine del driver. Non deve fare altro che copiare
// il puntatore far contenuto in ES:BX nello spazio riservato dalla funzione
// fittizia  reqHdrAddr(). strategyRtn() e' dichiarata far perche' il DOS la chiama
// con una CALL FAR, assumendo che il segmento sia lo stesso del device driver e
// l'offset sia quello contenuto nell'apposito campo del device driver header

void far strategyRtn(void)                 // Driver Strategy Routine
{
    ReqHdrPtr = (ReqHdr far *)MK_FP(_ES,_BX);    // ReqHdrPtr e' una macro che
                                                 // rappresenta l'indirizzo del
                                                 // request header visto come
                                                 // puntatore far ad un dato di
                                                 // tipo ReqHdr
}

// interruptRtn() e' la interrupt routine del driver. Deve esaminare il Command Code
// (ReqHdrPtr->command) e decidere quale azione intraprendere: l'unico servizio
// implementato e' il servizio 0 (INIT). interruptRtn() e' dichiarata far _saveregs
// perche' il DOS la chiama con una CALL FAR (assumendo che il segmento sia lo
// stesso del device driver e l'offset sia quello contenuto nell'apposito campo del
// device driver header) e le tocca il compito di salvare tutti i registri. La
// dichiarazione far _saveregs e' equivalente alla dichiarazione interrupt, con la
```

```
// differenza (importantissima) che la funzione NON termina con una IRET. La
// _saveregs non e' necessaria se la funzione provvede esplicitamente a salvare
// tutti i registri (PUSH...) in entrata e a ripristinarli (POP...) in uscita

void _saveregs far interruptRtn(void)            // Driver Interrupt Routine
{
    asm pushf;
    switch(ReqHdrPtr->command) {
        case INIT:                               // servizio 0: inizializzazione

// visualizza un messaggio

            putstring((char far *)helloStr);

// chiama initDrvSrv(), la funzione dedicata al servizio 0 e memorizza nella status
// word ReqHdrPtr->status (vedere pag. 361) il valore restituito. Se questo non e' 0
// (SRV_OK), significa che l'inizializzazione e' fallita.

            if((ReqHdrPtr->status = initDrvSrv()) != SRV_OK) {

// L'inizializzazione e' fallita. Visualizza un messaggio di errore...

                putstring((char far *)errorStr);

// ...e comunica al DOS di non riservare memoria al driver (cioe' di non lasciarlo
// residente), scrivendo nel campo ad offset 0Eh del request header (vedere
// pag. 363) l'indirizzo al quale esso stesso e' caricato. Detto campo e' il secondo
// della parte variabile del request header (ReqHdrPtr->initData.endOfResCode).

                ReqHdrPtr->initData.endOfResCode = MK_FP(_CS,0);
            }
            else {

// L'inizializzazione e' OK. Visualizza un messaggio...

                putstring((char far *)okStr);

// ...e comunica al DOS l'indirizzo del primo byte di RAM che non serve alla parte
// residente del driver. In questo caso e' l'indirizzo della funzione initDrvSrv(),
// prima delle funzioni non residenti listate nel sorgente.

                ReqHdrPtr->initData.endOfResCode = initDrvSrv;
            }
            break;
        default:                                 // qualsiasi altro servizio

// Comunica al DOS che il servizio non e' supportato ponendo a 1 il bit 15 della
// status word e indicando 03h quale codice di errore nel byte meno significativo
// della medesima.

            ReqHdrPtr->status = E_NOTSUPP | BIT_15;
    }
    asm popf;                                   // fa "coppia" con la PUSH iniziale
}

// Fine della parte residente. Tutte le funzioni listate a partire da questo punto
// vengono sovrascritte dal DOS al termine della fase di inizializzazione del
// driver.

//****************************************************************************

// initDrvSrv() e' la funzione dedicata al servizio 0. Essa effettua alcuni
// controlli per determinare se il nuovo buffer di tastiera puo' essere
// installato: in caso affermativo restituisce 0, diversamente restituisce
```

```
// un valore che reppresenta un "errore non identificato" per la status word
// del request header, a cui esso e' assegnato. L'indirizzo di intDrvSrv() e'
// utilizzato per individuare la fine della parte residente.

int initDrvSrv(void)                                // INIT routine: non residente
{                                                   // perche' usata solo una volta
    register int bOff, temp;

// Installazione del nuovo buffer di tastiera, mediante l'aggiornamento dei
// puntatori con cui esso e' gestito. I calcoli ed i controlli effettuati hanno
// lo scopo di detereminare se l'installazione e' possibile: l'algoritmo non ha
// particolari implicazioni per cio' che riguarda i device driver, percio' se ne
// rimanda la discussione a pag. 384.

    bOff = FP_OFF(kbdBuffer);
    temp = FP_SEG(kbdBuffer);
    if(temp > BIOSDSEG) {
        if((temp -= BIOSDSEG) > 0xFFF)
            return(E_GENERIC | BIT_15);                 // Overflow!
        if((bOff += temp << 4) < bOff)
            return(E_GENERIC | BIT_15);                 // Overflow!
    }
    else {
        if((temp = (BIOSDSEG-temp)) > 0xFFF)
            return(E_GENERIC | BIT_15);                 // Overflow!
        if(bOff < (temp <<= 4))
            return(E_GENERIC | BIT_15);                 // Overflow!
        bOff -= temp;
    }
    if((temp = bOff+(2*BUFDIM)) < bOff)
        return(E_GENERIC | BIT_15);                     // Overflow!
    kbdStartBiosPtr = bOff;
    kbdEndBiosPtr = temp;
    kbdHeadBiosPtr = bOff;
    kbdTailBiosPtr = bOff;
    return(SRV_OK);                                 // restituzione di valore OK
}

// putstring() stampa una stringa via int 21h, funzione 09h. Impossibile usare
// puts() perche', come descritto a pag. 383, l'assenza dello startup module
// rende inconsistenti le convenzioni sul contenuto dei registri di segmento sulle
// quali si basano le funzioni di libreria.

void putstring(char far *string)                    // visualizza una stringa: non
{                                                   // residente perche' usata solo
    asm push ds;                                    // in initDrvSrv()
    asm lds dx,dword ptr string;
    asm mov ah,9;
    asm int 021h;
    asm pop ds;
}

// Fine del codice transiente. Tutte le funzioni listate a partire da questo punto
// sono funzioni fittizie il cui scopo e' riservare spazio ai dati globali necessari
// alla porzione transiente del driver. E' stato necessario ricorrere alle funzioni
// fittizie invece delle normali variabili globali C per gli stessi motivi per i
// quali e' stata implementata putstring() in luogo di puts()

//****************************************************************************

// La sequenza 0dh, 0ah, '$' che chiude ogni stringa rappresenta un CarriageReturn
// LineFeed seguito dal terminatore di stringa, che in assembler e' il '$', a
// differenza del C che utilizza lo zero binario (NULL)
```

```
DUMMY helloStr(DUMMY)                          // spazio stringa: non residente
{                                              // perche' usata solo una volta
    asm db 0dh,0ah
    asm db 'KBDBUF 1.0 - Keyboard Buffer Driver - Barninga Z! 1993.'
    asm db 0dh,0ah,'$'
}

DUMMY errorStr(DUMMY)                          // spazio stringa: non residente
{                                              // perche' usata solo una volta
    asm db 'KBDBUF: illegal buffer address. Not installed.'
    asm db 0dh,0ah,0dh,0ah,'$'
}

DUMMY okStr(DUMMY)                             // spazio stringa: non residente
{                                              // perche' usata solo una volta
    asm db 'KBDBUF: successfully installed.'
    asm db 0dh,0ah,0dh,0ah,'$'
}
```

La complessità del listato non è eccessiva; qualche precisazione va però fatta circa la modalità di compilazione e linking. Dal momento che il device driver header deve occupare i primi 18 byte del file binario, non è possibile compilare nel modo consueto, con una sintassi del tipo:

```
bcc kbdbuf.c
```

in quanto il compilatore chiamerebbe il linker richiedendo di costruire l'eseguibile inserendovi in testa lo startup module. Occorre allora compilare e consolidare il file in due passi separati, escludendo lo startup module dal processo. Inoltre, la compilazione deve generare un file `.COM`: non è possibile creare un file `.EXE` perché avrebbe in testa la Relocation Table (vedere pag. 278). La sintassi per la compilazione è allora:

```
bcc -c -mt kbdbuf.c
```

L'opzione `-c` arresta il processo alla creazione del modulo oggetto `KBDBUF.OBJ`, mentre l'opzione `-mt` richiede che la compilazione sia effettuata per il modello di memoria tiny (vedere pag. 144), adatto alla generazione di eseguibili `.COM`. L'opzione `-k-`, necessaria per evitare l'inserimento automatico del codice di gestione dello stack anche nelle funzioni in cui ciò non deve avvenire (pag. 173 per i dettagli) non è specificata sulla riga di comando, in quanto automaticamente attivata dalla direttiva

```
#pragma option -k-
```

inserita nel sorgente.

Il linking deve essere effettuato come segue:

```
tlink -c -t kbdbuf.obj,kbdbuf.sys
```

ove l'opzione `-c` forza il linker a considerare i caratteri maiuscoli diversi da quelli minuscoli[358] e l'opzione `-t` richiede la generazione di un eseguibile in formato `.COM`, il cui nome è specificato dall'ultimo parametro: `KBDBUF.SYS`. Il nostro device driver è pronto: è sufficiente, a questo punto, inserire in `CONFIG.SYS` una riga analoga alla

```
DEVICE=KBDBUF.SYS
```

[358] Il C è, notoriamente, un linguaggio case-sensitive.

indicando anche l'eventuale pathname del driver ed effettuare un bootstrap per vederlo all'opera (cioè per leggere il messaggio visualizzato durante il caricamento e per disporre di un buffer di tastiera più "spazioso" del normale).

Per dovere di chiarezza è necessario spendere alcune parole sull'algoritmo tramite il quale `initDrvSrv()` verifica la possibilità di installare il nuovo buffer. Tutti i puntatori per la gestione della tastiera sono di tipo `near` ed esprimono offset relativi al segmento `0040h`: è pertanto possibile installare la funzione fittizia `kbdBuffer()` quale nuovo buffer solamente se il suo indirizzo viene trasformato in un valore segmento:offset espresso come `0040h:offset` e, al tempo stesso, `offset+(2*BUFDIM) < FFFFh` (se tale seconda condizione non fosse rispettata, il buffer inizierebbe ad un indirizzo lecito, ma terminerebbe al di là del limite massimo di 65535 byte indirizzabile a partire dal già citato segmento di default). Riprendiamo il listato della `initDrvSrv()` per commentare con maggiore facilità l'algoritmo implementato.

```
int initDrvSrv(void)                            // INIT routine: non residente
{                                               // perche' usata solo una volta
    register unsigned bOff, temp;

    bOff = FP_OFF(kbdBuffer);
    temp = FP_SEG(kbdBuffer);
    if(temp > BIOSDSEG) {

// Se il segmento dell'indirizzo di kbdBuffer() e' maggiore di 0040h
// la differenza tra i due valori, trasformata in termini di offset (cioe'
// moltiplicata per 16) deve essere sommata all'offset di kbdBuffer().

        if((temp -= BIOSDSEG) > 0xFFF)

// Se la differenza tra il segmento di kbdBuffer() e 0040h e' maggiore di
// 0FFFh, la moltiplicazione per 16 (lo shift a sinistra di 4 bit)
// produrrebbe un overflow: inutile continuare.

            return(E_GENERIC | BIT_15);                // Overflow!
        if((bOff += temp << 4) < temp)

// Vi e' overflow anche se la somma tra il segmento shiftato e l'offset
// originario e' minore del segmento shiftato stesso (cio' significa che
// il risultato e' maggiore di FFFFh, massimo valore esprimibile da una
// variabile unsigned: i bit necessari oltre il sedicesimo sono persi).

            return(E_GENERIC | BIT_15);                // Overflow!
    }
    else {

// Se il segmento dell'indirizzo di kbdBuffer() e' minore di 0040h:
// la differenza tra i due valori, trasformata in termini di offset (cioe'
// moltiplicata per 16) deve essere sottratta all'offset di kbdBuffer().

        if((temp = (BIOSDSEG-temp)) > 0xFFF)

// Se la differenza tra 0040h e il segmento di kbdBuffer() e'maggiore di
// 0FFFh, la moltiplicazione per 16 (lo shift a sinistra di 4 bit)
// produrrebbe un overflow: inutile continuare.

            return(E_GENERIC | BIT_15);                // Overflow!
        if(bOff < (temp <<= 4))

// Vi e' overflow anche se l'offset originario e' minore del segmento
// shiftato stesso (cio' significa che kbdBuffer() e' caricata ad un
// indirizzo minore di 0040h:0000h

            return(E_GENERIC | BIT_15);                // Overflow!
```

```
        bOff -= temp;
    }

// Occorre ancora controllare che il nuovo buffer, oltre ad iniziare ad
// un indirizzo lecito, cioe' tra 0040h:0000h e 0040h:FFFFh, termini
// all'interno dello stesso intervallo.

    if((temp = bOff+(2*BUFDIM)) < bOff)
        return(E_GENERIC | BIT_15);                    // Overflow!

// Inizializzazione dei puntatori: da questo momento in poi il nuovo buffer
// di tastiera e' in funzione a tutti gli effetti.

    kbdStartBiosPtr = bOff;             // Inizio del buffer.
    kbdEndBiosPtr = temp;               // Fine del buffer.
    kbdHeadBiosPtr = bOff;              // Uguale valore per testa e coda:
    kbdTailBiosPtr = bOff;              // il buffer, all'inizio, e' vuoto.

// La initDrvSrv() segnala che tutto e' ok.

    return(SRV_OK);
}
```

Il traguardo, seppure faticosamente, è raggiunto. Non si può fare a meno di osservare, però, che i problemi da aggirare appaiono esasperanti anche per il più paziente e tenace dei programmatori... In particolare, l'impossibilità di utilizzare le funzioni di libreria, persino nella sola parte transiente del driver, è un limite davvero troppo pesante.

E' necessario pensare in grande...

Progetto di un toolkit

I maggiori ostacoli alla realizzazione di un device driver in C derivano dal fatto che in testa ad ogni programma C, dopo la compilazione, è consolidato lo startup module: questo provvede a caricare i registri di segmento (DS, ES, SS) con i valori necessari per una corretta gestione dello stack e del segmento dati (in accordo con le caratteristiche del modello di memoria[359]) effettua la scansione della command line (per generare argv e argc) ed inizializza alcune variabili globali utilizzate dalle funzioni di libreria (o da parte di esse); infine chiama la funzione main(), in uscita dalla quale richiama le funzioni che si occupano di terminare il programma in modo "pulito" (chiudendo i file aperti, etc.). Dal momento che in testa ad ogni device driver deve trovarsi il device driver header, non è possibile utilizzare lo startup module, perdendo così le fondamentali funzionalità in esso implementate.

Il problema può essere aggirato scrivendo uno startup module (o qualcosa di simile), adatto ai device driver, da utilizzare in sostituzione di quello offerto dal compilatore. E' necessario, ahinoi, scriverlo in assembler, ma la consapevolezza che si tratta di un lavoro fatto una volta per tutte è di grande conforto...

Già che ci siamo, possiamo progettare e mettere insieme anche alcune funzioni di evidente utilità (scritte in assembler, per maggiore efficienza) e raccoglierle in una libreria.

Infine ci serve un programmino in grado di modificare il device driver header del device driver compilato e consolidato: è così possibile dargli il nome logico desiderato e gli opportuni attributi senza necessità di modificare ogni volta il sorgente dello startup module e riassemblarlo.

---

[359] Con il compilatore è fornito uno startup module apposito per ogni modello di memoria supportato: al linker è indicato dal compilatore stesso quale file .OBJ costituisce il modulo appropriato. Circa i modelli di memoria e le loro caratteristiche, vedere pag. 143.

Ancora un piccolo sforzo (o meglio tre), dunque, e disporremo di un efficace (speriamo!) toolkit per la realizzazione di device driver in linguaggio C. Vediamo come fare.

Il nuovo startup module

Realizzare uno startup module non è poi così difficile, quando si abbia l'accortezza di andare a sbirciare il sorgente di quello che accompagna il compilatore[360]; nel caso dei device driver è comunque assai comodo inserirvi anche altre funzionalità particolari, quali la strategy routine e un nucleo di base della interrupt: non si tratta, perciò, solamente di un vero e proprio codice di startup (cioè di avviamento).

Particolare importanza deve essere attribuita alla definizione dei segmenti[361]: devono essere presenti tutti i segmenti definiti nello startup module originale ed è fondamentale che il segmento di codice sia definito per primo. Tutte le definizioni di segmento sono date nel file `DDSEGCOS.ASI`, riportato di seguito: si tratta di un file utilizzato in modo analogo ai file `.H` del C. Il listato è abondantemente commentato.

```
; DDSEGCOS.ASI - Barninga Z! - 1994
;
; ASSEMBLER INCLUDE FILE PER DEVICE DRIVER TOOLKIT
;
; dichiarazione dei segmenti di codice
;
; Le righe che seguono dichiarano i segmenti di codice del device driver e
; stabiliscono l'ordine in cui essi devono essere presenti nel file binario. Quasi
; tutte le definizioni sono riprese dallo startup module del TINY MODEL e riportate
; NEL MEDESIMO ORDINE. Alcune di esse non hanno utilita' nel caso dei device driver
; ma devono essere ugualmente riportate per evitare errori di compilazione o di
; runtime.

; Segmento del codice eseguibile

_TEXT               segment byte public      'CODE'            ; da startup code
_TEXT               ends

; Segmento dati inizializzati

_DATA               segment word public      'DATA'            ; da startup code
_DATA               ends

; Segmento dati costanti

_CONST              SEGMENT WORD PUBLIC      'CONST'           ; da startup code
_CONST              ENDS

; Segmenti di riferimento

_CVTSEG             SEGMENT WORD PUBLIC      'DATA'            ; da startup code
_CVTSEG             ENDS
```

[360] Il sorgente dello startup module fa quasi sempre parte della dotazione standard dei compilatori, a beneficio dell'utilizzatore che desideri personalizzarlo o ricarvarne nuove idee. Ne consegue che l'implementazione di "device driver startup module" qui presentata, derivata dallo startup code del compilatore Borland C++ 3.1, necessita sicuramente modifiche più o meno pesanti per essere utilizzata con altri compilatori.

[361] Detto in povere ed approssimative parole, i segmenti rappresentano porzioni di sorgente che devono essere sottoposte a linking in un certo ordine e indirizzate dai registri di segmento secondo predefinite modalità. Ad ogni segmento sono attribuiti un nome ed una classe: l'assemblatore raggruppa tutti i segmenti che hanno medesimo nome, seguendo l'ordine nel quale li individua nel sorgente (o nei diversi sorgenti); la classe definisce gruppi di segmenti e fornisce indicazioni sull'indirizzamento.

```
_SCNSEG            SEGMENT WORD PUBLIC      'DATA'           ; da startup code
_SCNSEG            ENDS

; Segmento dati non inizializzati

_BSS               segment word public      'BSS'
_BSS               ends

; Segmento che inizia al termine del segmento _BSS (e' una definizione dummy che
; consente di identificare con facilita' la fine del segmento _BSS.

_BSSEND            SEGMENT BYTE PUBLIC      'BSSEND'         ; da startup code
_BSSEND            ENDS

; Segmenti definiti per la gestione delle operazioni di inizializzazione e
; terminazione del programma.

_INIT_             SEGMENT WORD PUBLIC      'INITDATA'       ; da startup code
_INIT_             ENDS

_INITEND_          SEGMENT BYTE PUBLIC      'INITDATA'       ; da startup code
_INITEND_          ENDS

_EXIT_             SEGMENT WORD PUBLIC      'EXITDATA'       ; da startup code
_EXIT_             ENDS

_EXITEND_          SEGMENT BYTE PUBLIC      'EXITDATA'       ; da startup code
_EXITEND_          ENDS

; Segmento definito appositamente per i device driver. Consente di conoscere con
; facilita' l'indirizzo di fine codice.

_DRVREND           segment byte public      'DRVREND'
_DRVREND           ends

;------------------------------------------------------------------------------

; I segmenti sono tutti quanti inseriti nel gruppo DGROUP, secono lo schema del
; tiny model, che prevede un unico segmento di RAM (64 Kb) per tutti i segmenti
; del sorgente (i loro indirizzi si differenziano nell'offset). La direttiva ASSUME
; indica all'assemblatore che durante l'esecuzione tutti i registri di segmento
; verranno inizializzati con l'indirizzo di DGROUP; cio' permette all'assemblatore
; di calcolare correttamente gli offsets di tutti gli indirizzamenti nel listato.

DGROUP    group    _TEXT,       \
                   _DATA,       \
                   _CVTSEG,     \
                   _SCNSEG,     \
                   _BSS,        \
                   _BSSEND,     \
                   _INIT_,      \
                   _INITEND_,   \
                   _EXIT_,      \
                   _EXITEND_,   \
                   _DRVREND     \

assume    cs :     DGROUP,      \
          ds :     DGROUP,      \
          ss :     DGROUP,      \
          es :     DGROUP       \

;------------------------------------------------------------------------------
```

```
; Seguono le definizioni di alcune costanti manifeste. Importante e' la STKSIZE,
; che definisce l'ampiezza iniziale dello stack locale del device driver. Circa
; la gestione dello stack vedere la funzione setStack() a pag. 416 e 419.

STKSIZE          equ    512           ; dimensione in bytes dello stack locale
                                      ; 512 bytes e' il MINIMO possibile

CMDSIZE          equ    128           ; max lunghezza cmd line

SYSINIT_KB       equ    96            ; Kb riservati da Top of Mem per SYSINIT

; Seguono le definizioni delle costanti manifeste per la gestione della status
; word del request header (vedere pag. 361).

    ; Codici di status da restituire al DOS in OR tra loro. Modificano i bits del
    ; byte piu' significativo della status word.

S_SUCCESS        equ    0000h
S_ERROR          equ    8000h         ; usare in caso di errore
S_BUSY           equ    0200h
S_DONE           equ    0100h

    ; Codici di ritorno in OR coi precedenti. Indicano lo stato con cui si e' chiuso
    ; il servizio richiesto dal DOS (byte meno significativo della status word).

E_OK             equ    0000h
E_WR_PROTECT     equ    0
E_UNKNOWN_UNIT   equ    1
E_NOT_READY      equ    2
E_UNKNOWN_CMD    equ    3             ; richiesto servizio non implementato
E_CRC            equ    4
E_LENGTH         equ    5
E_SEEK           equ    6
E_UNKNOWN_MEDIA  equ    7
E_SEC_NOTFOUND   equ    8
E_OUT_OF_PAPER   equ    9
E_WRITE          equ    10
E_READ           equ    11
E_GENERAL        equ    12

; Costante manifesta per la gestione dello stack. Ha un fondamentale ruolo
; nell'implementazione del meccanismo che consente al device driver di
; modifcare la dimensione dello stack iniziale in fase di inizializzazione.
; Vedere setStack() a pag. 416 e 419 per i particolari.

; SE SI INTENDE MODIFICARE LA GESTIONE DELLO STACK TRA L'ISTRUZIONE CHE IN
; driverInit() COPIA SP IN __savSP1 E QUELLA CHE IN setStack() COPIA SP IN __savSP2,
; LE COSTANTI MANIFESTE CHE SEGUONO DEVONO ESSERE MODIFICATE DI CONSEGUENZA!!

NPA_SP_DIFF      equ    8     ; differenza tra SP prima delle PUSH dei parms
                              ; di main() e SP in ingresso a setStack() nel
                              ; caso di main(void) e senza variabili auto.
```

Il testo del file `DDSEGCOS.ASI` è incluso in tutti i listati assembler laddove sia presente la riga

```
include ddsegcos.asi
```

in caratteri maiuscoli o minuscoli: a differenza del C, l'assembler non distingue, per default, maiuscole e minuscole. Vediamo ora il listato dello startup module per i device driver:

```
; DDHEADER.ASM - Barninga Z! - 1994
;
```

```
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - STARTUP
;
;
; device driver STARTUP MODULE. Da linkare in testa all'object prodotto dal
; compilatore e contenente la reale implementazione C del driver.

; DDHEADER.ASM implementa il device driver header, e le strategy e interrupt
; routines del device driver. Sono inoltre inserite incluse alcune routines di
; inizializzazione e le dichiarazioni delle variabili globali (alcune delle quali
; non pubblicate e quindi non visibili in C) per il supporto delle librerie
; C, in base allo startup module dei normali programmi. Si noti che i nomi C
; devono ignorare l'underscore (o il primo degli underscore) in testa ai nomi
; assembler.

include DDSEGCOS.ASI                        ; definizioni dei segmenti!!!

org 0                                       ; no PSP: e' un device driver

; Dichiarazione dei simboli esterni: sono dichiarate le variabili e le funzioni
; referenziate ma non definite in questo sorgente. Tutte le funzioni sono near
; dal momento che si lavora con il modello TINY.

          extrn  __stklen         : word      ; da libreria C: gestione stack.
                                              ; Vedere setStack(), pag. 416 e 419.

; seguono le dichiarazioni delle funzioni che gestiscono i vari servizi del device
; driver. La loro implementazione e' libera, ma il nome nel sorgente C deve essere
; identico a quello qui riportato (salvo l'underscore iniziale). Se il sorgente C
; non implementa una o piu' delle seguenti funzioni, all'eseguibile viene linkata
; la corrispondente dummy function inclusa nella libreria (pag. 397 e seguenti).
; Fa eccezione _driverInit() (DDINIT.ASM), che implementa il servizio INIT standard
; e chiama la init() definita dal programmatore nel sorgente C.

          extrn  _driverInit      : near
          extrn  _mediaCheck      : near      ; servizi del driver, da
          extrn  _buildBPB        : near      ; implementare in C. Le
          extrn  _inputIOCTL      : near      ; implementazioni assembler in
          extrn  _input           : near      ; libreria servono solo come
          extrn  _inputND         : near      ; placeholders per quelle non
          extrn  _inputStatus     : near      ; realizzate in C e non fanno che
          extrn  _inputFlush      : near      ; chiamare errorReturn(), definita
          extrn  _output          : near      ; in questo stesso sorgente
          extrn  _outputVerify    : near
          extrn  _outputStatus    : near
          extrn  _outputFlush     : near
          extrn  _outputIOCTL     : near
          extrn  _deviceOpen      : near
          extrn  _deviceClose     : near
          extrn  _mediaRemove     : near
          extrn  _outputBusy      : near
          extrn  _genericIOCTL    : near
          extrn  _getLogicalDev   : near
          extrn  _setLogicalDev   : near
          extrn  _endOfServices   : near

;-------------------------------------------------------------------------------

_TEXT        segment                 ; inizio del segmento _TEXT (codice)

;-------------------------------------------------------------------------------

; Il codice (segmento _TEXT) - e quindi il file binario - inizia con il
; device driver header. Notare il primo campo (4 bytes) posto a -1 (FFFFFFFFh).
; La device attribute word (secondo campo) e il logical name (ultimo campo)
```

```
; sono "per default" posti a 0 e, rispettivamente, 8 spazi. Dovranno essere
; settati dopo il linking, utilizzando la utility  descritta a pag. 431.
; E' fondamentale che il device driver header sia il primo oggetto definito
; nel segmento del codice.

            public _DrvHdr                      ; header del device driver
_DrvHdr     dd      -1
            dw      0                           ; DA SETTARE CON DRVSET
            dw      offset _TEXT:_Strategy
            dw      offset _TEXT:_Interrupt
            db      8 dup(32)                   ; DA SETTARE CON DRVSET

            ;------------------------------------------------------------------

; Tabella dei servizi del device driver. E' utilizzata dalla interrupt routine
; (listata poco piu' avanti) per individuare la funzione da chiamare a seconda
; del servizio richiesto dal DOS. Il posizionamento della tabella all'inizio del
; modulo forza l'assemblatore a referenziare per prime le funzioni dei servizi
; (solo Strategy() e Interrupt() sono referenziate prima di esse, nello header):
; in tal modo esse sono ricercate per prime nelle librerie (se non definite nel
; sorgente) e linkate all'eseguibile prima di tutte le funzioni di libreria C.
; Cio' permette l'utilizzo di _endOfServices(), come dichiarata al termine della
; tabella stessa.

_FuncTab    dw      offset _TEXT:_driverInit          ;  0
            dw      offset _TEXT:_mediaCheck          ;  1
            dw      offset _TEXT:_buildBPB            ;  2
            dw      offset _TEXT:_inputIOCTL          ;  3
            dw      offset _TEXT:_input               ;  4
            dw      offset _TEXT:_inputND             ;  5
            dw      offset _TEXT:_inputStatus         ;  6
            dw      offset _TEXT:_inputFlush          ;  7
            dw      offset _TEXT:_output              ;  8
            dw      offset _TEXT:_outputVerify        ;  9
            dw      offset _TEXT:_outputStatus        ; 10   A
            dw      offset _TEXT:_outputFlush         ; 11   B
            dw      offset _TEXT:_outputIOCTL         ; 12   C
            dw      offset _TEXT:_deviceOpen          ; 13   D
            dw      offset _TEXT:_deviceClose         ; 14   E
            dw      offset _TEXT:_mediaRemove         ; 15   F
            dw      offset _TEXT:_outputBusy          ; 16  10
            dw      offset _TEXT:_unSupported         ; 17  11
            dw      offset _TEXT:_unSupported         ; 18  12
            dw      offset _TEXT:_genericIOCTL        ; 19  13
            dw      offset _TEXT:_unSupported         ; 20  14
            dw      offset _TEXT:_unSupported         ; 21  15
            dw      offset _TEXT:_unSupported         ; 22  16
            dw      offset _TEXT:_getLogicalDev       ; 23  17
            dw      offset _TEXT:_setLogicalDev       ; 24  18

; E' dichiarata una label (etichetta) il cui indirizzo (offset) indica la fine
; della tabella delle funzioni di servizio.

            public  __endOfSrvc
__endOfSrvc label   word                        ; off DGROUP:fine f() servizi

; E' dichiarata una funzione dummy inserita in libreria dopo tutte le funzioni
; di servizio: il suo indirizzo indica, nel file binario, la fine del codice
; eseguibile delle funzioni di servizio. Vedere anche endOfServices() a pag. 410.

            dw      offset _TEXT:_endOfServices  ; dummy func per segnare
                                                 ; indirizzo fine ultima
                                                 ; f() di servizio.
```

```
            ;----------------------------------------------------------------

; L'espressione $-_FuncTab-2 calcola la distanza (in bytes) tra l'inizio della
; tabella dei servizi e l'indirizzo attuale, meno due bytes. In pratica si ottiene
; la dimensione della tabella dei puntatori alle funzioni di servizio e, dal momento
; che ogni puntatore occupa due bytes (tutti puntatori near), __FuncIDX__ puo'
; fungere da indice per individuare la funzione da chiamare a seconda del servizio
; richiesto dal DOS.

__FuncIDX__  dw      $ - _FuncTab - 2       ; max indice (word off) in tabella
                                            ; il 2 e' per la f() dummy

; Seguono le definizioni di diverse variabili globali. Solo quelle per le quali e'
; specificata la clausola PUBLIC sono visibili da C. Sono definite anche alcune
; labels (etichette) che servono per gestire indirizzi senza utilizzare veri e
; propri puntatori. Tutte le variabili e labels pubblicate al C sono dichiarate
; nell'include file della libreria toolkit BZDD.H (vedere pag. 397 e seguenti).

             public _RHptr
_RHptr       dd      0                      ; puntatore far al Request Header

_DosStkPtr   dd      0                      ; ptr far a DOS Stack (SS:SP ingresso)

             public _DrvStk
_DrvStk      db STKSIZE dup(0)              ; stack locale del driver

             public _DrvStkEnd
_DrvStkEnd   label word                     ; fine dello stack (e' solo una label)

; Il significato e l'uso di alcune delle variabili definite di seguito sono
; commentati a pagina 445.

; Kb mem conv installati (usata nel codice di inizializzazione per l'int 12h). Si
; tratta di un dato raccolto per comodita' del programmatore.

             public __systemMem
__systemMem  dw    0

; Di seguito sono definite un'etichetta ed una variabile. La prima rappresenta un
; sinonimo della seconda e possono essere utilizzate da C come se fossero la stessa
; cosa. La variabile contiene l'indirizzo di segmento al quale e' caricato il
; driver (CS); il sinonimo _psp e' definito per analogia con la libreria C, ma
; il driver non ha un PSP. Questa e' la parte segmento dell'indirizzo al quale
; si trova il device driver header; percio' RHptr vale _psp:0000 o _baseseg:0000.

             public __psp
__psp        label word

             public __baseseg
__baseseg    dw    0                        ; segmento di base del DevDrv (CS)

; Di seguito sono definite due variabili di comodita' per il programmatore. I device
; driver non hanno un far heap in cui allocare memoria far: _farMemBase e
; _farMemTop sono gli indirizzi far dell'inizio e, rispettivamente, della fine della
; memoria libera oltre il driver. Va tenuto presente che in quell'area di RAM ci
; sono parti di DOS attive: essa puo' percio' essere usata a discrezione, ma con
; prudenza.

             public __farMemBase
__farMemBase dd    0                        ; puntatore far alla memoria libera
                                            ; oltre il driver in init()

             public __farMemTop
__farMemTop  dd    0                        ; puntatore far alla fine della memoria
```

```
                                        ; libera oltre il driver in init().

; _cmdLine e' una copia della command line del driver in CONFIG.SYS, a partire dal
; carattere che segue "DEVICE=". In pratica e' un array di char.

            public __cmdLine
__cmdLine   db CMDSIZE dup(0)           ; copia locale della command line

; _cmdArgsN e _cmdArgs equivalgono a argc e argv. Il massimo numero di puntatori che
; _cmdArgs puo' contenere e' 64, perche' una command line e' al massimo di 128
; caratteri e ogni argomento occupa almeno 2 bytes (1 carattere + 1 spazio).

            public __cmdArgsN
__cmdArgsN  dw    0                     ; come argc

            public __cmdArgs
__cmdArgs   db CMDSIZE dup(0)           ; array puntat. ad argom. cmd line

; Definizione di alcuni puntatori per comodita' del puntatore. Gli indirizzi (near)
; sono calcolati mediante labels definite in coda a questo stesso sorgente.

            public __endOfCode
__endOfCode dw    offset DGROUP:_ecode@     ; offset (CS:) della fine codice

            public __endOfData
__endOfData dw    offset DGROUP:_edata@     ; offset (CS:) della fine dati

            public __endOfDrvr
__endOfDrvr dw    offset DGROUP:_edrvr@     ; offset (CS:) fine spazio driver

; Variabili per la gestione dello stack. Lo stack locale DrvStk e' rilocabile
; (vedere setStack() a pag. 416 e 419): qualora esso, a discrezione del
; programmatore, venga effettivamente rilocato, lo spazio di STACKDIM bytes occupato
; inizialmente puo' essere riutilizzato come un normale array di char, il cui
; indirizzo (near) e' _freArea. La sua effettiva dimensione e' _freAreaDim. Le altre
; variabili sono utilizzate da setStack() e sono pubblicate al C con finalita' di
; debugging.

            public  __freArea
__freArea   dw    0                     ; offset ex-stack per riutilizzo

            public __freAreaDim
__freAreaDim dw   0                     ; dimensione ex-stack

            public __savSP1
__savSP1    dw    0                     ; SP prima di push parms per init()

            public __savSP2
__savSP2    dw    0                     ; SP all'ingresso di SetStack()

            public __newTOS
__newTOS    dw    0                     ; offset del nuovo Top Of Stack

; Seguono definizioni di variabili date per analogia con lo startup code dei normali
; programmi C.

            public __version
__version   label word                  ; versione e revisione DOS

            public __osversion
__osversion label word                  ; versione e revisione DOS

            public __osmajor
__osmajor   db    0                     ; versione DOS
```

```
            public __osminor
__osminor   db    0                     ; revisione DOS

            public __StartTime
__StartTime dd    0                     ; clock ticks allo startup

            public _errno
_errno      dw    0                     ; codice di errore

            public ___MMODEL
___MMODEL   dw    0                     ; tiny model

            public _DGROUP@
_DGROUP@    dw    0                     ; segmento del gruppo DGROUP

; Le variabili definite di seguito sono necessarie alle funzioni di libreria C per
; la gestione dello heap (malloc(), etc.). Quelle relative al far heap non sono
; inizializzate in quanto ai device driver non e' mai possibile effettuare
; allocazioni far mediante farmalloc(). Le allocazioni dello heap sono effettuate
; all'interno dello stack, secondo lo schema del modello TINY.

            public ___heapbase
___heapbase dw    offset _DrvStk        ; inizio near heap

            public ___brklvl
___brklvl   dw    offset _DrvStk        ; attuale fine near heap

            public __heapbase
__heapbase  dd    0                     ; inizio far heap
                                        ;
            public __brklvl             ;
__brklvl    dd    0                     ; inizio far heap
                                        ;
            public __heaptop            ;
__heaptop   dd    0                     ; fine far heap

            ;-----------------------------------------------------------------

; Inizia qui il codice eseguibile. Siamo sempre nell'ambito del code segment (il che
; e' normale per le routine, un po' meno per le variabili appena definite. Si veda
; pero' quanto detto a proposito delle variabili nel code segment, a pag. 170.

; Ecco la strategy routine. Essa non fa altro che salvare l'indirizzo del request
; header, passato dal DOS in ES:BX, nel puntatore far RHptr. Si noti che Strategy()
; deve essere una funzione far; inoltre essa non e' pubblicata al C, in quanto si
; tratta di una routine interna al driver, che non deve mai essere chiamata da C.

_Strategy   proc far

            mov word ptr cs:[_RHptr],bx      ; salva ptr al Request Header
            mov word ptr cs:[_RHptr+2],es    ; prima offset poi segmento

            ret

_Strategy   endp

            ;-----------------------------------------------------------------

; Ed ecco la interrupt routine: come si vede, ha una struttura semplice. Essa
; attiva lo stack locale e salva i registri, dopodiche' scandisce la tabella
; dei servizi: se il servizio e' 0 (inizializzazione) richiama driverInit(),
; definita oltre in questo sorgente, la quale a sua volta chiama la init() del
; sorgente C. Se il servizio non e' definito chiama errorReturn(), definita oltre
```

```
; in questo sorgente, passandole E_UNSUPPORTED quale codice di errore, altrimenti
; chiama la funzione dedicata al servizio. Se il sorgente C implementa una funzione
; con quel nome (vedere la tabella in testa a questo sorgente) e' invocata proprio
; quella funzione, altrimenti e' chiamata la corrispondente funzione dummy della
; libreria toolkit. Al termine delle operazioni, Interrupt() ripristina i registri
; e lo stack DOS e termina restituendo il controllo al sistema. Interrupt(), come
; Strategy(), deve essere far e non e' pubblicata al C, che non la puo' invocare.

_Interrupt   proc far

             ; E' bene che il driver utilizzi un proprio stack, per evitare di
             ; sottarre risorse al DOS: quindi bisogna modificare SS:SP in modo
             ; che puntino allo stack del driver e non piu' a quello DOS. E' ovvio
             ; che l'indirizzo dello stack DOS (SS:SP) deve essere salvato (NON
             ; SULLO STACK STESSO!) per poterlo ripristinare in uscita. Allo
             ; scopo e' usata la variabile DosStkPtr.

             mov word ptr cs:[_DosStkPtr],sp    ; salva SS:SP (ptr a stack DOS)
             mov word ptr cs:[_DosStkPtr+2],ss  ; prima offset poi segmento

             ; Lo stack locale viene attivato caricando SS:SP con l'indirizzo
             ; della fine (lo stack e' usato dall'alto in basso!) dell'area
             ; allo scopo riservata. Percio' in SS e' caricato CS (nel modello
             ; TINY il segmento di stack e' lo stesso del codice) e in SP e'
             ; caricato l'offset della label DrvStkEnd, che indica la fine di
             ; DrvStk, l'array di STKSIZE bytes dedicato allo scopo.

             mov _DGROUP@,cs                    ; stack pointers settati allo
             mov ss,_DGROUP@                    ; stack locale (_DGROUP@ e'
             mov sp,offset _DrvStkEnd           ; una variabile di comodo)

             ; A questo punto si puo' usare lo stack locale per salvare tutti i
             ; registri e generare una standard stack frame, cioe' il settaggio
             ; di SS, SP e BP secondo le convenzioni C in ingresso alle funzioni
             ; (vedere pag. 160).

             push bp                            ; genera standard stack frame
             mov bp,sp

             push ax                            ; ...e salva tutti i registri
             push bx                            ; senza piu' intaccare lo
             push cx                            ; stack dos
             push dx
             push si
             push di
             push ds
             push es
             pushf

             ; Tutti i registri di segmento sono uguagliati a CS (tiny model); SS
             ; e' gia' stato settato.

             mov bx,cs
             mov ds,bx
             mov es,bx

             ; Qui viene esaminato il servizio richiesto ed e' lanciata la funzione
             ; corrispondente: il numero di servizio, letto ad offset 2 nel request
             ; header, e' moltiplicato per 2 (i puntatori alle funzioni sono near e
             ; percio' ciascuno occupa 2 bytes; quindi in tal modo si ottiene
             ; direttamente l'offset nella tabella FuncTab del puntatore alla
             ; funzione da lanciare). Se il risultato della moltiplicazione e'
             ; maggiore di _FuncIDX__ (massimo indice), e' chiamata la funzione
             ; unSupported() (definita oltre in questo sorgente), altrimenti si
```

```
            ; salta alla label EXECUTE.

            push ds
            lds si,DGROUP:_RHptr        ; DS:SI punta al Request Header
            mov al,[si+2]               ; AL = servizio (campo a offset 2)
            pop ds
            shl al,1                    ; per 2 (offsets in _FuncTab sono words)
            xor ah,ah                   ; AX = offset in termini di words
            cmp ax,__FuncIDX__          ; ** MAX VALORE DEL COMMAND BYTE x 2 **
            jle EXECUTE
            call _unSupported
            jmp EXITDRIVER
EXECUTE:

            ; Il servizio richiesto e' lecito: la coppia DS:SI e' caricata con
            ; l'indirizzo del puntatore alla funzione opportuna e si procede
            ; alla chiamata con una tecnica analoga all'indirezione di puntatore
            ; a funzione.

            mov si,offset _FuncTab
            add si,ax                   ; DS:SI punta al puntat. a funz. in _FuncTab
            call word ptr [si]          ; chiama funzione void f(void)
EXITDRIVER:

            ; Alla fine delle operazioni Interrupt() setta a 1 il bit 8 della
            ; status word nel request header: si presume che tutte le funzioni
            ; dedicate ai servizi restituiscano (e percio' il compilatore lo
            ; carichera' in AX) il valore della status word stessa.

            or ax,S_DONE                ; segnala fine operazioni
            push ds
            lds si,DGROUP:_RHptr        ; DS:SI punta al Request Header
            mov [si+3],ax               ; valorizza lo STATUS di ritorno
            pop ds

            ; Uscita da Interrupt(): tutti i registri sono ripristinati, viene
            ; eliminata la standard stack frame ed e' ricaricato in SS:SP
            ; l'indirizzo dello stack DOS prelevandolo da DosStkPtr. La funzione
            ; termina con una normale RET (far, dato che Interrupt() e' dichiarata
            ; tale) e NON con una IRET.

            popf                        ; ripristina tutti i registri
            pop es                      ; estraendone i valori di ingresso
            pop ds                      ; dallo stack locale
            pop di
            pop si
            pop dx
            pop cx
            pop bx
            pop ax

            pop bp                      ; rispristina BP (standard stack frame)

            mov sp,word ptr cs:[_DosStkPtr]    ; ripristina SS:SP (ora puntano
            mov ss,word ptr cs:[_DosStkPtr+2]  ; nuovamente allo stack DOS)

            ret                         ; non e' un interrupt!

_Interrupt  endp

            ;-----------------------------------------------------------------

; La funzione errorReturn() e' pubblicata al C e deve essere usata per segnalare
; che un servizio e' terminato in errore. Accetta come parametro un intero il cui
```

```
; byte meno significativo rappresenta il codice di errore e lo restituisce dopo
; avere settato a 1 il bit 15. Vedere pag. 361 (status word) e pag. 386
; (DDSEGCOS.ASI).

             public _errorReturn                 ; f() per restituzione errore
_errorReturn proc near              ; int errorReturn(int errcod);
                                    ; procedura per restituzione codice
             push bp                ; di errore secondo costanti manifeste
             mov bp,sp              ; Il parm e' referenziato [BP+4] infatti
             mov ax,[bp+4]          ; [BP+0] = BP e [BP+2] = IP (per la ret)
             or ax,S_ERROR          ; setta bit di errore
             pop bp

             ret

_errorReturn endp

             ;------------------------------------------------------------

; La funzione unSupported(), pubblicata al C, e' dedicata a restituire lo stato di
; errore per servizio non supportato. Non fa altro che chiamare errorReturn() con
; l'appropriato codice di errore.

             public _unSupported              ; f() per servizio non supportato
_unSupported proc near              ; int unSupported(void);

             mov ax,E_UNKNOWN_CMD
             push ax
             call _errorReturn           ; restituisce codice errore
             add sp,2

             ret

_unSupported endp

;-------------------------------------------------------------------------------

_TEXT        ends                                ; Fine del segmento di codice

;-------------------------------------------------------------------------------

; Labels pubbliche per individuare gli offsets dei segmenti (non possono
; essere dichiarate in DDSEGCOS.ASI) perche' devono essere dichiarate una
; volta soltanto.

                   ;------------------------------------------------------

_DATA              segment word public      'DATA'
                   public  _ecode@                     ; fine codice (_TEXT)
_ecode@            label   byte
_DATA              ends

                   ;------------------------------------------------------

_CVTSEG            SEGMENT WORD PUBLIC      'DATA'           ; da startup code
                   public  __RealCvtVector
__RealCvtVector    label   word
_CVTSEG            ENDS

                   ;------------------------------------------------------

_SCNSEG            SEGMENT WORD PUBLIC      'DATA'           ; da startup code
                   public  __ScanTodVector
__ScanTodVector    label   word
```

```
_SCNSEG             ENDS

                    ;------------------------------------------------------------

_BSS                segment word public      'BSS'            ; da startup code
                    public  _bdata@                     ; inizio BSS
_bdata@             label   byte
_BSS                ends

                    ;------------------------------------------------------------

_BSSEND             SEGMENT BYTE PUBLIC      'BSSEND'         ; da startup code
                    public  _edata@                     ; fine BSS
_edata@             label   byte
_BSSEND             ENDS

                    ;------------------------------------------------------------

_DRVREND            segment byte public      'DRVREND'
                    public  _edrvr@                     ; fine driver
_edrvr@             label   byte
_DRVREND            ends

;--------------------------------------------------------------------------------

              end
```

Assemblando il sorgente con il comando

```
tasm -ml ddheader.asm
```

si ottiene il file `DDHEADER.OBJ`, che deve essere consolidato in testa al file `.OBJ` prodotto dal compilatore a partire dal sorgente C implementante il device driver, al fine di ottenere il file binario caricabile dal sistema operativo. L'opzione `-ml` impone all'assemblatore di distinguere le maiuscole dalle minuscole nei nomi di segmento, di variabile e di funzione, onde consentirne l'utilizzo in C secondo le consuete convenzioni del linguaggio.

Il primo ostacolo è alle nostre spalle: nello scrivere un device driver in C possiamo quasi dimenticarci dell'esistenza del nuovo startup module, così come scrivendo normali programmi ignoriamo del tutto quello originale.

La libreria di funzioni

Perché una libreria di funzioni? Per comodità, ma, soprattutto, per ragioni di efficienza. Dal momento che un device driver è largamente assimilabile ad un programma TSR ed è sottoposto ai medesimi vincoli per quel che riguarda l'occupazione di memoria, inserire in una libreria alcune funzioni utilizzate solo durante la fase di inizializzazione può consentire di confinarle nella porzione transiente del codice.

Vale la pena di presentare per prima la funzione `driverInit()`, utilizzata una sola volta durante l'inizializzazione del driver: va precisato che inserirla nella libreria permette di evitarne la permanenza in memoria dopo la fase di inizializzazione stessa, in quanto essa è inclusa dal linker nell'eseguibile solo dopo le funzioni definite nel sorgente C. Tuttavia, dal momento che essa è referenziata nello startup module prima delle funzioni di gestione dei servizi, è indispensabile che queste ultime siano tutte definite nel sorgente C (e non siano utilizzate le dummy function presenti in libreria) perché la `_driverInit()` possa essere considerata transiente senza pericolo di scartare routine residenti.

```
; DDINIT.ASM - Barninga Z! - 1994
;
```

```
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione driverInit() per l'inizializzazione standard del device driver

include   ddsegcos.asi

; dichiarazione di init(). Equivale alla main() dei normali programmi C, assente
; nei device driver. E' richiamata dalla interrupt routine quando il DOS
; richiede il servizio 0, cioe' al caricamento del driver. Tutte le operazioni
; di inizializzazione devono percio' essere gestite nella init(), che deve essere
; presente in tutti i device driver scritti utilizzando il toolkit: vedere pag. 441
; per i dettagli. Ovviamente non deve esserci main().

          extrn  _init            : near      ; user defined! E' la "main()" del
                                              ; device driver (pag. 441).

; Dichiarazione della funzione di libreria toolkit setupcmd() (pag. 421), che
; effettua la scansione della command line e genera argc e argv per la init().

          extrn  _setupcmd        : near
; Altri simboli esterni dallo startup module

           extrn   __version      : word
           extrn   __stklen       : word
           extrn   __baseseg      : word
           extrn   __systemMem    : word
           extrn   __farMemTop    : dword
           extrn   __farMemBase   : dword
           extrn   __StartTime    : dword
           extrn   __cmdArgs      : word
           extrn   __cmdArgsN     : word
           extrn   __savSP1       : word
           extrn   _DGROUP@       : word

           extrn   _edrvr@        : byte
           extrn   _bdata@        : byte
           extrn   _edata@        : byte

;-------------------------------------------------------------------------------

_TEXT        segment

;-------------------------------------------------------------------------------

; driverInit() e' il vero e proprio startup code del driver. Dal momento che il
; suo indirizzo si trova ad offset 0 nella FuncTab (e' il primo puntatore nella
; tabella) essa e' chiamata da Interrupt() in corrispondenza del servizio 0 e
; procede alla inizializzazione del driver. driverInit() si occupa delle operazioni
; di inizializzazione standardizzate per tutti i driver: al termine di queste
; chiama init(), che deve essere definita dal programmatore nel sorgente C, la
; quale esegue le operazioni peculiari per quel driver. Il programmatore C deve
; gestire l'inizializzazione dedicata allo specifico driver esclusivamente con
; init(), di cui si parla a pagina 441.

             public  _driverInit
_driverInit  proc near          ; driverInit(): simula parte dello startup code

             mov ah,30h
             int 21h                              ; richiede versione dos
             mov __version,ax                     ; e la salva

             mov __stklen, STKSIZE                ; dimensione stack

             mov word ptr __baseseg,cs            ; segmento di caricamento
```

```
            ; L'algoritmo che segue non trova corrispondenza nello startup code
            ; dei normali programmi. Viene calcolato EMPIRICAMENTE il confine
            ; superiore della memoria libera oltre il driver: in caso di problemi
            ; può essere necessario aumentare il valore di SYSINIT_KB, costante
            ; manifesta definita in DDSEGCOS.ASI, che esprime il numero di Kb
            ; riservati alla routine SYSINIT del DOS (vedere pag. 353). E' poi
            ; individuato l'offset della fine dello spazio occupato dal driver e,
            ; a partire da quello, l'indirizzo far dell'inizio della memoria
            ; libera oltre il driver. Lo spazio tra gli indirizzi far _farMemBase
            ; e _farMemTop e' a disposizione del driver per qualsiasi utilizzo,
            ; purche' limitato alla sola fase di inizializzazione, in quanto
            ; quella memoria sara' successivamente usata dal DOS (e non e'
            ; comunque gestibile via farmalloc(), etc.).

            int 12h                             ; Kb RAM instal. (AX); <= 640
            mov __systemMem,ax                  ; salva valore restituito
            sub ax,SYSINIT_KB                   ; protegge spazio per SYSINIT
            mov cx,6
            shl ax,cl                           ; Kb * 64 = Seg esa
            mov word ptr __farMemTop+2,ax       ; top della memoria far libera

            mov ax,offset DGROUP:_edrvr@        ; offset fine spazio driver
            mov cx,4
            shr ax,cl                           ; off / 16 = normalizzazione
            inc ax                              ; annulla arrotondamento
            add ax,word ptr _DGROUP@            ; segmento normalizzato
            mov word ptr __farMemBase+2,ax      ; base far free mem e' il primo
                                                ; seg:0000 oltre il driver

            ; Seguono nuovamente operazioni derivate dal normale startup code.

            mov ah,0
            int 1ah                             ; BIOS time ticks
            mov word ptr __StartTime,dx         ; per la funzione C clock()
            mov word ptr __StartTime+2,cx
            or  al,al                           ; midnight flag settato?
            jz  NOT_MIDNIGHT
            push es
            mov ax,40h                          ; setta BIOS midnight flag
            mov es,ax                           ; all'indirizzo 0040:0070
            mov bx,70h
            mov byte ptr es:[bx],1
            pop es

NOT_MIDNIGHT:

            mov di,offset DGROUP:_bdata@     ; da startup code: azzera area
            mov cx,offset DGROUP:_edata@     ; BSS. (ES = CS = DGROUP)
            sub cx,di                        ; CX = dist. _bdata@-_edata@
            xor ax,ax
            cld
            rep stosb

            ; A questo punto e' chiamata la funzione di libreria toolkit
            ; setupcmd() (pag. 421), che effettua la scansione della command line
            ; data in CONFIG.SYS e setta _cmdArgsN e _cmdArgs, equivalenti a argc
            ; e, rispettivamente, argv.

            call _setupcmd                   ; scansione command line
            mov bx,offset __cmdArgs
            push bx                          ; analogo ad argv
            push __cmdArgsN                  ; analogo ad argc
```

```
            mov __savSP1,sp                  ; per setStack(): deve essere
                                             ; l'ultima prima di call _init

            ; Infine e' chiamata init(): il controllo dell'inizializzazione passa
            ; al sorgente C. Vedere pag. 441.

            call _init                       ; int init(...) USER DEFINED

            add sp,4                         ; pulizia stack

            ; Al termine dell'inizializzazione sono (per prudenza) azzerati i
            ; puntatori all'inizio e alla fine della memoria far disponibile oltre
            ; il driver, dal momento che, come sottolineato poco fa, dopo la fase
            ; di bootstrap essa non e' piu' disponibile.

            mov word ptr __farMemBase,0      ; dopo init() non ha piu'
            mov word ptr __farMemBase+2,0    ; alcun significato
            mov word ptr __farMemTop,0       ; dopo init() non ha piu'
            mov word ptr __farMemTop+2,0     ; alcun significato

            ret                              ; torna a Interrupt()

_driverInit  endp

;--------------------------------------------------------------------------------

_TEXT        ends

;--------------------------------------------------------------------------------

            end
```

La libreria comprende poi le funzioni per la gestione dei servizi non implementati dal driver, le quali non fanno altro che chiamare la `unSupported()` dello startup module (pag. 396): vediamole ordinate per codice crescente di servizio.

```
; DDMEDCHE.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione mediaCheck() dummy usata solo se non implementata in C

include   ddsegcos.asi

               extrn   _unSupported : near

;--------------------------------------------------------------------------------

_TEXT          segment

;--------------------------------------------------------------------------------

               public  _mediaCheck
_mediaCheck    proc near                   ; int mediaCheck(void);

               call _unSupported
               ret

_mediaCheck    endp

;--------------------------------------------------------------------------------

_TEXT          ends
```

```
;------------------------------------------------------------------------------

                end


; DDBUIBPB.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione buildBPB() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _buildBPB
_buildBPB       proc near                     ; int buildBPB(void);

                call _unSupported
                ret

_buildBPB       endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end


; DDINPIOC.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione inputIOCTL() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _inputIOCTL
_inputIOCTL     proc near                     ; int inputIOCTL(void);

                call _unSupported

_inputIOCTL     endp

;------------------------------------------------------------------------------
```

```
_TEXT           ends

;------------------------------------------------------------------------------

                end


; DDINPUT.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione input() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _input
_input          proc near                    ; int input(void);

                call _unSupported
                ret

_input          endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end


; DDINPND.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione inputND() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _inputND

_inputND        proc near                    ; int inputND(void);

                call _unSupported
                ret
```

```
_inputND        endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDINPSTA.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione inputStatus() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _inputStatus
_inputStatus    proc near                     ; int inputStatus(void);

                call _unSupported
                ret

_inputStatus    endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDINPFLU.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione inputFlush() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _inputFlush
_inputFlush     proc near                     ; int inputFlush(void);
```

```
                call _unSupported
                ret

_inputFlush     endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDOUTPUT.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione output() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _output
_output         proc near                     ; int output(void);

                call _unSupported
                ret

_output         endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDOUTVER.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione outputVerify() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------
```

```
                public  _outputVerify
_outputVerify   proc near                    ; int outputVerify(void);

                call _unSupported
                ret

_outputVerify   endp

;-------------------------------------------------------------------------------

_TEXT           ends

;-------------------------------------------------------------------------------

                end


; DDOUTSTA.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione outputStatus() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;-------------------------------------------------------------------------------

_TEXT           segment

;-------------------------------------------------------------------------------

                public  _outputStatus
_outputStatus   proc near                    ; int outputStatus(void);

                call _unSupported
                ret

_outputStatus   endp

;-------------------------------------------------------------------------------

_TEXT           ends

;-------------------------------------------------------------------------------

                end


; DDOUTFLU.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione outputFlush() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;-------------------------------------------------------------------------------
```

```
_TEXT           segment

;------------------------------------------------------------------------------

                public  _outputFlush
_outputFlush    proc near                       ; int outputFlush(void);

                call _unSupported
                ret

_outputFlush    endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDOUTIOC.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione outputIOCTL() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                public  _outputIOCTL
_outputIOCTL    proc near                       ; int outputIOCTL(void);

                call _unSupported
                ret

_outputIOCTL    endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDDEVOPE.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione deviceOpen() dummy usata solo se non implementata in C

include   ddsegcos.asi

                extrn   _unSupported : near
```

```
;------------------------------------------------------------------------------

_TEXT          segment

;------------------------------------------------------------------------------

               public  _deviceOpen
_deviceOpen    proc near                    ; int deviceOpen(void);

               call _unSupported
               ret

_deviceOpen    endp

;------------------------------------------------------------------------------

_TEXT          ends

;------------------------------------------------------------------------------

               end
```

```
; DDDEVCLO.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione deviceClose() dummy usata solo se non implementata in C

include   ddsegcos.asi

               extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT          segment

;------------------------------------------------------------------------------

               public  _deviceClose
_deviceClose   proc near                    ; int deviceClose(void);

               call _unSupported
               ret

_deviceClose   endp

;------------------------------------------------------------------------------

_TEXT          ends

;------------------------------------------------------------------------------

               end
```

```
; DDMEDREM.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione mediaRemove() dummy usata solo se non implementata in C
```

```
include   ddsegcos.asi

               extrn   _unSupported : near

;--------------------------------------------------------------------------------

_TEXT          segment

;--------------------------------------------------------------------------------

               public  _mediaRemove
_mediaRemove   proc near                    ; int MediaRemove(void);

               call _unSupported
               ret

_mediaRemove   endp

;--------------------------------------------------------------------------------

_TEXT          ends

;--------------------------------------------------------------------------------

               end
```

```
; DDOUTBUS.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione outputBusy() dummy usata solo se non implementata in C

include   ddsegcos.asi

               extrn   _unSupported : near

;--------------------------------------------------------------------------------

_TEXT          segment

;--------------------------------------------------------------------------------

               public  _outputBusy
_outputBusy    proc near                    ; int outputBusy(void);

               call _unSupported
               ret

_outputBusy    endp

;--------------------------------------------------------------------------------

_TEXT          ends

;--------------------------------------------------------------------------------

               end
```

```
; DDGENIOC.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
```

```
;
; funzione genericIOCTL() dummy usata solo se non implementata in C

include   ddsegcos.asi

               extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT          segment

;------------------------------------------------------------------------------

               public  _genericIOCTL
_genericIOCTL  proc near                    ; int genericIOCTL(void);

               call _unSupported
               ret

_genericIOCTL  endp

;------------------------------------------------------------------------------

_TEXT          ends

;------------------------------------------------------------------------------

               end
```

```
; DDGETLOG.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione getLogicalDev() dummy usata solo se non implementata in C

include   ddsegcos.asi

               extrn   _unSupported : near

;------------------------------------------------------------------------------

_TEXT          segment

;------------------------------------------------------------------------------

               public  _getLogicalDev
_getLogicalDev proc near                    ; int getLogicalDev(void);

               call _unSupported
               ret

_getLogicalDev endp

;------------------------------------------------------------------------------

_TEXT          ends

;------------------------------------------------------------------------------

               end
```

```
; DDSETLOG.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione setLogicalDev() dummy usata solo se non implementata in C

include   ddsegcos.asi

               extrn   _unSupported : near

;-------------------------------------------------------------------------------

_TEXT          segment

;-------------------------------------------------------------------------------

               public  _setLogicalDev
_setLogicalDev proc near                    ; int setLogicalDev(void);

               call _unSupported
               ret

_setLogicalDev endp

;-------------------------------------------------------------------------------

_TEXT          ends

;-------------------------------------------------------------------------------

               end
```

Le funzioni sin qui presentate hanno il solo scopo di evitare al programmatore l'obbligo di definire comunque una funzione dedicata per i servizi non supportati: infatti, se nel sorgente C non esiste una funzione con lo specifico nome previsto per ogni servizio, il linker include nel file binario la corrispondente funzione di libreria. Ad esempio, se il driver supporta unicamente i servizi 4, 5, 6 e 9 (vedere pag. 362), nel sorgente C devono essere definite, oltre alla `init()`, anche una `input()`, una `inputND()`, una `inputStatus()` e una `output()`, rispettivamente: i loro nomi non possono essere modificati. Per tutti gli altri servizi viene automaticamente importata nel file binario la corrispondente funzione di libreria, la quale segnala al DOS che il servizio non è supportato dal driver.

La funzione che segue è inserita in libreria dopo quelle di gestione dei servizi non supportati a soli fini di comodità: il suo indirizzo, infatti, rappresenta l'indirizzo al quale terminano in memoria le funzioni di servizio (quelle definite nel sorgente C precedono sempre, nel file binario, le funzioni di libreria). Essa non esegue alcuna azione e restituisce immediatamente il controllo alla funzione chiamante.

```
; DDENDOFS.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; funzione endOfServices() dummy usata solo per calcolare l'indirizzo al
; quale termina l'ultima delle f() dummy di servizio. non e' mai eseguita

include   ddsegcos.asi

;-------------------------------------------------------------------------------

_TEXT          segment

;-------------------------------------------------------------------------------
```

```
                public  _endOfServices
_endOfServices proc near

                ret

_endOfServices endp

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

I cinque listati che seguono rendono disponibili funzionalità normalmente incluse nello startup code originale fornito con il compilatore. Si tratta di variabili e funzioni che i normali programmi utilizzano in modo automatico: a scopo di maggiore effiecienza esse sono invece inserite in libreria e il programmatore deve farne esplicito uso se necessario[362]. Vale la pena di sottolineare che nei file `DDRESVEC.ASM` e `DDSAVVEC.ASM` sono definite `SaveVectors()` e `_restorezero()`, sulla quale ci si sofferma a pag. 327. Inoltre, `DDDUMMY.ASM` contiene il codice necessario a simulare alcune funzioni di uscita da programmi C (`exit()`, `abort()`, etc.), private però di qualunque effetto, in quanto i device driver non terminano mai nel modo consueto ai normali programmi.

```
; DD_EXPTR.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; procedure dummy che i normali programmi usano per uscire a DOS o
; resettare vettori ed effettuare il flush degli streams: dati
;

include  DDSEGCOS.ASI

                extrn _dummy : near        ; dummy() e' definita in DDDUMMY.ASM

;------------------------------------------------------------------------------

_DATA           segment

;------------------------------------------------------------------------------

                ; puntatori a funzioni di cleanup per streams e file

                ;------------------------------------------------------------

                public __exitbuf
__exitbuf       dw     offset _TEXT:_dummy

                public __exitfopen
__exitfopen     dw     offset _TEXT:_dummy

                public __exitopen
__exitopen      dw     offset _TEXT:_dummy

;------------------------------------------------------------------------------
```

[362] Alcune sono utilizzate da funzioni di libreria C: ne segue che sono consolidate al file binario solo se quelle funzioni sono invocate nel sorgente C.

```
_DATA           ends

;------------------------------------------------------------------------------

                end
```

```
; DD_VECT.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;

include    DDSEGCOS.ASI

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

_Int0Vector     dd      0                      ; area locale memorizzazione
_Int4Vector     dd      0                      ; vettori per SaveVectors() e
_Int5Vector     dd      0                      ; _restorezero()
_Int6Vector     dd      0                      ;

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDDUMMY.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; procedure dummy che i normali programmi usano per uscire a DOS o
; resettare vettori ed effettuare il flush degli streams
;

include  DDSEGCOS.ASI

;------------------------------------------------------------------------------

_TEXT           segment

;------------------------------------------------------------------------------

                ; labels per nomi funzioni di uscita e cleanup: puntano
                ; tutte a __dummy__proc, che segue solo una RET

                ;--------------------------------------------------------------

                public  dummy                ; static dummy
dummy           label

                ;--------------------------------------------------------------

                public  _abort               ; abort()
_abort          label
```

```
                ;------------------------------------------------------------

                public  _atexit             ; atexit()
_atexit         label

                ;------------------------------------------------------------

                public  _dummy              ; static dummy
_dummy          label

                ;------------------------------------------------------------

                public  _exit               ; exit()
_exit           label

                ;------------------------------------------------------------

                public  _keep               ; keep()
_keep           label

                ;------------------------------------------------------------

                public  __cexit             ; _cexit()
__cexit         label

                ;------------------------------------------------------------

                public  __checknull         ; asm
__checknull     label

                ;------------------------------------------------------------

                public  __cleanup           ; asm
__cleanup       label

                ;------------------------------------------------------------

                public  __c_exit            ; _c_exit()
__c_exit        label

                ;------------------------------------------------------------

                public  __dos_keep          ; _dos_keep()
__dos_keep      label

                ;------------------------------------------------------------

                public  __exit              ; _exit()
__exit          label

                ;------------------------------------------------------------

                public  __terminate         ; asm
__terminate     label

                ;------------------------------------------------------------

                public  ___EXIT             ; pascal ___exit()
___EXIT         label

                ;------------------------------------------------------------

                public  ___exit             ; pascal ___exit()
___exit         label
```

```
;------------------------------------------------------------------------------

__dummy__proc   PROC    near                 ; funzione dummy: esegue solo RET

                ret

__dummy__proc   ENDP

;------------------------------------------------------------------------------

_TEXT           ends

;------------------------------------------------------------------------------

                end
```

```
; DDRESVEC.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; procedure per la gestione dei vettori int 0, 4, 5, 6 e del gestore
; dell'int 0 (divide by zero). Nei normali programmi al ritorno a DOS
; e' chiamata _restorezero(), che ripristina detti vettori, salvati da
; SaveVectors() e modificati da alcune funzioni di libreria C. Il device
; driver deve invocare esplicitamente SaveVectors() e _restorezero()
; se necessario.
;

include    DDSEGCOS.ASI

                extrn _Int0Vector : dword             ; definiti in DD_VECT.ASM
                extrn _Int4Vector : dword
                extrn _Int5Vector : dword
                extrn _Int6Vector : dword

;------------------------------------------------------------------------------

_TEXT         segment

;------------------------------------------------------------------------------

              public  __restorezero
__restorezero proc near                               ; void _restorezero(void)

              push ds
              mov ax,2500h
              lds dx,_Int0Vector
              int 21h
              pop ds
              push ds
              mov ax,2504h
              lds dx,_Int4Vector
              int 21h
              pop ds
              push ds
              mov ax,2505h
              lds dx,_Int5Vector
              int 21h
              pop ds
              push ds
              mov ax,2506h
              lds dx,_Int6Vector
```

```
              int 21h
              pop ds
              ret

__restorezero endp

;--------------------------------------------------------------------------------

_TEXT         ENDS

;--------------------------------------------------------------------------------

              END


; DDSAVVEC.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; procedure per la gestione dei vettori int 0, 4, 5, 6 e del gestore
; dell'int 0 (divide by zero). Nei normali programmi SaveVectors() e'
; chiamata dallo startup code e installa ZeroDivision. Al ritorno a DOS
; e' chiamata _restorezero(), che ripristina i vettori. La ErrorDisplay
; scrive su stderr un mesaggio di errore. Il device driver deve invocare
; esplicitamente SaveVectors() e _restorezero() se necessario.
;

include    DDSEGCOS.ASI

                extrn _Int0Vector : dword             ; definiti in DD_VECT.ASM
                extrn _Int4Vector : dword
                extrn _Int5Vector : dword
                extrn _Int6Vector : dword

;--------------------------------------------------------------------------------

_TEXT        segment

;--------------------------------------------------------------------------------

ZDivMsg      db      'Divide error', 13, 10 ; messaggio di errore int 0
ZDivMsgLen   equ     $ - ZDivMsg


ErrorDisplay proc near                        ; void pascal ErrorDisplay(void)

             mov ah,040h
             mov bx,2
             int 021h
             ret

ErrorDisplay endp

             ;---------------------------------------------------------------

ZeroDivision proc far                  ; void far pascal ZeroDivision(void)

             mov cx,ZDivMsgLen
             mov dx,offset DGROUP:ZDivMsg
             push cs                            ; modello tiny: DS = CS
             pop  ds
             call ErrorDisplay
             mov  ax, 3
             push ax
```

```
             ret

ZeroDivision endp

             ;--------------------------------------------------------------

             public  SaveVectors
SaveVectors  proc near                      ; void pascal SaveVectors(void)

             push ds
             mov ax, 3500h
             int 021h
             mov word ptr _Int0Vector,bx
             mov word ptr _Int0Vector+2,es
             mov ax,3504h
             int 021h
             mov word ptr _Int4Vector,bx
             mov word ptr _Int4Vector+2,es
             mov ax,3505h
             int 021h
             mov word ptr _Int5Vector,bx
             mov word ptr _Int5Vector+2,es
             mov ax,3506h
             int 021h
             mov word ptr _Int6Vector,bx
             mov word ptr _Int6Vector+2,es
             mov ax,2500h
             mov dx,cs
             mov ds,dx
             mov dx,offset ZeroDivision
             int 21h
             pop ds
             ret

SaveVectors  endp

;---------------------------------------------------------------------------

_TEXT        ENDS

;---------------------------------------------------------------------------

             END
```

Il listato seguente è relativo alla funzione `setStack()`, che ha un ruolo di estrema importanza nel toolkit: essa, infatti, consente di rilocare lo stack originale del driver durante l'inizializzazione. Il device driver, per sicurezza, non deve utilizzare lo stack del DOS per effettuare le proprie operazioni; allo scopo, nello startup module, è definita la variabile `DrvStk`, la quale è semplicemente un array (cioè una sequenza) di byte. La `Interrupt()`, in ingresso, salva l'indirizzo attualmente attivo nello stack DOS (`SS:SP`) e carica in `SS:SP` l'indirizzo del primo byte successivo a `DrvStk`, individuato dalla label `DrvStkEnd` (lo stack è sempre usato a ritroso, e viene "riempito" dall'ultima word alla prima). La dimensione di default dello stack è pari a `STKSIZE` byte[363] e potrebbe rivelarsi insufficiente; d'altra parte, incrementare il valore di `STKSIZE` non rappresenterebbe una valida soluzione per tutti i device driver realizzati con il toolkit, in quanto, oltre a non garantire con certezza assoluta un'adeguata capienza di stack in alcuni casi, potrebbe, in altri, determinare uno spreco di memoria pari a tutta la parte di stack non utilizzata.

[363] La costante manifesta `STKSIZE` è definita nel file `DDSEGCOS.ASI` (pag. 386).

La `setStack()` permette al programmatore di creare un nuovo stack, dimensionato in modo ottimale secondo le presumibili esigenze del singolo driver e di forzare la `Interrupt()` a servirsi di questo, in luogo di quello originale (che può essere riutilizzato a runtime per ogni necessità, come un qualsiasi array). E' sufficiente invocare `setStack()` passandole come parametri l'indirizzo `near` (di tipo `(void *)`) del nuovo stack e un `unsigned int` che ne esprime la dimensione in byte; essa restituisce un intero senza segno pari al numero di byte effettivamente disponibili nel nuovo stack[364]. In caso di fallimento, `setStack()` restituisce `0`.

Una funzione jolly (pag. 170) è un mezzo semplice per implementare un nuovo stack: il nome della funzione può essere passato a `setStack()` come indirizzo near[365] (primo parametro).

```
#pragma  option -k-
#include <bzdd.h>                  // include file per la libreria toolkit (pag. 425)

....

void newDrvStack(void)
{
    asm db 1024;                                 // nuovo stack: 1024 bytes, 512 words
}

....

int init(int argc,char **argv)                          // inizializzazione del driver
{                                                      // init() e' descritta a pag. 441
    if(!setStack(newDrvStk,1024)) {
        discardDriver();                                              // vedere pag. 423
        return(errorReturn(E_GENERAL));
    }
    ....
}
```

Nulla di particolarmente complesso, come si può facilmente constatare, quando si osservino scrupolosamente alcuni accorgimenti: in primo luogo, `setStack()` deve essere chiamata da `init()` (vedere pag. 441). Questa, inoltre, non deve dichiarare variabili automatiche (ma può dichiarare variabili `static` e `register`, purché, si abbia una ragionevole certezza che queste ultime siano effettivamente gestite nei registri della CPU e non allocate nello stack[366]). Possono essere utilizzate variabili globali, eventualmente redichiarate come `extern`. In pratica, la rilocazione dello stack, se necessaria, deve essere la prima operazione effettuata dal driver; i suddetti limiti, però, non rappresentano un reale problema: è sufficiente che `init()` deleghi ad un'altra funzione tutte le successive operazioni di inizializzazione per eliminare ogni rischio.

---

[364] Nel nuovo stack può essere disponibile meno spazio di quanto richiesto: dal momento che ogni stack deve iniziare ad un indirizzo pari e deve essere composto da un numero pari di byte, `setStack()` effettua gli aggiustamenti eventualmente necessari. Ad esempio, se la dimensione specificata è pari, ma l'indirizzo base del nuovo stack è dispari, `setStack()` rende disponibile un byte in meno di quanto richiesto e modifica opportunamente l'indirizzo ricevuto come parametro. La differenza tra dimensione richiesta ed effettiva può essere al massimo pari a 2 byte.

[365] Il modello di memoria di riferimento è il tiny model, pertanto tutti gli indirizzi sono, per default, `near`; inoltre `CS` = `DS` = `SS`.

[366] In realtà, la dichiarazione di variabili automatiche, allocate nello stack, non è di per sé un problema, ma potrebbe esserlo il loro contenuto. Se, ad esempio, una di esse è un puntatore inizializzato, prima della chiamata a `setStack()`, con un indirizzo relativo allo stack stesso, appare evidente che "spostando" questo, detto indirizzo dovrebbe essere aggiornato di conseguenza; `setStack()`, tuttavia, conosce lo spazio occupato dai dati che è chiamata a rilocare, ma non il loro significato.

```
....

int install(int argc,char **argv)
{
    int a, b, c;
    long val;

    ....                               // qui possiamo fare tutto quello che ci pare!
    setResCodeEnd(_endOfDrvr);
    return(E_OK);                                        // vedere BZDD.H (pag. 425)
}

....

int init(int argc,char **argv)
{
    printf("Installazione di %s\n",argv[0]);        // visualizza il nome del driver
    if(!setStack(newDrvStk,1024)) {
        discardDriver();
        return(errorReturn(E_GENERAL));
    }
    return(install(argc,argv));
}
```

E' molto importante ricordare che la rilocazione dello stack è permanente: ciò significa che `setStack()` deve essere invocata una sola volta e che tutte le operazioni effettuate dal driver dopo la chiamata a `setStack()`, sia in fase di inizializzazione che durante la normale operatività del computer, utilizzano il nuovo stack; d'altra parte, non è possibile riattivare lo stack originale: questo rimane disponibile come un comune array, il cui indirizzo `near` è dato dal puntatore `_freArea` e la cui dimensione in byte è pari a `_freAreaDim` (vedere `BZDD.H`, a pag. 425 e seguenti). Le variabili `void *_freArea` e `unsigned _freAreaDim` valgono `NULL` e, rispettivamente, `0` se lo stack non è stato rilocato.

Quanto stack serve al driver? Nell'implementazione qui descritta, la costante manifesta `STKSIZE` vale 512 byte: tale valore, per esigenze intrinseche alla libreria C, non può essere diminuito; tuttavia esso è appena sufficiente per aprire pochi file via stream (vedere pag. 116) e per allocare (con `malloc()`, etc.: pag. 109) poche decine di byte. La rilocazione dello stack è, pertanto, un'operazione quasi obbligatoria per molti device driver: 2 o 4 Kb sono, di norma, sufficienti per la maggior parte delle esigenze, ma non vi sono problemi ad utilizzare uno stack di dimensioni superiori, a parte il maggior "consumo" di memoria.

```
; DDSETSTK.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
; unsigned setStack(unsigned base,unsigned len);
;
; genera un nuovo stack locale per il device driver
;
; unsigned base;    offset del (puntatore near al) nuovo stack
; unsigned len;     lunghezza dello stack in bytes
;
; restituisce:          la lunghezza effettiva del nuovo stack; puo' essere
;                       inferiore a quella richiesta se quest'ultima e'
;                       dispari o se e' dispri l'offset dello stack (viene
;                       effettuato allineamento alla word); se il valore
;                       restituito e' 0 uno dei parametri e' errato e lo
;                       stack non e' stato generato
;
; note:                 se utilizzata, e' opportuno che sia la prima f()
;                       chiamata da init(), inoltre non e' disponibile una
```

```
;                       funzione per ritornare allo stack precedente ed
;                       eliminare quello generato da setStack(); se il
;                       nuovo stack viene effettivamente attivato, allora
;                       l'area statica occupata dallo stack originale del
;                       driver e' riutilizzabile secondo le necessita'
;                       dell'utente: il suo offset (puntatore near) e la
;                       sua lunghezza in bytes sono disponibili in _freArea
;                       e _freAreaDim rispettivamente.
;

INCLUDE   DDSEGCOS.ASI

; dichiarazione simboli esterni

             extrn  __savSP1     : word
             extrn  __savSP2     : word
             extrn  __newTOS     : word
             extrn  _DrvStk      : byte
             extrn  _DrvStkEnd   : byte
             extrn  ___heapbase  : word
             extrn  ___brklvl    : word
             extrn  __freArea    : word
             extrn  __freAreaDim : word

             extrn  __stklen     : word     ; da libreria C

             extrn  __setupio    : near     ; e' una funzione di libreria C
                                            ; chiamata dallo startup code dei
                                            ; normali programmi: prepara le
                                            ; strutture statiche di tipo FILE
                                            ; per gli streams

;--------------------------------------------------------------------------------

_TEXT        segment

;--------------------------------------------------------------------------------

             public _setStack
_setStack    proc near        ; unsigned setStack(unsigned base,unsigned len);

             mov __savSP2,sp             ; salva SP in ingresso: deve essere la
                                         ; prima istruzione di setStack()
             push bp                     ; genera std stack frame per accedere
             mov bp,sp                   ; ai parametri passati da init()
             mov dx,word ptr [bp+4]      ; DX = offset di inizio del nuovo stack
             mov ax,word ptr [bp+6]      ; AX = lunghezza
             pop bp                      ; elimina stack frame per operaz. copia
             test dx,1                   ; offset base del nuovo stack e' pari?
             jz W_ALIGNED_BASE           ; si; salta
             inc dx                      ; allinea base stack e heap a word
             dec ax                      ; no: lo spazio e' diminuito di 1 byte!
W_ALIGNED_BASE:
             and ax,0FFFEh               ; impone lunghezza stack pari
             mov cx,offset _DrvStkEnd    ; CX = attuale TopOfStack
             sub cx,__savSP2             ; CX = TOS - SP = bytes da copiare
             cmp ax,cx                   ; lungh. (AX) > bytes (CX) ?
             jbe ERROR                   ; no: stack insufficiente; salta
             mov bx,dx                   ; si: BX = offset base nuovo stack
             add dx,ax                   ; DX = off base + lunghezza = nuovo TOS
             jnc PARMS_OK                ; CARRY = 0: TOS <= FFFFh: OK; salta
ERROR:
             xor ax,ax                   ; segnala errore
             jmp EXIT_FUNC
```

```
PARMS_OK:
            mov __newTOS,dx          ; salva nuovo TOS
            mov ___heapbase,bx       ; base heap = base nuovo stack
            mov ___brklvl,bx         ; attuale fine heap = base nuovo stack

            mov __freArea,offset _DrvStk  ; offset ex-stack per riutilizzo
            mov __freAreaDim,STKSIZE      ; dimensione vecchio stack

            mov bx,__savSP1          ; SP prima di push parametri di init()
            sub bx,__savSP2          ; differenza tra i due SP
            cmp bx,NPA_SP_DIFF       ; diff se init() non ha parms e var. auto
            je BP_ADJUST             ; se = basta settare BP e copiare stack
STACK_ADJUST:                        ; altrimenti prima di copiare:
            mov word ptr [bp],dx     ; BP = offset da SS della copia di BP
                                     ; PUSHed in init(), cioe' SS:[BP]
                                     ; = precedente BP (TOS), ora = __newTOS

            mov bx,offset _DrvStkEnd ; BP e' offset rispetto a SS, ora BX =
            sub bx,bp                ; distanza tra quell'offset e fine stack
            mov bp,dx                ; BP viene trasformato per puntare allo
            sub bp,bx                ; stesso offset rispetto a __newTOS (AX)
            jmp COPY_STACK           ; BP gia' valorizzato
BP_ADJUST:
            mov bp,dx                ; BP = __newTOS: init(void) e no var auto
COPY_STACK:
            push es                  ; salva ES (DS e' sempre = CS = SS), SI
            push si                  ; e DI. Non copiati perche' estratti da
            push di                  ; stack prima dell'attivazione nuovo stk
            std                      ; copia dall'alto al basso
            mov si,offset _DrvStkEnd ; fine vecchio stack
            sub si,2                 ; DS:SI -> prima word da copiare (BP DOS)
            push ds
            pop es                   ; ES = DS
            mov di,dx                ; DI = __newTOS
            sub di,2                 ; ES:DI -> ultima word del nuovo stack
            mov bx,cx                ; CX contiene ancora numero bytes stack
            cli
            rep movsb                ; copia il contenuto dello stack
            sti
            pop di                   ; ripristina i registri salvati
            pop si                   ; estraendoli ancora dal vecchio stack
            pop es                   ; ES era la prima word non copiata
            mov sp,dx                ; SP = __newTOS
            sub sp,bx                ; __newTOS - bytes_in_stack = nuovo SP

            mov __stklen,ax          ; per libreria C
            push ax                  ; usa gia' il nuovo stack
            call __setupio           ; ora ha senso: c'e' spazio (da startup)
            pop ax                   ; restit. AX = LEN: nuovo stack attivato!
EXIT_FUNC:
            ret

_setStack   endp

;-------------------------------------------------------------------------------

_TEXT       ends

;-------------------------------------------------------------------------------

            end
```

Un altro tassello della libreria toolkit è rappresentato dalla funzione `setupcmd()`, che analizza la command line del driver e inizializza una variabile ed un array che possono essere utilizzati da `init()` in modo del tutto analogo a quello comunemente seguito nella `main()` dei normali programmi per `argc` e `argv` (vedere pag. 105).

La `setupcmd()` non accetta parametri e non restituisce alcunché; è progettata come procedura di servizio per lo startup module e da questo viene automaticamente invocata: essa accede alla command line presente in `CONFIG.SYS` tramite il puntatore passato dal DOS nel request header del servizio 0 (vedere pag. 363) e ne effettua una copia locale, sulla quale opera la scansione, sostituendo con un `NULL` lo spazio immediatamente successivo ad ogni parametro[367]. Al termine della scansione la copia della command line risulta trasformata in una sequenza di stringhe: i loro indirizzi sono memorizzati nell'array `char **_cmdArgs` ed il loro numero nella variabile `int _cmdArgsN`; lo startup module (`driverInit()`, vedere pag. 399), prima di invocare `init()`, copia nello stack l'indirizzo del primo e il valore contenuto nella seconda, predisponendo così i due parametri che la stessa `init()` può utilizzare, se dichiarati (vedere pag. 441).

Va ancora precisato che, qualora il device driver non abbia necessità di accedere alla command line, è possibile definire nel sorgente C una

```
void setupcmd(void)
{
}
```

per evitare che il linker importi nel file binario la versione della funzione presente in libreria, col vantaggio di ottenere un driver di dimensioni minori.

```
; DDSETCMD.ASM - Barninga Z! - 1994
;
; ASSEMBLER SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY
;
;
; void setupcmd(void);
;
; funzione di parsing della command line
;
; genera una copia locale statica della command line e in questa tronca
; opportunamente le stringhe con NULLs, valorizzando un array statico di
; puntatori a char (_cmdArgs) e un intero (_cmdArgsN) che contiene il
; numero degli items presenti sulla cmd line (compreso il nome del driver)
;

include  DDSEGCOS.ASI

; dichiarazioni dei simboli esterni

             extrn _RHptr     : dword
             extrn __cmdLine  : word
             extrn __cmdArgs  : word
             extrn __cmdArgsN : word

;------------------------------------------------------------------------------

_TEXT        segment
```

[367] L'indirizzo passato dal DOS nel request header è quello del byte che segue immediatamente la stringa `DEVICE=` in `CONFIG.SYS`. La stringa che si trova a quell'indirizzo non deve essere modificata: di qui la necessità di crearne una copia locale ad uso esclusivo del device driver. Si tenga inoltre presente che la stringa non termina con un `NULL`, ma con un `CR` o un `LF` o un `EOF` (ASCII 13, 10, 26 rispettivamente).

```
;------------------------------------------------------------------------------

            public _setupcmd
_setupcmd   proc near               ; void setupcmd(void) prepara due parametri
                                    ; per init() analoghi a argc e argv
            push si
            push di
            push ds
            lds si,DGROUP:_RHptr         ; DS:SI punta a Request Header
            lds si,[si+18]               ; DS:SI punta a Command Line
            mov di,offset __cmdLine      ; ES:DI punta a _cmdLine (ES = SS = CS)
            mov cx,CMDSIZE
            rep movsb                    ; crea copia locale della command line
            pop ds
            mov si,offset __cmdLine      ; DS:SI -> _cmdLine (ES = DS = SS = CS)
            inc si                       ; punta al secondo byte (emula LODSB)
            mov di,offset __cmdArgs      ; ES:DI -> _cmdArgs (ES = DS = SS = CS)
            xor dx,dx                    ; contatore argomenti
            cld                          ; operazioni stringa: forward
            call _setarg                 ;  funz. di servizio: vedere a fine listato
NEXTARG:
NEXTCHAR:                                ;;;; scansione di un parametro
            lodsb
            cmp al,32
            je ENDARG                    ; blank \
            cmp al,9                     ;        > fine argomento
            je ENDARG                    ; tab   /
            cmp al,13
            je ENDCMD                    ; CR    \
            cmp al,10                    ;        |
            je ENDCMD                    ; LF      > fine command line
            cmp al,26                    ;        |
            je ENDCMD                    ; EOF   /
            cmp al,34                    ;
            je DQUOTES                   ; DOUBLE QUOTES
            jmp NEXTCHAR
ENDARG:                                  ;;;; fine parametro
            mov byte ptr [si-1],0        ; termina str. con NULL (SI gia' incr.)
ENDARG_1:                                ;;;; scansione spazio tra due parametri
            lodsb
            cmp al,32
            je ENDARG_1                  ; blank \
            cmp al,9                     ;        > separatori argomenti
            je ENDARG_1                  ; tab   /
            cmp al,13
            je ENDCMD_1                  ; CR    \
            cmp al,10                    ;        |
            je ENDCMD_1                  ; LF      > fine command line
            cmp al,26                    ;        |
            je ENDCMD_1                  ; EOF   /
            cmp al,34                    ;
            je DQUOTES                   ; DOUBLE QUOTES: inizio parametro
            call _setarg
            jmp NEXTARG
DQUOTES:                                 ;;;; virgolette nella command line
            mov byte ptr [si-1],0
            inc si                       ; le virgolette sono scartate:
            call _setarg                 ; in _setarg AX=SI e DEC AX
            dec si                       ; ripristina il puntatore
DQUOTES_1:
            lodsb
            cmp al,13
            je ENDCMD                    ; CR    \
            cmp al,10                    ;        |
```

```
            je ENDCMD                 ; LF      > fine command line
            cmp al,26                 ;         |
            je ENDCMD                 ; EOF    /
            cmp al,34                 ;
            je ENDARG                 ; DQUOTES: fine parametro
            jmp DQUOTES_1
ENDCMD:                               ;;;; fine parametro e command line
            mov byte ptr [si-1],0     ; termina str. con NULL (SI gia' incr.)
ENDCMD_1:                             ;;;; fine command line
            xor ax,ax
            stosw                     ; _cmdArgs[DX] = NULL
            mov __cmdArgsN,dx         ; _cmdArgsN = numero argomenti
            pop di
            pop si

            ret

_setupcmd   endp

            ;------------------------------------------------------------------

_setarg     proc near        ;;;; inizio di un nuovo parametro
                             ;;;; routine di servizio per setupcmd()
            mov ax,si
            dec ax                    ; SI e' gia' stato incrementato
            stosw                     ; _cmdArgs[DX] -> argomento
            inc dx                    ; trovato un altro argomento

            ret

_setarg     endp

;-------------------------------------------------------------------------------

_TEXT       ends

;-------------------------------------------------------------------------------

            end
```

Ancora un sorgente, questa volta in C. La funzione `discardDriver()` comunica al DOS che il device driver non deve rimanere residente in memoria. Le operazioni effettuate seguono le indicazioni di Microsoft per la gestione del servizio 0 (vedere pag. 363). Essa può essere invocata quando, fallite le operazioni di inizializzazione, si renda necessario evitare l'installazione in memoria del driver. Per un esempio di utilizzo, vedere pag. 446.

```
/******************************************

    DDDISCRD.C - Barninga Z! - 1994

    C SOURCE FILE PER DEVICE DRIVER TOOLKIT - LIBRARY

    discardDriver() - comunica al DOS di NON lasciare residente
                      il device driver, secondo la procedura
                      definita nelle specifiche Microsoft.

    Sintassi:

    void discardDriver(void);

    Compilare con
```

```
    bcc -mt -c dddiscrd.c

*******************************************/
#include "bzdd.h"

void discardDriver(void)
{
    extern RequestHeaderFP RHptr;                        // accesso al request header
    extern DevDrvHeader    DrvHdr;                 // accesso ad device driver header

    DrvHdr.attrib &= 0x7FFF;                      // trasforma in block device driver
    RHptr->cp.initReq.nUnits = 0;                        // nessuna unita' supportata
    setResCodeEnd(NOT_RESIDENT);          // primo byte libero = indirizzo del driver
}
```

La nostra libreria è (finalmente!) completa. Non resta che assemblare tutti i sorgenti e generare il file `.LIB`; la prima operazione è banale:

```
tasm -ml *.asm
```

L'opzione `-ml` richiede all'assemblatore di distinguere i caratteri maiuscoli da quelli minuscoli; l'unica precauzione da prendere consiste nell'evitare di riassemblare, se non necessario, il file `DDHEADER.ASM`, contenente lo startup module (dunque è meglio rinominarlo o spostarlo temporaneamente in un'altra directory).

Non va poi dimenticato il sorgente C di `discardDriver()`, per il quale bisogna effettuare la compilazione senza linking (opzione `-c`) per il modello di memoria tiny (`-mt`):

```
bcc -c -mt dddiscrd.c
```

La libreria può essere costruita utilizzando la utility `TLIB`: visto il numero di file coinvolti nell'operazione, può risultare comodo predisporre un *response file* analogo a quello presentato di seguito.

```
+-ddinit   &
+-ddmedche &
+-ddbuibpb &
+-ddinpioc &
+-ddinput  &
+-ddinpnd  &
+-ddinpsta &
+-ddinpflu &
+-ddoutput &
+-ddoutver &
+-ddoutsta &
+-ddoutflu &
+-ddoutioc &
+-dddevope &
+-dddevclo &
+-ddmedrem &
+-ddoutbus &
+-ddgenioc &
+-ddgetlog &
+-ddsetlog &
+-ddendofs &
+-dd_exptr &
+-dd_vect  &
+-dddummy  &
+-ddresvec &
+-ddsavvec &
+-ddsetstk &
```

```
+-ddsetcmd &
+-dddiscrd
```

Il response file (in questo esempio `BZDD.LST`) elenca tutti i file `.OBJ` da inserire in libreria[368]; la presenza di entrambi i simboli `+` e `–` davanti ad ogni nome forza `TLIB` a sostituire nella libreria il corrispondente modulo `.OBJ`, se già esistente (il carattere "`&`" indica che l'elenco prosegue sulla riga successiva). Pertanto, il comando

```
tlib /C bzdd @bzdd.lst
```

produce il file `BZDD.LIB`, contenente tutte le funzioni sin qui presentate. L'opzione `/C` impone a `TLIB` di distinguere i caratteri maiuscoli da quelli minuscoli nei nomi dei simboli referenziati e definiti all'interno di ogni singolo object file (vedere anche pag. 149).

Per utilizzare produttivamente la libreria è ancora necessario creare un file `.H` (include file) contenente, al minimo, i prototipi delle funzioni, le dichiarazioni delle costanti e le definizioni di alcune macro. Il file `BZDD.H` è listato di seguito.

```
// BZDD.H - Barninga Z! - 1994
//
// include file per libreria device driver
//

#ifndef __BZDD_H                     // evita doppia inclusione
#define __BZDD_H

// necessario includere DOS.H se non ancora incluso

#ifndef __DOS_H
#include <dos.h>
#endif

// COSTANTI MANIFESTE

    // codici di ritorno e di stato

#define  S_SUCCESS         0x0000
#define  S_ERROR           0x8000     // codici di status
#define  S_BUSY            0x0200     // in OR tra di loro
#define  S_DONE            0x0100

#define  E_OK              0          // codici di ritorno in OR coi precedenti
#define  E_WR_PROTECT      0
#define  E_UNKNOWN_UNIT    1
#define  E_NOT_READY       2
#define  E_UNKNOWN_CMD     3
#define  E_CRC             4
#define  E_LENGTH          5
#define  E_SEEK            6
#define  E_UNKNOWN_MEDIA   7
#define  E_SEC_NOTFOUND    8
#define  E_OUT_OF_PAPER    9
#define  E_WRITE           10
#define  E_READ            11
#define  E_GENERAL         12
#define  E_RESERVED_1      13
#define  E_RESERVED_2      14
#define  E_INVALID_DSKCHG  15
```

[368] Il nome del response file deve essere passato a `TLIB` preceduto dal carattere `@`.

```
    // servizi del driver

#define  C_INIT            0
#define  C_MEDIACHECK      1
#define  C_BUILDBPB        2
#define  C_INPUTIOCTL      3
#define  C_INPUT           4
#define  C_INPUTND         5
#define  C_INPUTSTATUS     6
#define  C_INPUTFLUSH      7
#define  C_OUTPUT          8
#define  C_OUTPUTVERIFY    9
#define  C_OUTPUTSTATUS    10
#define  C_OUTPUTFLUSH     11
#define  C_OUTPUTIOCTL     12
#define  C_DEVICEOPEN      13
#define  C_DEVICECLOSE     14
#define  C_MEDIAREMOVE     15
#define  C_OUTPUTBUSY      16
#define  C_GENERICIOCTL    19
#define  C_GETLOGICALDEV   23
#define  C_SETLOGICALDEV   24

    // altre

#define  NO_VLABEL         "NO NAME"     // per dischi senza volume label
#define  RB_CHG            0xFF          // dischetto sostituito
#define  RB_MAYBE          0             // dischetto sostituito... forse
#define  RB_NOTCHG         1             // dischetto non sostituito

#define  NOT_RESIDENT      0             // da passare a setResCodeEnd() se il
                                         // driver non deve rimanere residente.
                                         // setResCodeEnd() e' una macro definita
                                         // in questo sorgente a pag. 431.

// FUNZIONI CORRISPONDENTI AI SERVIZI DEI DRIVER
// servizi del driver, da implementare in C. Le implementazioni assembler in
// libreria servono solo come placeholders per quelle non realizzate in C e
// non fanno che chiamare unSupported()

#ifdef __cplusplus                          // per poter usare il toolkit con il C++
extern "C" {
#endif

int mediaCheck(void);
int buildBPB(void);
int inputIOCTL(void);
int input(void);
int inputND(void);
int inputStatus(void);
int inputFlush(void);
int output(void);
int outputVerify(void);
int outputStatus(void);
int outputFlush(void);
int outputIOCTL(void);
int deviceOpen(void);
int deviceClose(void);
int mediaRemove(void);
int outputBusy(void);
int genericIOCTL(void);
int getLogicalDev(void);
int setLogicalDev(void);
```

```
// ALTRE FUNZIONI

void discardDriver(void);     // comunica al DOS di non lasciare residente il driver
int errorReturn(int errNum);                      // restit. al DOS il codice errNum
unsigned setStack(unsigned base,unsigned len);              // genera un nuovo stack
int unSupported(void);                           // rest. al DOS l'err E_UNKNOWN_CMD

// FUNZIONI DA STARTUP CODE

void pascal SaveVectors(void);         // salva vett. 0 4 5 6 installa handler int 0
void _restorezero(void);                              // ripristina vettori 0 4 5 6

// VARIABILI GLOBALI (derivate da startup code o definite liberamente)

extern int errno;                // codice di errore
extern unsigned _version;        // versione e revisione DOS
extern unsigned _osversion;      // versione e revisione DOS
extern unsigned char _osmajor;   // versione DOS
extern unsigned char _osminor;   // revisione DOS
extern unsigned long _StartTime; // timer clock ticks al caricamento
extern unsigned _systemMem;      // Kb di memoria convenzionale installati
extern unsigned _psp;            // segmento di caricamento (CS), non vero PSP
extern unsigned _baseseg;        // segmento di caricamento (CS)
extern void huge *_farMemBase;   // ptr huge a mem libera dos (varia atomatic.)
extern void huge *_farMemTop;    // ptr huge a fine mem libera dos (varia aut.)
extern void *_endOfSrvc;         // offset fine funzioni dummy dei servizi
extern void *_endOfCode;         // offset fine codice eseguibile
extern void *_endOfData;         // offset fine spazio dati (= Drvr)
extern void *_endOfDrvr;         // offset fine spazio driver
extern void *_freeArea;          // offset area statica libera (ex-stack)
extern unsigned _freeAreaDim;    // dimensione dell'area statica libera
extern int _cmdArgsN;            // num. parametri cmd line (argc)
extern char **_cmdArgs;          // array ptr param. cmd line (argv)

#ifdef __cplusplus
}
#endif

// typedefs per semplificare le dichiarazioni

typedef  unsigned char      BYTE;
typedef  unsigned int       WORD;
typedef  unsigned long      DWORD;

// strutture di vario tipo

typedef struct {                    // struct per Bios Parameter Block (pag. 364)
    WORD      secsize;                  // bytes in un settore
    BYTE      clusecs;                  // settori in un cluster
    WORD      ressecs;                  // settori riservati prima della FAT
    BYTE      fatno;                    // numero di FATs esistenti
    WORD      rootntr;                  // numero di entries nell root
    WORD      totsecs;                  // numero di settori nel disco
    BYTE      mediadb;                  // Media Descriptor Byte
    WORD      fatsecs;                  // numero di settori in una copia della FAT
} BPBLK;

// GESTIONE DEL DEVICE DRIVER HEADER

// la struct e la union seguenti possono essere utilizzate per la gestione del campo
// nome logico del device driver header (pag. 358): infatti la union consente di
// utilizzare gli 8 bytes a disposizione come una stringa di caratteri (e' il caso
// dei character device driver) oppure come una struttura blkName, definita come
```

```
// 1 byte (numero i unita' supportate) e un campo riservato di 7 bytes (e' il caso
// dei block device driver)

typedef struct {                    // struttura Nome Logico per Block Device
    BYTE nUnits;                            // unita' supportate
    char breserved[7];                      // riservato
} blkName;

typedef union {                     // union Nome Logico per Dev. Driver Header
    char    cname[8];                       // nome character device
    blkName bn;                             // nome block device
} h_logName;

// La struct DevDrvHeader ricalca la struttura del device driver header, consentendo
// altresi' l'uso della union h_logName per la gestione del nome logico del device

typedef struct {                    // struttura per Device Driver Header
    void far *nextDrv;                      // ptr al prossimo device driver
    WORD      attrib;                       // device attribute word
    WORD      stratOff;                     // offset della Strategy routine
    WORD      intrOff;                      // offset della Intterupt routine
    h_logName ln;                           // gestione nome logico del device
} DevDrvHeader;

// GESTIONE DEL REQUEST HEADER

// E' definita una struct per la parte di request header differenziata per ogni
// specifico servizio ed una union che le comprende tutte. Vi e' poi una struct
// che rappresenta la parte fissa del reqest header piu' la union da utilizzare
// per il servizio. In altre parole, i templates di seguito definiti consentono
// di gestire il request header come una struttura (la stessa per tutti i servizi)
// che rappresenta la parte fissa, alla quale ne e' "accodata" una seconda a
// scelta tra quelle definite appositamente per i vari servizi.

     // parti variabili per i diversi servizi

typedef struct {                    //  servzio 0 (init)
    BYTE      nUnits;
    void far *endAddr;
    char far *cmdLine;
    BYTE      firstUnit;
} c_init;

typedef struct {                    //  1 (media check)
    BYTE      mdByte;
    BYTE      retByte;
    char far *vLabel;
} c_mediaCheck;

typedef struct {                    //  2 (build BPB)
    BYTE      mdByte;
    BYTE far *trAddr;
    BPBLK    *bpb;
} c_buildBPB;

typedef struct {                    //  3 (input IOCTL)
    BYTE      mdByte;
    BYTE far *trAddr;
    WORD      itemCnt;
    WORD      startSec;
    char far *vLabel;
} c_inputIOCTL;

typedef struct {                    //  4 (input)
```

```
    BYTE      mdByte;
    BYTE far *trAddr;
    WORD      itemCnt;
    WORD      startSec;
    char far *vLabel;
} c_input;

typedef struct {                    //  5 (non destructive input)
    BYTE      readCh;
} c_inputND;

typedef struct {                    //  6 (input status)
    BYTE      dummy;
} c_inputStatus;

typedef struct {                    //  7 (flush input buffers)
    BYTE      dummy;
} c_inputFlush;

typedef struct {                    //  8 (output)
    BYTE      mdByte;
    BYTE far *trAddr;
    WORD      itemCnt;
    WORD      startSec;
    char far *vLabel;
} c_output;

typedef struct {                    //  9 (output with verify)
    BYTE      mdByte;
    BYTE far *trAddr;
    WORD      itemCnt;
    WORD      startSec;
    char far *vLabel;
} c_outputVerify;

typedef struct {                    // 10  0A (output status)
    BYTE      dummy;
} c_outputStatus;

typedef struct {                    // 11  0B (flush output buffers)
    BYTE      dummy;
} c_outputFlush;

typedef struct {                    // 12  0C (output IOCTL)
    BYTE      mdByte;
    BYTE far *trAddr;
    WORD      itemCnt;
    WORD      startSec;
    char far *vLabel;
} c_outputIOCTL;

typedef struct {                    // 13  0D (device open)
    BYTE      dummy;
} c_deviceOpen;

typedef struct {                    // 14  0E (device close)
    BYTE      dummy;
} c_deviceClose;

typedef struct {                    // 15  0F (is media removable)
    BYTE      dummy;
} c_mediaRemove;

typedef struct {                    // 16  10 (output until busy)
```

```
    BYTE      mdByte;
    BYTE far *trAddr;
    WORD      itemCnt;
    WORD      startSec;
    char far *vLabel;
} c_outputBusy;

typedef struct {                    // 19  13 (generic IOCTL interface)
    BYTE      category;
    BYTE      function;
    WORD      siReg;
    WORD      diReg;
    void far *packet;
} c_genericIOCTL;

typedef struct {                    // 23  17 (get logical device)
    BYTE      dummy;
} c_getLogicalDev;

typedef struct {                    // 24  18 (set logical device)
    BYTE      dummy;
} c_setLogicalDev;

    // union raggruppante le struct che descrivono la parte variabile di request
    // header per i vari servizi

typedef union {
    c_init          initReq;
    c_mediaCheck    mCReq;
    c_buildBPB      bBReq;
    c_inputIOCTL    iIReq;
    c_input         iReq;
    c_inputND       iNReq;
    c_inputStatus   iSReq;
    c_inputFlush    iFReq;
    c_output        oReq;
    c_outputVerify  oVReq;
    c_outputStatus  oSReq;
    c_outputFlush   oFReq;
    c_outputIOCTL   oIReq;
    c_deviceOpen    dOReq;
    c_deviceClose   dCReq;
    c_mediaRemove   mRReq;
    c_outputBusy    oBReq;
    c_genericIOCTL  gIReq;
    c_getLogicalDev gLReq;
    c_setLogicalDev sLReq;
} cParms;

    // struct rappresentante il request header (5 campi per la parte fissa piu' la
    // union per la parte variabile). La typedef consente di definire, per
    // comodita', tipi di dato corrispondenti al request header stesso,
    // all'indirizzo near e all'indirizzo far di un request header.

typedef struct {
    BYTE   length;
    BYTE   unitCode;
    BYTE   command;
    WORD   status;
    BYTE   reserved[8];
    cParms cp;
} RequestHeader, *RequestHeaderP, far *RequestHeaderFP;

// DICHIARAZIONE DEGLI ITEMS DEFINITI NEL MODULO DI STARTUP (DDHEADER.ASM, pag. 386)
```

```
extern RequestHeaderFP RHptr;        // puntatore al request header
extern DevDrvHeader    DrvHdr;       // header del device driver

// MACRO di comodo

// La macro che segue puo' essere utilizzata per settare nella parte variabile del
// request header (per il servizio 0) l'indirizzo di fine codice residente del
// device driver. Per scaricare il driver dalla memoria evitandone l'installazione
// e' sufficiente passarle 0. Vedere la costante manifesta NOT_RESIDENT sopra
// definita e la funzione discardDriver() a pag. 423, il cui utilizzo sostituisce la
// chiamata setResCodeEnd(NOT_RESIDENT). La setResCodeEnd() fornisce, tra l'altro,
// un esempio di utilizzo delle strutture e della union definite per manipolare il
// request header.

#define  setResCodeEnd(off)       (RHptr->cp.initReq.endAddr = MK_FP(_CS,off));

#endif  // __BZDD_H
```

E' sufficiente includere `BZDD.H` nel sorgente C del device driver per poter utilizzare tutte le funzioni di libreria, le costanti manifeste, le macro e i template di struttura.

La utility per modificare gli header

Lo startup module e la libreria ci consentono, come vedremo a pag. 439, di scrivere un device driver interamente in linguaggio C; tuttavia ci occorre ancora uno strumento. Infatti, il device driver header è incorporato nello startup module (pag. 386): questo viene compilato una volta per tutte, mentre alcuni campi dello header, quali device attribute word e nome logico (vedere pag. 359 e dintorni) variano per ogni driver. Non ci sono scappatoie: o ci si adatta a riassemblare ogni volta `DDHEADER.ASM`, o si modificano i campi del device driver header direttamente nel file binario risultante da compilazione e linking[369]. Alla seconda ipotesi può facilmente fornire supporto una utility appositamente confezionata: il listato di `DRVSET.C` è presentato e commentato di seguito.

```
/**************************************************************

    DRVSET.C - Barninga Z! - 1994

    Utility per modificare la device attribute word e il logical name nello header
    dei device driver. Funziona con qualsiasi device driver (purché non .EXE)
    anche se non realizzato con la libreria toolkit. Lanciare DRVSET con:

    drvset [opzioni] nome_di_file

    dove:

    nome_di_file   e' il nome del device driver da modificare
    opzioni        puo' essere:
                   -b o -d o -h e -n

    Le opzioni -b, -d e -h sono alternative tra loro e devono essere seguite (senza
    spazi frapposti) dalla nuova device attribute word in binario, decimale e,
    rispettivamente, esadecimale.
```

[369] A dire il vero esiste una terza possibilità: scorporare il device driver header da `DDHEADER.ASM`: si otterebbe così un (piccolo) `DDHEADER.ASM`, contenente il solo device driver header, e un `DDSTART.ASM`, contenente tutto il resto. `DDSTART.ASM` potrebbe essere assemblato una volta per sempre, mentre `DDEHADER.ASM` dovrebbe essere editato e riassemblato per ogni driver. Inoltre bisognerebbe ricordarsi sempre di specificare al linker, prima dell'object file risultante dalla compilazione del sorgente C, `DDHEADER.OBJ` e `DDSTART.OBJ`, in questo preciso ordine.

```
    L'opzione -n deve essere seguita (senza spazi frapposti) dal nome logico del
    device driver, che viene troncato o completato con blanks, se necessario, e
    convertito in maiuscole. Nel caso di block device driver, in luogo del nome
    logico deve essere specificato il numero di unita' supportate, racchiuso tra
    barre ('/').

    I campi dello header sono aggiornati solo previa conferma da parte dell'utente.

    Compilato con Borland C++ 3.1

    bcc drvset.c parseopt.obj

    Circa PARSEOPT.OBJ vedere pag. 479 e seguenti.

************************************************************/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <ctype.h>

#include <parseopt.h>        // per la gestione della command line (vedere pag. 479)

#define  PRG          "DRVSET"
#define  VER          "1.0"
#define  YEAR         "1994"
#define  SWCHAR       '-'
#define  BLANK        ' '
#define  UFLAG        '/'
#define  ILLMSG       "Illegal Option"
#define  ATTR_MASK    0x17A0              // tutti i bits illeciti nella attrib word
#define  BIT_15       0x8000                              // character device driver
#define  NAMELEN      8
#define  MIN_UNITS    1L
#define  MAX_UNITS    26L
#define  MAX_LINE     128

typedef struct {                          // struttura gestione device driver header
    long     nextDev;
    unsigned attrib;
    unsigned strategyOff;
    unsigned interruptOff;
    char     name[NAMELEN];
} DEVHDR;            // la typedef consente di usare DEVHDR per dichiarare variabili

// prototipi di gestione delle opzioni di command line (vedere pag. 479 e seguenti)

int valid_b(struct OPT *vld,int cnt);
int valid_d(struct OPT *vld,int cnt);
int valid_h(struct OPT *vld,int cnt);
int valid_n(struct OPT *vld,int cnt);
int err_handler(struct OPT *vld,int cnt);
int name_flag(struct OPT *vld,int cnt);

// prototipi delle altre funzioni

int main(int argc,char **argv);
void checkAttrBits(void);
int confirm(char *prompt,char yes,char no);
void displayHeader(DEVHDR *hdr,char *title);
int setDevDrvHdr(char *fname);
```

```
DEVHDR DevDrvHdr;  // globale per semplicita'; e' la struttura per gestire lo header

const char *helpStr = "\
filename  is :  the name of the device driver file to be updated\n\
option(s) are:\n\n\
  One of the following:\n\
        -bBinAttr\n\
        -dDecAttr\n\
        -hHexAttr\n\
    where BinAttr, DecAttr and HexAttr are the Device Driver Attribute Word\n\
    in binary, decimal or hexadecimal notation.\n\
  And/or one of the following:\n\
        -nDevDrvName (for character devive driver)\n\
        -n/LogUnits/ (for block device driver)\n\
    Device Driver Logical Name (will be uppercased and truncated if necessary).\n\
    If Block Device Driver, /LogUnits/ specifies the number of Supported\n\
    Logical Units (slashes must be typed).\n\n\
*** No update done. ***\n\
";

const char *invAttrib = "%s: Invalid Device Driver Attribute Word.\n";

const char *invName   = "%s: Invalid Device Driver Logical Name.\n";

const char *invUnits  = "%s: Invalid Device Driver Supported Units.\n";

const char *invFile   = "%s: Too many filenames specified.\n";

const unsigned char nonFNameChars[] = {           // caratteri illeciti in nomi file
    0x22,0x2A,0x2C,0x2E,0x2F,0x3A,0x3B,0x3C,0x3D,0x3E,0x5B,0x5C,0x5D,0x7C,0x81,
    0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x91,0x93,0x94,
    0x95,0x96,0x97,0x98,0xA0,0xA1,0xA2,0xA3,0xA4,NULL
};

unsigned optCnt;                    // contatore opzioni: massimo una tra -b, -h, -d
unsigned nameFlag;                                           // nome file specificato?


// FUNZIONI DI CONTROLLO DELLE OPZIONI DELLA COMMAND LINE

#pragma  warn -par
#pragma  warn -rvl

// La funzione valid_b() effettua i controlli sul parametro specificato per
// l'opzione -b, onde verificare la correttezza dei bits impostati per la device
// attribute word. Il parametro specificato e' un numero BINARIO. Non vengono
// effettuati controlli sulla coerenza reciproca dei bits settati.

int valid_b(struct OPT *vld,int cnt)                        // convalida opzione "b"
{
    extern unsigned optCnt;
    extern DEVHDR DevDrvHdr;
    extern const char *invAttrib;
    char *ptr;
    register i;

    if(optCnt++)
        err_handler(NULL,NULL);
    for(ptr = vld->arg; *ptr == '0'; ptr++);
    if(strlen(ptr) > 16) {
        fprintf(stderr,invAttrib,PRG);
        err_handler(NULL,NULL);
    }
    strrev(ptr);
```

```
    for(i = 0; ptr[i]; i++)
        switch(ptr[i]) {
            case '1':
                DevDrvHdr.attrib += (1 << i);
            case '0':
                break;
            default:
                fprintf(stderr,invAttrib,PRG);
                err_handler(NULL,NULL);
        }
    checkAttrBits();
}

// La funzione valid_d() effettua i controlli sul parametro specificato per
// l'opzione -d, onde verificare la correttezza dei bits impostati per la device
// attribute word. Il parametro specificato e' un numero DECIMALE. Non vengono
// effettuati controlli sulla coerenza reciproca dei bits settati.

int valid_d(struct OPT *vld,int cnt)                        // convalida opzione "d"
{
    extern unsigned optCnt;
    extern DEVHDR DevDrvHdr;
    extern const char *invAttrib;
    register temp;

    if(optCnt++)
        err_handler(NULL,NULL);
    if(atol(vld->arg) > UINT_MAX) {
        fprintf(stderr,invAttrib,PRG);
        err_handler(NULL,NULL);
    }
    if((temp = atoi(vld->arg)) < 0) {
        fprintf(stderr,invAttrib,PRG);
        err_handler(NULL,NULL);
    }
    DevDrvHdr.attrib = temp;
    checkAttrBits();
}

// La funzione valid_h() effettua i controlli sul parametro specificato per
// l'opzione -h, onde verificare la correttezza dei bits impostati per la device
// attribute word. Il parametro specificato e' un numero ESADECIMALE. Non vengono
// effettuati controlli sulla coerenza reciproca dei bits settati.

int valid_h(struct OPT *vld,int cnt)                        // convalida opzione "h"
{
    extern unsigned optCnt;
    extern DEVHDR DevDrvHdr;
    extern const char *invAttrib;
    register i;

    if(optCnt++)
        err_handler(NULL,NULL);
    for(i = 0; vld->arg[i] == '0'; i++);
    if(strlen(vld->arg+i) > 4) {
        fprintf(stderr,invAttrib,PRG);
        err_handler(NULL,NULL);
    }
    sscanf(vld->arg+i,"%X",&DevDrvHdr.attrib);
    checkAttrBits();
}

// La valid_n() controlla la validita' del nome logico specificato per il device e
// lo copia nel campo apposito del device driver header, trasformando tutti i
```

```
// caratteri in maiuscoli, troncandolo se piu' lungo di 8 caratteri e aggiungendo
// spazi a "tappo" se piu' corto. Nel caso sia usata la sintassi /units/ per
// specificare il numero di unita' supportate (block device driver) controlla la
// correttezza sintattica e la validita' del parametro e lo copia nel primo byte
// del campo. Il controllo tra tipo di parametro e tipo di device e' effettuato
// dalla name_flag(), listata poco sotto.

int valid_n(struct OPT *vld,int cnt)                      // convalida opzione "n"
{
    extern const unsigned char nonFNameChars[];
    extern DEVHDR DevDrvHdr;
    extern const char *invName;
    extern const char *invUnits;
    static int instance;
    register i;
    long units;
    char *ptr;
    char line[MAX_LINE];

    if(instance++)
        err_handler(NULL,NULL);                           // consentito solo un nome
    if(*vld->arg == UFLAG) {                                // BLOCK DEVICE DRIVER
        if(sscanf(vld->arg+1,"%ld%s",&units,line) != 2) {
            fprintf(stderr,invUnits,PRG);
            err_handler(NULL,NULL);
        }
        if((units > MAX_UNITS) || (units < MIN_UNITS)) {
            fprintf(stderr,invUnits,PRG);
            err_handler(NULL,NULL);
        }
        if(strcmp(line,"/")) {
            fprintf(stderr,invUnits,PRG);
            err_handler(NULL,NULL);
        }
        DevDrvHdr.name[0] = (unsigned char)units;
    }
    else {                                                // CHARACTER DEVICE DRIVER
        ptr = vld->arg;
        if(strpbrk(strupr(ptr),(char *)nonFNameChars)) {
            fprintf(stderr,invName,PRG);
            err_handler(NULL,NULL);
        }
        strncpy(DevDrvHdr.name,ptr,NAMELEN);
        for(i = strlen(ptr); i < NAMELEN; i++)
            DevDrvHdr.name[i] = BLANK;            // lunghezza fissa 8 blank padded
    }
}

// La err_handler() gestisce il caso di opzione errata

int err_handler(struct OPT *vld,int cnt)                  // gestione opz. errate
{
    extern const char *helpStr;

    fprintf(stderr,"%s: Syntax is:\n%s option[s] filename\n\n%s",PRG,PRG,
                                                                    helpStr);
    exit(1);
}

// la name_flag() e' chiamata dalla parseopt() una volta per ogni parametro
// non-option incontrato sulla command line, pertanto e' chiamata una sola volta
// se sulla cmd line e' specificato un solo nome di file. Tramite il contatore
// nameFlag verifica di non essere chiamata piu' di una volta. Inoltre essa
// controlla che vi sia coerenza tra il tipo di device driver indicato nella
```

```
// device attribute word (bit 15 settato) e il tipo di parametro per l'opzione
// -n (nome logico o numero di unita').

int name_flag(struct OPT *vld,int cnt)                // gestione nome file
{
        extern DEVHDR DevDrvHdr;
        extern unsigned nameFlag;
        extern const char *invAttrib;
        extern const char *invFile;

    if(nameFlag++) {
        fprintf(stderr,invFile,PRG);
        err_handler(NULL,NULL);
    }
    if((DevDrvHdr.name[0] < BLANK) && (DevDrvHdr.attrib & BIT_15)) {
        fprintf(stderr,invAttrib,PRG);
        err_handler(NULL,NULL);
    }
}


#pragma  warn .par
#pragma  warn .rvl

// Struttura per la gestione delle opzioni (associa ogni opzione lecita alla
// funzione corrispondente).

struct VOPT valfuncs[] = {
    {'?',err_handler},
    {'b',valid_b},
    {'d',valid_d},
    {'h',valid_h},
    {'n',valid_n},
    {ERRCHAR,err_handler},
    {NULL,name_flag}
};

// stringa di definizione delle opzioni (se seguite dai due punti richiedono un
// parametro)

const char *optionS = "?b:d:h:n:";

// FUNZIONI: main(), poi tutte le altre in ordine alfabetico

int main(int argc,char **argv)
{
    extern unsigned optCnt;
    extern unsigned nameFlag;
    extern const char *optionS;
    extern struct VOPT valfuncs[];
    struct OPT *opt;

    fprintf(stderr,"%s %s - Set DevDrv Header - Barninga Z! %s. -? help\n\n",
                                                                PRG,VER,YEAR);

// main() usa perseopt (da PARSEOPT.C, vedere pag. 479) per analizzare le opzioni
// specificate sulla command line. Sono queste ad effettuare tutti i controlli
// (vedere sopra).

    if(!(opt = parseopt(argc,argv,(char *)optionS,SWCHAR,ILLMSG,ILLMSG,
                                                                 valfuncs))) {
        perror(PRG);
        return(1);
    }
```

```
    if(!optCnt) {
        fprintf(stderr,"%s: No option specified.\n",PRG);
        return(1);
    }
    if(!nameFlag) {
        fprintf(stderr,"%s: No filename specified.\n",PRG);
        return(1);
    }
    return(setDevDrvHdr(opt[opt[0].opt].arg));
}

// Controlla che i bits settatti nella attribute word del device driver header
// non siano tra quelli riservati DOS (che devono essere zero)

void checkAttrBits(void)
{
    if(DevDrvHdr.attrib & ATTR_MASK) {
        fprintf(stderr,invAttrib,PRG);
        err_handler(NULL,NULL);
    }
}

// Chiede conferma all'utente. Dal momento che attende lo standard input, la
// conferma puo' essere letta da un file (con la redirezione '<') contenente
// la lettera 'Y' o 'N' seguita da un CR LF. Cio' risulta utile nei casi in cui
// si voglia automatizzare l'operazione di agiornamento, ad esempio in un
// batch file di compilazione del device driver.

int confirm(char *prompt,char yes,char no)
{
    int ch;

    fprintf(stderr,prompt,yes,no);
    do {
        ch = toupper(getch());
    } while((ch != yes) && (ch != no));
    fprintf(stderr," %c\n\n",ch);
    return((ch == yes) ? 1 : 0);
}

// Visualizza i campi dello header

void displayHeader(DEVHDR *hdr,char *title)
{
    register i = 0;

    fprintf(stdout,"%s\n",title);
    fprintf(stdout,"\tNext Device Address:      %Fp\n",hdr->nextDev);
    fprintf(stdout,"\tAttribute Word:           %04X\n",hdr->attrib);
    fprintf(stdout,"\tStrategy Routine Offset:  %04X\n",hdr->strategyOff);
    fprintf(stdout,"\tInterrupt Routine Offset: %04X\n",hdr->interruptOff);
    fprintf(stdout,"\tLogical Name:             \"");
    if(*hdr->name < BLANK) {
        i = 1;
        putch(*hdr->name);
    }
    for( ; i < NAMELEN; i++)
        fputc(hdr->name[i],stdout);
    fprintf(stdout,"\"\n\n");
}

// Legge lo header attuale del driver e chiama displayHeader() una prima volta per
// visualizzarne i campi. Successivamente visualizza, sempre tramite displayHeader()
// i campi come saranno scritti nel driver in base ai parametri passati sulla
```

```
// command line e attende conferma via confirm().

int setDevDrvHdr(char *fname)
{
    extern DEVHDR DevDrvHdr;
    DEVHDR fileHdr;
    FILE *file;

    if(!(file = fopen(fname,"r+b"))) {
        perror(PRG);
        return(1);
    }
    if(fread(&fileHdr,sizeof(DEVHDR),1,file) < 1) {
        perror(PRG);
        return(1);
    }
    displayHeader(&fileHdr,"Current Header Fields:");
    fileHdr.attrib = DevDrvHdr.attrib;
    strncpy(fileHdr.name,DevDrvHdr.name,NAMELEN);
    displayHeader(&fileHdr,"New Header Fields:");
    if(confirm("Confirm Update (%c/%c)?",'Y','N')) {
        rewind(file);
        if(fwrite(&fileHdr,sizeof(DEVHDR),1,file) < 1) {
            perror(PRG);
            fprintf(stdout,"%s: %s may have been partially modified.\n",PRG,
                                                              strupr(fname));
            return(1);
        }
        fprintf(stdout,"%s: %s updated successfully.\n",PRG,strupr(fname));
    }
    else
        fprintf(stdout,"%s: %s not updated.\n",PRG,strupr(fname));
    return(0);
}
```

Il programma non si preoccupa di controllare la coerenza reciproca dei bit della attribute word, perciò è compito del programmatore evitare di violare le regole che stabiliscono quali bit debbano, contemporaneamente, avere medesimo o diverso valore. Tuttavia, è verificato che i bit riservati al DOS siano lasciati a `0`. E' effetuato un solo controllo di carattere logico: la consistenza tra utilizzo del campo riservato al nome logico nel device driver header e tipo del driver, come desumibile dalla attribute word impostata (vedere pag. 358 e dintorni per i particolari).

Compilando il sorgente, occorre richiedere che ad esso sia consolidato `PARSEOPT.OBJ`, necessario alla gestione delle opzioni della command line, come descritto a pag. 479 e seguenti. Il comando

```
bcc drvset.c parseopt.obj
```

consente di ottenere `DRVSET.EXE` che, invocato con l'opzione `-?` visualizza un testo di aiuto.

Vediamone un esempio di utilizzo:

```
drvset -h8000 -nZ! devprova.sys
```

Il comando presentato modifica il device driver header di `DEVPROVA.SYS`, impostando la device attribute word a `8000h` (solo il bit 15 a `1`, per indicare che si tratta di un character device driver) ed il nome logico del device `"Z!"`. L'output prodotto da `DRVSET` è analogo al seguente:

```
Current Header Fields:
    Next Device Address:      FFFF:FFFF
    Attribute Word:           0000
    Strategy Routine Offset:  038A
    Interrupt Routine Offset: 0395
```

```
    Logical Name:             "        "

New Header Fields:
    Next Device Address:      FFFF:FFFF
    Attribute Word:           8000
    Strategy Routine Offset:  038A
    Interrupt Routine Offset: 0395
    Logical Name:             "Z!      "

Confirm Update (Y/N)?
```

Digitando `Y` o `N`, `DRVSET` tenta, o meno, di modificare lo header del file `DEVPROVA.SYS`, visualizzando poi un messaggio di conferma dell'avvenuta modifica o della rinuncia. Se nella directory corrente è presente un file, ad esempio `YES.TXT`, costituito di una sola riga di testo contenente il solo carattere `Y` (in pratica il file si compone di 3 byte: `Y`, `CR` e `LF`), e si redirige lo standard input (vedere pag. 116 e seguenti) di `DRVSET` a quel file, la risposta affermativa alla domanda diviene automatica: il comando

```
drvset -h8000 -nZ! devprova.sys < yes.txt
```

si rivela particolarmente adatto ad essere inserito in un file batch di compilazione e linking del driver `DEVPROVA.SYS` (vedere, ad esempio, pag. 450).

Il toolkit al lavoro

Abbiamo a disposizione un nuovo startup module, una libreria di funzioni dedicate ai device driver e un programma in grado di modificare secondo le nostre esigenze la device attribute word e il logical name nel device driver header del file binario risultante dal linking. Per avere un device driver manca soltanto... il sorgente C, che deve implementare tutte le funzionalità desiderate per il driver, senza mai perdere d'occhio i necessari requisiti di efficienza. Vediamo un elenco delle principali regole a cui attenersi nello scrivere il driver.

1) Nel sorgente C deve essere definita la funzione `init()`, le cui caratteristiche sono discusse a pagina 441.

2) Nel sorgente C devono inoltre essere definite tutte le funzioni che implementano i servizi desiderati. Dette funzioni devono necessariamente uniformarsi ai prototipi dichiarati in `BZDD.H` (pag. 425 e seguenti). Ad esempio, il servizio 19 (generic IOCTL request), deve sempre essere implementato da una funzione, definita nel sorgente C, avente prototipo `int genericIOCTL(void)`: tutte queste funzioni devono restituire un intero e non possono richiedere parametri.

3) L'intero restituito dalle funzioni di servizio rappresenta lo stato dell'operazione eseguita ed è utilizzato dalla `Interrupt()` per valorizzare la status word (pag. 361) nel device driver request header. Allo scopo possono essere utilizzate le costanti manifeste definite in `BZDD.H`.

4) L'inizializzazione del driver deve includere una chiamata alla macro `setResCodeEnd()` o alla funzione `discardDriver()`. Vedere pag. 441.

5) Possono essere chiamate liberamente le funzioni di libreria C, tenendo presente che il loro utilizzo nella parte residente del driver comporta i problemi tipici dei programmi TSR discussi a pag. 289. Si noti che la parte residente si compone almeno di tutte le funzioni di servizio e di quelle da esse invocate direttamente o indirettamente, mentre non sono necessariamente residenti la `init()` e le funzioni chiamate esclusivamente all'interno di questa.

6) Va tenuto presente che vi sono, comunque, limiti all'uso delle funzioni di libreria C: alcune di esse non possono essere referenziate in quanto incoerenti con la logica di implementazione dei device driver. Ad esempio, non è possibile effettuare allocazioni di memoria `far` (`farmalloc()`, etc.): tali operazioni falliscono sistematicamente (`farmalloc()` restituisce sempre `0L`) in quanto i device driver non hanno far heap. Inoltre i device driver sono installati residenti da parte del DOS, perciò non deve essere usata la funzione `keep()`. Ancora, dal momento che i device driver non terminano mai la propria esecuzione, non è possibile utilizzare `exit()`, `_exit()`, `abort()`, etc.. Inoltre, vista la mancanza di environment, i device driver non possono usare `getenv()` e `putenv()`.

7) Alcune funzioni di libreria non possono essere utilizzate nella fase di inizializzazione, mentre possono esserlo nell'espletamento di tutti gli altri servizi. Ad esempio, l'allocazione di memoria via DOS (int 21h, servizio 48h) è possibile solo a caricamento del sistema completato, quindi solamente dopo l'installazione di tutti i device driver e dell'interprete dei comandi: pertanto `allocmem()` fallisce se chiamata da `init()` o da sue subroutine, mentre può avere successo se chiamata dalle funzioni di servizio durante la sessione di lavoro del computer.

8) Le variabili globali possono essere dichiarate e referenziate come in qualsiasi programma C; si tenga però presente che esse risiedono in memoria oltre il codice dell'ultima funzione estratta dalle librerie: ciò può porre vincoli qualora si intenda ridurre al minimo l'ingombro in memoria della porzione residente del driver. L'ostacolo può essere facilmente aggirato col solito trucco delle funzioni jolly, analogamente ai TSR (vedere pag. 280).

9) La compilazione del sorgente C deve sempre essere effettuata con le opzioni `-mt` (modello di memoria tiny; vedere pag. 143 e seguenti) e `-c` (generazione del file `.OBJ` senza linking). Il linker deve essere lanciato successivamente, con le opzioni `-t` (generazione di un file `.COM`) e `-c` (case sensitivity), elencando `DDHEADER.OBJ` (lo startup module) in testa a tutti i file `.OBJ`; l'ordine in cui elencare le librerie (`BZDD.LIB`; la libreria C per il modello di memoria small `CS.LIB`; le altre librerie eventualmente necessarie) non è fondamentale. Si ricordi, però, che `BZDD.LIB` e `CS.LIB` devono essere sempre indicate, mentre altre librerie devono esserlo solo se in esse si trovano funzioni o simboli comunque referenziati. Infine, il device driver header deve essere modificato con `DRVSET`, secondo necessità. Ad esempio, il character device driver `DEVPROVA.SYS` (nome logico `ZDEV`) può essere ottenuto a partire da `DEVPROVA.C` attraverso i seguenti 3 passi:

```
bcc -c -mt devprova.c
tlink -c -t ddheader.obj devprova.obj,devprova.sys,,bzdd.lib cs.lib altre.lib
drvset -h8000 -nzdev devprova.sys
```

L'operazione di linking produce anche `DEVPROVA.MAP` (file ASCII contenente l'elenco dei simboli pubblici definiti nel driver con i rispettivi indirizzi), che può essere tranquillamente gettato alle ortiche[370].

[370] Beh, vale la pena di dargli almeno un'occhiata: si può capire molto circa la struttura del file binario e la posizione, al suo interno, dei segmenti, delle funzioni e delle variabili globali.

Le complicazioni sono, per la maggior parte, più apparenti che reali. La descrizione della `init()` e qualche esempio lo possono dimostrare.

La funzione init()

Come più volte si è detto, nel sorgente C di ogni device driver realizzato con il toolkit deve essere definita una funzione avente nome `init()`, analogamente a quanto avviene nei comuni programmi C, nei quali deve essere definita una `main()`. In effetti, tra `init()` e `main()` vi sono analogie, in quanto entrambe sono automaticamente chiamate dallo startup module e possono accedere alla command line attraverso i parametri formali; tuttavia le due funzioni sono differenti, in quanto `main()` può accedere anche alle variabili d'ambiente (vedere pag. 105), mentre `init()` non ha tale possibilità, dal momento che i device driver non hanno environment. Inoltre, una istruzione `return` eseguita in `main()` determina sempre la fine dell'esecuzione del programma, mentre in `init()` causa la restituzione del controllo al DOS da parte del driver, che può rimanere, però, residente in memoria (a seconda dell'indirizzo di fine codice residente impostato nel request header). Ancora, `main()` può restituire o meno un valore (in altre parole, può essere dichiarata `int` o `void`), mentre `init()` è obbligatoriamente `int`: il valore restituito è utilizzato dalla `Interrupt()` per impostare la status word (pag. 361) del request header[371].

In particolare, la `init()` può essere definita secondo 3 differenti prototipi:

```
int init(void);
int init(int argc);
int init(int argc,char **argv);
```

Nella prima forma, `init()` non riceve parametri; nella seconda essa rende disponibile un intero, che esprime il numero di argomenti presenti sulla riga di comando del device driver, incluso il nome del driver stesso. La terza forma, oltre all'intero di cui si è detto, rende disponibile un puntatore a puntatore a carattere, cioè un array di puntatori a carattere o, in parole povere, un array di stringhe. Ogni stringa è un argomento della command line: la prima (indice `0`) è il nome del driver (completo di eventuale path); il puntatore all'ultimo argomento è seguito da un puntatore nullo (`NULL`). La stretta parentela con `argc` e `argv` della `main()` dei comuni programmi C è evidente e da essa, del resto, sono derivati i nomi utilizzati[372]; dal punto di vista tecnico essi sono le copie di `_cmdArgsN` e `_cmdArgs` effettuate nello stack da `driverInit()` prima di effettuare la chiamata alla stessa `init()` (vedere pag. 399).

La `init()` è eseguita una sola volta, durante il caricamento del driver da parte del sistema, pertanto deve effettuare, eventualmente tramite funzioni richiamate direttamente o indirettamente, tutte le operazioni necessarie all'inizializzazione del driver. Qualora il device driver abbia la necessità di rilocare il proprio stack iniziale, è proprio `init()` che deve provvedervi, invocando `setStack()` con le precauzioni descritte a pagina 416.

Inoltre `init()` ha l'importante compito di comunicare al DOS se installare o no il driver in memoria e, in caso affermativo, di indicare l'indirizzo del primo byte successivo all'area di RAM destinata al driver stesso. Allo scopo è definita in `BZDD.H` la macro `setResCodeEnd()`, ed esiste in libreria la funzione `discardDriver()`: la prima accetta detto indirizzo quale parametro: va ricordato che si

[371] Va infine sottolineato che il sorgente del device driver può definire anche una `main()`, ma con il ruolo di una funzione qualsiasi: non viene invocata in modo automatico e riceve i parametri che le passa la funzione che la chiama, coerentemente al prototipo definito "per l'occasione".

[372] Come nel caso di `main()`, non è obbligatorio utilizzare `argc` e `argv`: i nomi possono essere liberamente scelti dal programmatore.

tratta di un indirizzo `near`, cioè, in altre parole, della parte offset dell'indirizzo `far`, la cui parte segmento è rappresentata dal valore del registro `CS`. Esempio:

```
....

#include <bzdd.h>

int init(int argc,char **argv)
{
    ....
    if(....) {                                              // in caso di errore...
        ....
        discardDriver();                              // non lascia residente il driver
        return(E_GENERAL))                      // restituisce errore per la status word
    }
    ....
    setResCodeEnd(_endOfDrvr);          // lascia residente tutto il codice del driver
    return(E_OK);                                  // restituisce OK per la status word
}
```

L'indirizzo `_endOfDrvr`, passato a `setResCodeEnd()`, è una variabile, definita nello startup module[373], esprimente l'offset di una porzione di driver fittizia, collocata dal linker in coda al file binario e può validamente rappresentare, di conseguenza, un indirizzo di sicurezza. A `setResCodeEnd()` la `init()` può passare, ad esempio, il proprio indirizzo quando il sorgente sia organizzato in modo tale che `init()` sia definita per prima tra tutte le funzioni transienti e nessuna di queste referenzi funzioni di libreria (toolkit o C):

```
    ....
    setResCodeEnd(init);
    ....
```

E' ovvio che `init()` può valorizzare con l'opportuno indirizzo l'apposito campo del request header accedendo direttamente ad esso, senza utilizzare `setResCodeEnd()`[374]; inoltre, non necessariamente tale operazione deve essere svolta immediatamente prima di eseguire un'istruzione `return`, anche se, spesso, ciò è causato dalla logica stessa dell'algoritmo di inizializzazione.

La `init()` invoca, al contrario, `discardDriver()` (vedere pag. 423) se la procedura di inizializzazione deve concludersi senza rendere residente il device driver: sebbene nel caso dei character device driver si riveli sufficiente chiamare la macro `setResCodeEnd()` con la costante manifesta `NOT_RESIDENT`, definita in `BZDD.H`, o il valore `0` quale parametro, si raccomanda di utilizzare comunque `discardDriver()`, come nell'esempio poco sopra presentato, dal momento che questa è aderente alle indicazioni in materia presenti nella documentazione ufficiale del DOS.

Altre funzioni e macro

Nello startup module sono definite due funzioni, utili per la restituzione di codici di errore alla `Interrupt()` del device driver. Esse sono:

```
int errorReturn(int errcode);
```

[373] Si veda `BZDD.H` per la dichiarazione di questa ed altre variabili esprimenti gli indirizzi di diverse porzioni del device driver. Esse, inoltre, sono descritte a pag. 445.

[374] Come fare? Basta un'occhiata alla definizione della stessa `setResCodeEnd()` in `BZDD.H` per capirlo. Vedere anche, di seguito, la descrizione della modalità di accesso ai campi del request header.

che valorizza il byte meno significativo della status word (pag. 361) del request header con `errcode` e pone a 1 il bit di errore del byte più significativo, e

```
int unSupported(void);
```

che chiama `errorReturn()` passandole quale parametro il codice di errore corrispondente allo stato di servizio non definito. Entrambe le funzioni, come si è detto, sono definite nello startup module: pertanto non provocano l'inclusione nel file binario di moduli `.OBJ` dalle librerie.

In libreria è presente la

```
void discardDriver(void);
```

che ha lo scopo di richiedere al DOS di non installare il device driver in memoria. Il suo utilizzo è descritto a pag. 441, con riferimento alla funzione user-defined `init()`.

Per installare il driver occorre invece chiamare la

```
setResCodeEnd(off);
```

macro definita in `BZDD.H` (pag. 431): `off` rappresenta l'offset, rispetto a `CS`, del primo byte di memoria libera oltre la parte residente del driver e deve essere un valore di tipo `unsigned int`.

L'accesso al device driver request header

La gestione del request header (pag. 360) è di fondamentale importanza, in quanto esso è il mezzo attraverso il quale DOS e device driver si scambiano tutte le informazioni necessarie all'espletamento dei diversi servizi. Quasi tutte le funzioni di servizio devono quindi accedere al request header per conoscere i parametri forniti dal DOS e, spesso, memorizzarvi i risultati delle loro elaborazioni.

L'indirizzo del request header è comunicato dal DOS alla `Strategy()`, la quale lo memorizza in una variabile dichiarata nello startup module, per uso successivo da parte della `Interrupt()` e delle funzioni di servizio. Allo scopo, nel file `BZDD.H` sono definiti template di `struct` e `union`, che consentono l'accesso ai campi delle parti fissa e variabile del request header tramite un puntatore dichiarato globalmente.

In particolare, per ogni servizio è definito un template di struttura che rappresenta tutti i campi della parte variabile del request header secondo le specifiche del servizio medesimo. Detti template sono raggruppati in una `union`, che rappresenta così la parte variabile di tutti i servizi. Il request header è infine rappresentato da un template di struttura, i cui elementi includono i campi della parte fissa e, da ultimo, la `union` definita come descritto. Le `typedef` associate ai template rendono più leggibili e concise eventuali dichiarazioni.

Vediamo un esempio pratico di accesso al request header, avendo sott'occhio il listato di `BZDD.H` (pag. 425 e seguenti): la funzione `mediaCheck()`, che implementa il servizio 1 (vedere pag. 365), deve conoscere il numero e il media ID byte dell'unità disco sulla quale il DOS richiede informazioni per poi restituire il media change code e l'indirizzo `far` dell'etichetta di volume. Il campo media ID byte si trova nella parte fissa del request header, mentre tutti gli altri sono nella parte variabile. Innanzitutto, per chiarezza, in `mediaCheck()` è opportuno redichiarare come `extern` il puntatore al request header:

```
    extern RequestHeaderFP RHptr;
```

Il tipo di dato `RequestHeaderFP` (definito con una `typedef`) indica un puntatore `far` ad una struttura di template `RequestHeader`.

L'accesso al numero dell'unità è ottenuto in modo assai semplice, con l'espressione:

```
RHptr->unitCode
```

Infatti, `unitCode` è un campo (di tipo `BYTE`, cioè `unsigned char`) della parte fissa del request header e, come tale, è direttamente membro del template `RequestHeader`.

Le espressioni che accedono ai campi della parte variabile sono più complesse, in quanto devono tenere presente che questa è rappresentata come una `union`, membro dello stesso template `RequestHeader`, avente nome `cp`: la base dell'espressione per accedere ad ogni campo della parte variabile è dunque

```
RHptr->cp
```

Nella union `cp` occorre, a questo punto, selezionare il template di struttura che rappresenta la parte variabile del request header dello specifico servizio di nostro interesse: quello relativo al servizio 1 ha nome `mCReq`. Ne segue che la base dell'espressione necessaria per accedere ad ogni campo della parte variabile per il servizio 1 è

```
RHptr->cp.mCReq
```

Il gioco è fatto: ogni membro di `mCReq` è, come accennato, un campo della parte variabile per il servizio 1. Le espressioni complete per l'accesso ai campi usati sono pertanto:

```
RHptr->cp.mCreq.mdByte
```

per il media ID byte (di tipo `BYTE`, cioè `unsigned char`);

```
RHptr->cp.mCreq.retByte
```

per il *media change code* (anch'esso di tipo `BYTE`, cioè `unsigned char`), ed infine

```
RHptr->cp.mCreq.vLabel
```

per la *volume label* (di tipo `char far *`).

Anche la macro `setResCodeEnd()` è definita in base alla tecnica descritta, ma della `union` "parte variabile" (`cp`) utilizza il membro struttura che rappresenta proprio la parte variabile del servizio 0 e, all'interno di quest'ultima, il campo opportuno:

```
RHptr->cp.initReq.endAddr
```

Un po' di allenamento consente di orientarsi nel labirinto dei template ad occhi (quasi) chiusi.

Le variabili globali dello startup module

Nel file `BZDD.H` (pag. 425) sono dichiarate (`extern`) le variabili globali accessibili al C definite nello startup module `DDHEADER.ASM` (pag. 386). Alcune di esse sono il perfetto equivalente delle variabili globali definite nello startup code dei normali programmi C:

```
extern int errno;                     // codice di errore
extern unsigned _version;             // versione e revisione DOS
extern unsigned _osversion;           // versione e revisione DOS
extern unsigned char _osmajor;        // versione DOS
extern unsigned char _osminor;        // revisione DOS
extern unsigned long _StartTime;      // timer clock ticks al caricamento
```

La variabile `_psp` è anch'essa definita nello startup code C, ma con differente significato: per un programma essa rappresenta la parte segmento dell'indirizzo al quale è caricato il proprio PSP; nel

caso di un device driver, non essendo presente un PSP, essa rappresenta la parte segmento dell'indirizzo al quale è caricato il driver stesso, cioè il valore del registro CS:

```
extern unsigned _psp;
```

Le altre variabili globali dichiarate in BZDD.H sono caratteristiche del toolkit startup module e contengono dati che possono risultare di qualche utilità per il programmatore.

La variabile

```
extern unsigned _baseseg;
```

è del tutto equivalente alla _psp.

La variabile

```
extern unsigned _systemMem;
```

contiene il numero di Kb di memoria convenzionale installati sul personal computer.

Le variabili

```
extern void huge *_farMemBase;
extern void huge *_farMemTop;
```

esprimono gli indirizzi dell'inizio e, rispettivamente, della fine della memoria convenzionale libera, compresa tra la RAM occupata dal device driver e quella occupata dalla routine SYSINIT del DOS (vedere pag. 353). La memoria compresa tra i due indirizzi è disponibile per il device driver, ma va tenuto presente che il valore di _farMemTop è determinato empiricamente ed è quindi da utilizzare con cautela. Dette variabili sono significative solo durante l'esecuzione di init() e vengono azzerate quando essa termina.

Alcune variabili rappresentano puntatori near a zone di memoria "notevoli":

```
extern void *_endOfSrvc;
extern void *_endOfCode;
extern void *_endOfData;
extern void *_endOfDrvr;
```

La _endOfSrvc contiene l'indirizzo del primo byte successivo all'ultima delle funzioni di servizio del driver; la _endOfCode punta al primo byte successivo al codice eseguibile del driver[375]; la _endOfData punta al primo byte successivo al segmento riservato ai dati statici, globali e alle costanti. Detto indirizzo coincide con quello di inizio della memoria libera al di sopra del driver: _endOfDrvr contiene perciò il medesimo valore di _endOfData.

Le variabili

```
extern void *_freArea;
extern unsigned _freAreaDim;
```

sono significative solamente dopo la rilocazione dello stack originale. Se la chiamata a setStack() ha successo (vedere pag. 416), _freArea contiene l'indirizzo near dello stack originale, ora riutilizzabile come generico buffer, mentre _freAreaDim ne esprime la dimensione in byte. Se lo stack non viene rilocato (setStack() non è chiamata o fallisce) esse contengono entrambe 0.

[375] Si noti che lo stack del driver è sempre all'interno dell'area del codice eseguibile: infatti, lo stack originale è esplicitamente definito (startup module) nel code segment, mentre quello rilocato lo è implicitamente, essendo definito mediante una funzione (fittizia).

Infine, le variabili

```
extern int _cmdArgsN;
extern char **_cmdArgs;
```

sono l'equivalente dei parametri formali attribuibili alla `init()` (pag. 441): `_cmdArgsN` contiene il numero di argomenti della command line del driver, incluso il pathname del driver stesso, mentre `_cmdArgs` è un array di puntatori a carattere, ogni elemento del quale punta ad una stringa contenente un argomento della command line: `_cmdArgs[0]` punta al nome del driver, come specificato in `CONFIG.SYS`; `_cmdArgs[_cmdArgsN]` è `NULL`.

Esempio: alcune cosette che il toolkit rende possibili

Il device driver `TESTINIT.SYS` fa ciò che il nome suggerisce: pasticcia nella `init()` per saggiare alcune delle funzionalità offerte dal toolkit: rilocazione dello stack, allocazione dinamica della memoria, gestione dei file via stream... Il listato è presentato di seguito; i numerosi commenti in esso presenti rendono superfluo soffermarsi oltre sulle sue caratteristiche.

```
/****************************************************************

    TESTINIT.C - Barninga Z! - 1994

    Device driver di prova - funzionalita' toolkit

    Il driver effettua varie operazioni di inizializzazione in init()
    ma non si installa residente in memoria.

    Compilato con Borland C++ 3.1:

    bcc -c -mt testinit.c
    tlink -c -t ddheader.obj testinit.obj,testinit.sys,,bzdd.lib cs.lib
    drvset -h8000 -nZ! testinit.sys

****************************************************************/
#pragma  inline

#include <stdio.h>
#include <conio.h>
#include <alloc.h>

#include <bzdd.h>

#define  MAXLIN  128

// Le variabili extern dichiarate di seguito sono definite nello startup module ma
// non sono dichiarate in BZDD.H (tuttavia sono pubbliche perche' devono essere
// visibili per alcune funzioni di libreria C): le dichiarazioni qui effettuate
// hanno lo scopo di renderle utilizzabili nel listato esclusivamente a scopo di
// debugging e di controllo. I LORO VALORI NON DEVONO ESSERE MODIFICATI.

extern unsigned _newTOS;          // DEBUG
extern unsigned __brklvl;         // DEBUG
extern unsigned __heapbase;       // DEBUG
extern void far *_heapbase;       // DEBUG
extern void far *_heaptop;        // DEBUG

void testMemFile(char **argv);

// La stk() e' la funzione jolly che riserva lo spazio per il nuovo stack del driver

void stk(void)
```

```
{
    asm db 4000 dup(0);                          // 4000 bytes di stack per il driver
}

// La variabile globale base e' inizializzata con l'offset dell'indirizzo di stk()
// mentre len contiene la lunghezza del nuovo stack: esse sono passate a setStack()

unsigned base = (unsigned)stk;
unsigned len = 4000;

// init(): tutte le operazioni di inizializzazione devono essere svolte qui. La
// dichiarazione di init() rende disponibili il numero di parametri della riga di
// comando in CONFIG.SYS (argc) e le stringhe dei parametri stessi (argv). In init()
// sono tranquillamente utilizzate le funzioni di libreria printf() e getch().

int init(int argc,char **argv)
{
    extern RequestHeaderFP RHptr;                   // per accedere al request header
    register unsigned i;

// _baseseg: segmento di caricamento del driver (CS)
// _osmajor e _osminor: versione e revisione DOS

    printf("\nDevice Driver di prova a %04X:0000. DOS %d.%d.\n",
                                                      _baseseg,_osmajor,_osminor);

// visualizzati: l'indirizzo di stk(), il suo offset, la lunghezza del nuovo stack

    printf("stk: %Fp  base: %04X  len: %d\n",(void far *)stk,base,len);

// __brklvl: confine tra heap e stack
// __heapbase: offset di inizio dello heap
// _freArea e _freAreaDim: offset e lunghezza dello stack originale se rilocato e
// quindi disponibile per altri usi. Non e' ancora chiamata setStack(), percio'
// entrambe valgono 0.

    printf("__brklvl: %04X  __heapbase: %04X  _freArea: %04X  Dim: %d\n",
                                       __brklvl,__heapbase,_freArea,_freAreaDim);

// e' chiamata setStack() e il risultato e' memorizzato in i. Si noti che la
// variabile i e' register e RHptr e' extern: e' soddisfatta la condizione di
// non dichiarare in init() variabili che facciano uso dello stack se e'
// usata setStack() (vedere pag. 416). Se i non e' 0, la rilocazione ha avuto
// successo e il suo valore esprime la lunghezza effettiva del nuovo stack

    printf("%d = setStack(%04X,%d)\n",i = setStack(base,len),base,len);

// visualizzate nuovamente __brklvl, __heapbase, _freArea e _freAreaDim: questa
// volta, se la rilocazione ha avuto successo, _freArea e _freAreaDim non sono 0

    printf("__brklvl: %04X  __heapbase: %04X  _freArea: %04X  Dim: %d\n",
                                       __brklvl,__heapbase,_freArea,_freAreaDim);

// visualizzate la lunghezza del nuovo stack e l'offset del nuovo top of stack

    printf("newLen: %d (%04X)  __newTOS: %04X\n",i,i,_newTOS);

// _endOfSrvc: offset dell'indirizzo di fine ultima funzione di servizio del driver
// _endOfCode: offset dell'indirizzo di fine segmento codice eseguibile
// _endOfData: offset dell'indirizzo di fine segmento dati globali e statici
// _endOfDrvr: offset dell'indirizzo di fine spazio occupato in memoria dal driver
// La parte segmento di tutti questi indirizzi e' _baseseg, ovvero _psp, ovvero CS
// Si noti inoltre che il nuovo stack fa sempre parte del segmento di codice
// eseguibile, essendo definito mediante una funzione (fittizia)
```

```
    printf("_endOfSrvc:%04X  _endOfCode:%04X  _endOfData:%04X  _endOfDrvr:%04X\n",
                                    _endOfSrvc,_endOfCode,_endOfData,_endOfDrvr);

// indirizzi (empirici) dell'inizio e fine della memoria libera oltre il driver

    printf("_farMemBase: %Fp    _farMemTop: %Fp\n",_farMemBase,_farMemTop);

// indirizzo del request header

    printf("Request Header a %Fp.\n",RHptr);

// indirizzo della command line. Notare l'espressione di accesso al puntatore alla
// command line (e' un campo della parte variabile del request header specifica del
// servizio 0)

    printf("Indirizzo della command line: %Fp.\n",RHptr->cp.initReq.cmdLine);

// ciclo di visualizzazione dei parametri della command line (sfrutta argc e argv)

    for(i = 0; i < argc; i++)
        printf("P_%02d:::%s:::\n",i,argv[i]);

// invoca la fuzione testMemFile(), definita dopo init()

    testMemFile(argv);

// visualizza il valore passato a setResCodeEnd() (offset di fine parte residente
// del driver) e poi attende la pressione di un tasto per terminare le operazioni di
// inizializzazione

    printf("\nsetResCodeEnd(%04X).\nPremere un tasto...\n",0);
    getch();

// chiama discardDriver(), richiedendo cosi' al DOS di non installare in memoria
// il driver: in effetti lo scopo era unicamente testare il toolkit in init().

    discardDriver();
    return(E_OK);                                                        // tutto OK
}

// testMemFile() effettua operazioni di allocazione e disallocazione di memoria
// e apre e visualizza un file ASCII, il cui nome e' il primo argomento della
// command line di TESTINIT.SYS

void testMemFile(char **argv)
{

// Non siamo in init(), dunque, indipendentemente dall'uso di setStack(), si puo'
// dichiarare un po' di tutto...

    register i;
    char *p[3];
    FILE *f;
    char line[MAXLIN];

    printf("Premere un tasto...\n");
    getch();

// ciclo di allocazioni successive di 100 bytes memoria tramite malloc(). Ad ogni
// iterazione e' visualizzato lo heap libero, l'indirizzo allocato e lo heap residuo

    for(i = 0; i < 3; i++) {
        printf("Coreleft: %u.\n",coreleft());
```

```
        printf("malloc(100): %04X\n",p[i] = malloc(100));
    }
    printf("Coreleft: %u.\n",coreleft());

// ciclo di disallocazione per step successivi della memoria allocata in
// precedenza. Ad ogni ciclo e' visualizzato lo heap libero, l'indirizzo
// disallocato e il nuovo heap libero

    for(i--; i >= 0; i--) {
        printf("free(%04X):\n",p[i]);
        free(p[i]);
        printf("Coreleft: %u.\n",coreleft());
    }
    printf("Premere un tasto...\n\n");
    getch();

// TESTINIT.SYS presume che sulla command line gli sia passato almeno un argomento
// utilizzato qui come nome di file da aprire. Deve essere un file ASCII. Il file
// e' aperto con fopen() e letto e visualizzato riga per riga con fgets() e printf()
// Infine il file e' chiuso con fclose()

    if(!(f = fopen(argv[1],"r")))
        perror("TESTINIT.SYS");
    else {
        while(fgets(line,MAXLIN,f))
            printf(line);
        fclose(f);
        }
}
```

La generazione di `TESTINIT.SYS` a partire da `TESTINIT.C` può essere automatizzata con un semplice file batch:

```
@echo off
REM
REM *** generazione di testinit.obj
REM
bcc -c -mt testinit
if errorlevel 1 goto error
REM
REM *** generazione di testinit.sys
REM
tlink -c -t ddheader testinit,testinit.sys,,bzdd cs
if errorlevel 1 goto error
REM
REM *** attribute word settata per character device driver; nome logico: "Z!"
REM
drvset -h8000 -nZ! testinit.sys < yes.txt
if errorlevel 1 goto error
REM
REM ***  eliminazione file inutili
REM
del testinit.obj
del testinit.map
goto end
REM
REM *** visualizza messaggio in caso di errore
REM
:error
echo TESTINIT.SYS non generato!
REM
REM *** fine batch job
REM
```

```
:end
```

Si noti che DRVSET riceve lo standard input dal file YES.TXT, contenente esclusivamente il carattere "Y" seguito da CR e LF (vedere pag. 438).

TESTINIT.SYS è copiato nella directory root del drive C: dal file batch medesimo, perciò la riga di CONFIG.SYS che ne determina il caricamente deve essere analoga alla seguente:

```
DEVICE=C:\TESTINIT.SYS C:\CONFIG.SYS 1234 " abc 678" DeF
```

Il primo argomento (C:\CONFIG.SYS) è l'unico significativo, in quanto indica il file ASCII che la testMemFile() deve aprire e visualizzare; i rimanenti parametri hanno unicamente lo scopo di verificare il buon funzionamento della funzione di libreria toolkit setupcmd() (vedere pag. 421). Si noti che " abc 678" è un unico parametro, grazie alla presenza delle virgolette, le quali consentono inoltre la prsenza di uno spazio prima dei caratteri abc. Tutti i parametri sono convertiti in caratteri maiuscoli da setupcmd().

Esempio: esperimenti di output e IOCTL

Il driver TESTDEV.SYS gestisce i servizi Output, Output With Verify e Generic IOCTL Request. Si tratta di routine estremamente semplificate, che hanno unicamente lo scopo di dimostrarne le funzionalità di base.

```
/****************************************************************

    TESTDRV.C - Barninga Z! - 1994

    Device driver di prova - funzionalita' toolkit

    Il driver gestisce una funzione di output e di output with
    verify equivalenti e consente di modificare la modalita'
    di output mediante una generic IOCTL request.

    Compilato con Borland C++ 3.1:

    bcc -c -mt testdrv.c
    tlink -c -t ddheader.obj testdrv.obj,testdrv.sys,,bzdd.lib cs.lib
    drvset -h8040 -nzeta testdrv.sys

*****************************************************************/
#pragma  inline

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

#include <bzdd.h>

void colortext(WORD count,BYTE far *buffer);    // funzione di servizio per output()

// costanti manifeste definite per comodita'.

#define  DEFAULT_ATTR    7                             // Bianco/Nero
#define  DEFAULT_PAGE    0                             // Pagina video (unica usata)
#define  BLANK           ' '                           // Spazio

// costanti manifeste definite per supporto a Generic IOCTL Request (servizio 19)
// IOCTL_CATEGORY e' un identificativo del driver, una specie di parola d'ordine
// inventata di sana pianta. Anche i sottoservizi implementati, IOCTL_GETATTR e
// IOCTL_SETATTR, sono stati numerati 1 e 2 per libera scelta.
```

```
#define  IOCTL_CATEGORY  0x62                          // Identificativo
#define  IOCTL_GETATTR   1                             // IOCTL servizio 1
#define  IOCTL_SETATTR   2                             // IOCTL servizio 2

// attrib e' una normale variabile globale, destinata a contenere l'attributo video
// per gli output di testo.

BYTE attrib;

// funzione per la gestione del servizio Write (8). Il suo prototipo e' identico a
// quello dichiarato in BZDD.H. Essa chiama la funzione colortext(), che effettua
// la vera e propria operazione di output, passandole i necessari parametri,
// prelevati dalla parte variabile del request header. Restituisce E_OK, definita in
// BZDD.H per segnalare la fine dell'operazione.

int output(void)
{
    extern RequestHeaderFP RHptr;                  // per accedere al request header

    colortext(RHptr->cp.oReq.itemCnt,RHptr->cp.oReq.trAddr);
    return(E_OK);
}

// funzione per la gestione del servizio Write With Verify (9). Come si vede non fa
// altro che chiamare la output, quindi non vi e' alcuna verifica. E' qui solamente
// a scopo dimostrativo. Il prototipo e' identico a quello dichiarato in BZDD.H.

int outputVerify(void)
{
    return(output());
}

// funzione per la gestione del servizio Generic IOCTL Request (19). Il prototipo
// e' identico a quello dichiarati in BZDD.H. Essa consente di modificare
// l'attributo video usato dalla output(), o meglio, dalla colortext(), o,
// semplicemente, di conoscere quello attuale, memorizzato nella variabile globale
// attrib. La generic IOCTL Request e' attivata dalle applicazioni mediante
// l'int 21h, servizio 44h, subfunzione 0Ch (character device driver) o 0Dh (block
// device driver). Vedere pagina 456 per un esempio. Per entrambe le funzioni
// IOCTL_GETATTR e IOCTL_SETATTR il formato del campo packet (parte variabile del
// request header) e' molto semplice: il primo byte contiene l'attributo video sia
// in ingresso (se comunicato dall'applicazione al driver) che in uscita (comunicato
// dal driver all'applicazione).

int genericIOCTL(void)             // buffer dati "packet": 1 byte = nuovo attributo
{
    extern RequestHeaderFP RHptr;                  // per accedere al request header
    extern BYTE attrib;
    register oldAttr;

    if(RHptr->cp.gIReq.category != IOCTL_CATEGORY)
        return(unSupported());

// Se il campo category della parte variabile del request header contiene
// IOCTL_CATEGORY, viene analizzato il contenuto del campo function.

    switch(RHptr->cp.gIReq.function) {
        case IOCTL_GETATTR:

// e' richiesto di comunicare l'attuale valore dell'attributo per il video

            *(BYTE far *)(RHptr->cp.gIReq.packet) = attrib;
            break;
        case IOCTL_SETATTR:
```

```
// e' richiesto di sostituire l'attuale valore dell'attributo per il video con
// quello specificato dall'applicazione; inoltre viene comunicato il vecchio
// valore.

            oldAttr = (WORD)attrib;
            attrib = *(BYTE far *)(RHptr->cp.gIReq.packet);
            *(BYTE far *)(RHptr->cp.gIReq.packet) = (BYTE)oldAttr;
            break;
        default:

// non e' supportata alcuna altra funzione

            return(unSupported());
    }
    return(E_OK);
}

void colortext(WORD count,BYTE far *buffer)
{
    extern BYTE attrib;

    _BH = DEFAULT_PAGE;
    _AH = 3;
    geninterrupt(0x10);    // in uscita dall'interrupt: DH,DL = riga,col del cursore
    _BL = attrib;
    _CX = count;

// BP puo' essere modificata solo dopo avere terminato di referenziare
// tutte le variabili allocate nello stack (in questo caso i parametri
// count e buffer); solo a questo punto percio' e'possibile settare ES:BP
// con l'indirizzo del buffer di bytes da scrivere a video.

    asm push es;
    asm push bp;
    _ES = FP_SEG(buffer);
    _BP = FP_OFF(buffer);

// L'uso degli pseudoregistri puo' produrre codice assembly in maniera non
// sempre trasparente (vedere pag. 167). Per questo motivo AX e' caricato
// solo dopo avere modificato ES, in quanto la sequenza sopra potrebbe
// generare una istruzione LES BP (nel qual caso non vi sarebbe alcun
// problema), oppure potrebbe causare il caricamento del segmento di buffer
// in AX seguito da una PUSH AX e una POP ES: e' evidente che in questo caso
// il valore di AX sarebbe perso.

    _AH = 0x13;
    _AL = 1;               // aggiorna cursore e usa BL come attributo (pag. 317)
    geninterrupt(0x10);    // scrive CX bytes da ES:BP con attributo BL
    asm pop bp;
    asm pop es;
}

// init() inizializza il driver, valorizzando la variabile globale attrib. Le altre
// operazioni sono semplicemente la visualizzazione di alcuni dati

int init(int argc,char **argv)
{
    extern DevDrvHeader DrvHdr;              // per accedere al device driver header
    extern BYTE attrib;
    register i;
    char logicalName[9];

// se vi e' un parametro sulla command line, si assume che esso sia il valore
```

```
// iniziale di attrib. Se la sua trasformazione da stringa in intero con atoi()
// fornisce 0, o non c'e' alcun parametro, attrib e' inizializzata con il valore
// di default DEFAULT_ATTR, cioe' 7, cioe' testo bianco su fondo nero.

    if(argc == 2) {
        if(!(attrib = atoi(argv[1])))
            attrib = DEFAULT_ATTR;
    }
    else
        attrib = DEFAULT_ATTR;

// visualizza il segmento di caricamento del driver e la versione di DOS

    printf("\nDevice Driver di prova a %04X:0000. DOS %d.%d.\n",
                                                _baseseg,_osmajor,_osminor);

// copia il nome logico del device dal device driver header ad un buffer locale

    for(i = 0; i < 9; i++)
        if((logicalName[i] = DrvHdr.ln.cname[i]) == BLANK)
            break;

// trasformazione in stringa ASCIIZ

    logicalName[i] = NULL;

// visualizza valore iniziale di attrib e il nome logico del device

    printf("Attributo testo device %s : %d\n\n",logicalName,attrib);

// richiede di lasciare residente in memoria tutto il codice/dati del driver

    setResCodeEnd(_endOfDrvr);
    return(E_OK);
}
```

Vale la pena di soffermarsi sul servizio 13h dell'int 10h (vedere pag. 317), che implementa la funzionalità di output: esso, richiedendo che l'indirizzo della stringa da visualizzare sia caricato in `ES:BP`, introduce alcune difficoltà nella realizzazione della funzione C `colortext()`. Infatti, il registro `BP` è utilizzato dal compilatore C per generare tutti i riferimenti a parametri attuali e variabili locali, cioè ai dati memorizzati nello stack (vedere pag. 158): quando se ne modifichi il valore, come è necessario fare in questo caso, diviene impossibile accedere ai parametri e alle variabili automatiche della funzione (eccetto le variabili `register`) fino a quando il valore originale di `BP` non sia ripristinato. Come si vede, `colortext()` salva `BP` sullo stack con una istruzione `PUSH` e lo ripristina con una istruzione `POP`: ciò è possibile in quanto dette istruzioni referenziano lo stack mediante il registro `SP`. L'implementazione di una funzione dedicata (la `colortext()`), che riceve i dati della parte variabile del request header come parametri, si è rivelata preferibile all'inserimento dell'algoritmo nella `output()`, in quanto l'accesso ai campi di una struttura è effettuato, generalmente, mediante i registri `ES` e `BX`: ciò avrebbe reso più difficoltoso l'utilizzo degli pseudoregistri (vedere pag. 167).

L'uso della macro `geninterrupt()` non interessa le librerie C (vedere pag. 169): il driver è pertanto realizzato in modo da rendere possibile il troncamento della parte residente all'indirizzo della `init()`. Perché ciò sia possibile è ancora necessario definire la variabile `attrib` mediante una funzione jolly (e non come vera variabile globale) e implementare nel sorgente C tutte le funzioni di servizio prima della stessa `init()`, come nell'esempio che segue:

```
int input(void)
{
    return(unSupported());
}
```

Anche il riferimento a `unSupported()`, come nel caso di `geninterrupt()`, non interessa le librerie (`unSupported()` è definita nel toolkit startup module).

Segue il listato del batch file utilizzato per la generazione di `TESTDRV.SYS`:

```
@echo off
REM
REM *** generazione di testdrv.obj
REM
bcc -c -mt testdrv
if errorlevel 1 goto error
REM
REM *** generazione di testdrv.sys
REM
tlink -c -t ddheader testdrv,testdrv.sys,,bzdd cs
if errorlevel 1 goto error
REM
REM *** attribute word per character device driver con generic IOCTL supportato;
REM *** nome logico: "ZETA"
REM
drvset -b1000000001000000 -nzeta testdrv.sys < yes.txt
if errorlevel 1 goto error
REM
REM *** eliminazione file inutili
REM
del testdrv.obj
del testdrv.map
goto end
REM
REM *** visualizza messaggio in caso di errore
REM
:error
echo TESTDRV.SYS non generato!
REM
REM *** fine batch job
REM
:end
```

Si noti che l'opzione `-b` richiede a `DRVSET` di modificare la device attribute word del driver in modo che il DOS lo riconosca come un character device driver (bit 15) in grado di supportare la funzionalità di generic IOCTL request (bit 6).

TESTDRV può essere installato inserendo in `CONFIG.SYS` una riga analoga alla seguente:

```
DEVICE=C:\TESTDRV.SYS 23
```

ove il parametro 23 rappresenta l'attributo video iniziale (nell'esempio testo bianco su fondo blu, ma si può, ovviamente, scegliere qualsiasi combinazione di colori).

Dopo il bootstrap è sufficiente redirigere lo standard output al device `ZETA` per vedere il nostro driver in azione: il comando

```
type c:\config.sys > zeta
```

visualizza il contenuto del file `CONFIG.SYS` in caratteri bianchi su fondo blu. Modificando il parametro sulla command line del driver ed effettuando un nuovo bootstrap è possibile sperimentare altre combinazioni di colori (ad esempio 77 produce caratteri magenta su fondo rosso[376]).

[376] I numeri da 0 a 7 rappresentano, rispettivamente: nero, blu, verde, azzurro, rosso, viola, marrone e bianco. Al codice di ogni colore si può sommare 8: nel caso del testo si ottiene il medesimo colore, con effetto alta intensità (il

E la generic IOCTL request? L'implementazione di `genericIOCTL()` consente di indagare o modificare al volo l'attributo usato per il device `ZETA`, senza che vi sia necessità di un reset del computer. L'interfaccia DOS è rappresentata dall'int 21h, servizio 44h, subfunzioni 0Ch (character device driver) e 0Dh (block device driver).

INT 21H, SERV. 44H, SUBF. 0CH E 0DH: GENERIC IOCTL REQUEST

Input	AH	44h
	AL	0Ch (character device driver) 0Dh (block device driver)
	CH	Category Code
	CL	Function Code
	DS:DX	indirizzo del buffer dati (packet)
Output	AX	codice di errore se CarryFlag = 1. Se CarryFlag = 0, la chiamata ha avuto successo.
Note		A partire dalla versione 3.3 del DOS sono state adottate alcune convenzioni circa Category Code e Function Code, non tutte documentate ufficialmente.

Il programma `DEVIOCTL`, listato di seguito, consente di pilotare il driver `TESTDEV.SYS` mediante l'invio di una generic IOCTL request al DOS, che, a sua volta (e in modo del tutto trasparente all'applicazione) la trasmette al device driver e riceve da questo il risultato, poi trasferito (sempre in modo trasparente) all'applicazione[377].

```
/***************************************************************

    DEVIOCTL.C - Barninga Z! - 1994

    Applicazione per test funzionalita' Generic IOCTL Request
    nel driver TESTDEV.SYS.

    Lanciato senza parametri richiede l'attributo testo attuale;
    lanciato con un parametro assume che esso sia il nuovo
    attributo testo desiderato e lo passa al driver.

    Compilato con Borland C++ 3.1:
```

marrone appare giallo, il viola appare magenta), mentre nel caso del fondo si ha l'effetto intermittenza sul testo. L'attributo si calcola con la seguente formula:

```
attributo = (colore_fondo * 16) + colore_testo
```

dal che si evidenzia che, ad esempio, `23` si ottiene da `(1*16)+(7)`.

[377] In pratica i device driver, secondo il normale schema di azione, dialogano esclusivamente con il DOS. A sua volta, anche l'applicazione dialoga solo con il DOS, il quale isola e, al tempo stesso, interfaccia le due "controparti".

```
    bcc devioctl.c

****************************************************************/
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <dos.h>
#include <string.h>

#define  HELPSTR            "?"

#define  IOCTL_CATEGORY     0x62
#define  IOCTL_GETATTR      1
#define  IOCTL_SETATTR      2

int openZETA(void);                               // funzione di apertura del device
void requestIOCTL(int argc,char **argv,int handle);// gestione generic IOCTL request

char *help = "\                                                 // stringa di help
Sintassi : DEVIOCTL [nuovo_attrib]\n\
  Senza alcun parametro:     richiede l'attributo corrente;\n\
  Con un parametro numerico: lo utilizza per modificare l'attributo corrente\n\
";

// main() pilota le operazioni, controllando il parametro della command line,
// lanciando le funzioni di apertura device e di invio della IOCTL request e
// chiudendo il device a fine lavoro.

int main(int argc,char **argv)
{
    int handle;

    printf("DEVIOCTL - Prova TESTDRV.SYS (genIOCTL) - Barninga Z! '94; help: %s\n",
                                                                       HELPSTR);
    if((argc > 2) || (!strcmp(argv[1],HELPSTR))) {            // controllo parametri
        puts(help);
        return(1);
    }
    if(!(handle = openZETA())) {                                  // apertura device
        puts("DEVPROV1 (device ZETA) non installato.");
        return(1);
    }
    requestIOCTL(argc,argv,handle)              // invio della generic IOCTL request
    close(handle);                                            // chiusura del device
    return(0);
}

// Un'applicazione puo' scrivere o leggere un character device solo dopo averlo
// aperto, utilizzando il nome logico come un vero e proprio nome di file. La
// funzione openZETA() apre il device avente nome logico ZETA mediante la funzione di
// libreria C open(); se l'operazione ha successo utilizza l'int 21h, servizio 44h,
// subfunzione 00h per assicurarsi di avere aperto un device e non un file: se il
// carry flag e' 0 e il bit 7 di DX e' 1, allora e' proprio un device driver.
// Infatti il carry flag a 1 indica un errore, mentre il bit 7 di DX a 0 indica che
// si tratta di un file (TESTDRV.SYS non e' installato ed esiste un file "ZETA"
// nella directory corrente del drive di default).

int openZETA(void)
{
    int handle;
    struct REGPACK r;
```

```
    if((handle = open("ZETA",O_RDWR)) == -1)
        return(NULL);
    r.r_ax = 0x4400;
    r.r_bx = handle;
    intr(0x21,&r);
    if((!(r.r_flags & 1)) && (r.r_dx & 0x80))
        return(handle);                                  // e' un device driver
    close(handle);                                                // e' un file
    return(NULL);
}

// La generic IOCTL request e' inviata invocando l'int 21h attraverso la funzione
// di libreria C intr() (vedere pag. 115 e seguenti). Il request packet si compone
// di un solo byte, usato per comunicare al driver il nuovo attributo video e
// ricevere in risposta quello attuale.

void requestIOCTL(int argc,char **argv,int handle)
{
    struct REGPACK r;
    unsigned char attrib, newAttrib;

    r.r_bx = handle;
    r.r_ax = 0x440C;

// come request packet e' utilizzata la stessa variabile attrib

    r.r_ds = FP_SEG((unsigned char far *)&attrib);
    r.r_dx = FP_OFF((unsigned char far *)&attrib);

// la IOCTL_CATEGORY va in CH

    r.r_cx = IOCTL_CATEGORY << 8;                        // CH = 0x62 (category)
    switch(argc) {
        case 1:

// nessun parametro sulla command line di DEVIOCTL: si richiede al driver
// l'attributo video attualmente utilizzato.

            r.r_cx |= IOCTL_GETATTR;                        // CL = 1 (function)

// l'indirizzo della variabile attrib, che funge da IOCTL packet, e' gia' stato
// caricato in DS:DX prima della switch

            intr(0x21,&r);
            if(r.r_flags & 1) {
                printf("Errore %d\n",r.r_ax);
                break;
            }

// il valore per l'attributo e' stato posto nel primo byte del request packet,
// cioe' direttamente nella variabile attrib

            printf("Attributo corrente = %d\n",attrib);
            break;
        case 2:

// un parametro sulla command line: e' il nuovo attributo da comunicare al driver

            r.r_cx |= IOCTL_SETATTR;                        // CL = 2 (function)
            attrib = newAttrib = (unsigned char)atoi(argv[1]);

// l'indirizzo della variabile attrib, che funge da IOCTL packet, e' gia' stato
// caricato in DS:DX prima della switch
```

```
            intr(0x21,&r);
            if(r.r_flags & 1) {
                printf("Errore %d\n",r.r_ax);
                break;
            }

// il valore per l'attributo e' stato posto nel primo byte del request packet,
// cioe' direttamente nella variabile attrib

            printf("Attributo: corrente = %d; nuovo = %d\n",attrib,newAttrib);
            break;
    }
}
```

DEVIOCTL può essere compilato con il comando

```
bcc devioctl.c
```

che produce l'eseguibile DEVIOCTL.EXE. Questo, se lanciato con un punto interrogativo ("?") quale unico parametro della command line, visualizza un breve testo di aiuto.

Se invocato senza alcun parametro, DEVIOCTL richiede al driver la funzione IOCTL_GETATTR per conoscere l'attributo video attualmente utilizzato per il device ZETA e visualizza la risposta del driver.

Se invocato con un parametro numerico, DEVIOCTL richiede al driver la funzione IOCTL_SETATTR, per forzare il driver a utilizzare quale nuovo attributo video il parametro della command line; il driver risponde restituendo il precedente attributo utilizzato, che viene visualizzato da DEVIOCTL.

Se TESTDRV non è installato (e quindi il device ZETA non è attivo), DEVIOCTL segnala l'errore.

E' sufficiente installare TESTDRV.SYS come sopra descritto e giocherellare con DEVIOCTL per provare l'ebbrezza di pilotare direttamente il device driver.

LINGUAGGIO C E PORTABILITÀ

Una caratteristica di rilievo del linguaggio C consiste nella portabilità. In generale, si dice portabile un linguaggio che consente di scrivere programmi in grado di funzionare correttamente su piattaforme hardware diverse e sotto differenti sistemi operativi, richiedendo semplicemente la ricompilazione dei sorgenti nel nuovo ambiente (e dunque, implicitamente, con una differente implementazione del compilatore).

Tutto ciò è reso possibile, nel caso del C, dalla standardizzazione del medesimo operata dall'ANSI e dal fatto che si tratta di un linguaggio basato in massima parte su routine (funzioni) implementate in librerie esterne al compilatore, e dunque sempre disponibili con caratteristiche coerenti a quelle dei differenti ambienti in cui il compilatore deve operare.

Tali caratteristiche non sono però, da sole, sufficienti a rendere portabile qualsiasi programma C: molto dipende dalla cura spesa nella realizzazione del medesimo nonché, in ultima analisi, dagli scopi che il programmatore si prefigge. In molti casi è impossibile, o sostanzialmente inutile, eliminare quelle caratteristiche del programma che lo rendono più o meno dipendente dallo hardware, dal compilatore e dal sistema operativo, e ciò soprattutto quando si desideri controllare e sfruttare a fondo le prestazioni dell'ambiente in cui il programma deve operare[378].

DIPENDENZE DALLO HARDWARE

Un programma può risultare hardware-dipendente per molte cause, dalle più scontate a quelle di più difficile individuazione.

Esempio di banalità mostruosa: un programma che assuma aprioristicamente la presenza di un disco rigido non è portabile (fatta salva la possibilità di modificare il sorgente) su macchine che non ne siano dotate.

E ancora: l'accesso diretto al buffer video è un ottimo metodo per rendere molto efficienti le operazioni di output, ma comporta la necessità di conoscerne l'indirizzo fisico, che può variare a seconda dello hardware installato.

Più sottili considerazioni si possono fare sulle relazioni intercorrenti tra i tipi di dati gestiti dal programma e il microprocessore installato sulla macchina. Il tipo di dato forse più "gettonato" nei programmi C è l'integer, e proprio l'integer può essere fonte di fastidiosi grattacapi. Il C consente tre modi di dichiarare integer una variabile:

```
short s;
long l;
int i;
```

Va osservato che non è possibile specificarne la dimensione in bit; il compilatore garantisce soltanto che la variabile dichiarata short integer ha dimensione minore o uguale a quella `long`, mentre la variabile dichiarata semplicemente `int` viene gestita in modo da ottimizzarne la manipolazione da parte del processore. Se la compilazione avviene su una macchina a 16 bit essa risulta equivalente a quella `short`, ma la compilazione su macchine a 32 bit la rende equivalente a quella `long`. Non è difficile immaginare i problemi che potrebbero manifestarsi portando ad una macchina a 16 bit un programma compilato su una a 32 bit. E' dunque opportuno utilizzare (quando le dimensioni in bit o byte delle

---

[378] In effetti questo è proprio lo scopo, più o meno manifesto, di gran parte degli esempi presentati nel testo: dal loro esame appare evidente come spesso la portabilità sia stata sacrificata, a fronte di altri vantaggi.

variabili assumano rilevanza) l'operatore `sizeof()` (vedere pag. 68) e le costanti manifeste definite in base allo standard ANSI negli header file `LIMITS.H` e `FLOAT.H`.

In `LIMITS.H` troviamo, ad esempio, `CHAR_BIT` (numero di bit in un `char`); `INT_MIN` (minimo valore per un `int`); `INT_MAX` (massimo valore per un `int`); `UINT_MAX` (massimo valore per un `unsigned int`). A queste si aggiungono minimi e massimi per gli altri tipi: `SCHAR_MIN`, `SCHAR_MAX` e `UCHAR_MAX` per `signed char` e `unsigned char`; `CHAR_MIN` e `CHAR_MAX` per i `char`; `SHRT_MIN`, `SHRT_MAX` e `USHRT_MAX` per gli `short`; `LONG_MIN`, `LONG_MAX` e `ULONG_MAX` per i `long`.

In `STDDEF.H` è definito un gruppo di costanti manifeste il cui simbolo è costituito da un prefisso indicante il tipo di dato (`FLT_` per `float`; `DBL_` per `double`; `LDBL_` per `long double`) e da un suffisso al quale è associato il significato della costante medesima. Presentiamo un elenco dei soli suffissi (per brevità) precisando che l'elenco completo delle costanti manifeste si ottiene unendo ogni prefisso ad ogni suffisso (ad es.: `FLT_DIG`; `DBL_DIG`; `LDBL_DIG`; etc.):

SUFFISSI PER LE COSTANTI MANIFESTE DEFINITE IN `STDDEF.H`

SUFFISSI	SIGNIFICATI	SUFFISSI	SIGNIFICATI
`MAX`	Massimo valore	`MIN`	Minimo valore positivo
`MAX_10_EXP`	Max.esponente decimale	`MIN_10_EXP`	Min.esponente decimale
`MAX_EXP`	Max.esponente binario	`MIN_EXP`	Min.esponente binario
`DIG`	N.cifre di precisione	`MANT_DIG`	N.di bit in mantissa
`EPSILON`	Min.valore di macchina	`RADIX`	Radice dell'esponente

Con riferimento ai tipi in virgola mobile, va ancora ricordato che lo standard ANSI ha definito il tipo `long double`, stabilendo che esso deve essere di dimensione maggiore o uguale al double, senza fissare altri vincoli all'implementazione. Sulle macchine 80x86 esso occupa 80 bit[379], ma su altre piattaforme hardware i compilatori possono gestirlo diversamente.

Le difficoltà non si limitano alla dimensione dei tipi di dato. Appare rilevante, ai nostri fini, anche il modo in cui il processore ne gestisce l'aritmetica. Esistono infatti processori basati sull'aritmetica in complemento a due (come quelli appartenenti alla famiglia Intel 80x86) ed altri basati sull'aritmetica in complemento a uno. Le differenze sono notevoli: questi ultimi, a differenza dei primi, gestiscono due rappresentazioni dello zero (una negativa e una positiva). Inoltre varia la rappresentazione binaria interna dei numeri negativi: ad esempio, `-1` ad otto bit è `11111111` in complemento a due e `11111110` in complemento a uno[380].

Infine, alcune considerazioni sul metodo utilizzato dal processore per ordinare i byte in memoria. Gli Intel 80x86 lavorano nella modalità cosiddetta backwords, cioè a parole rovesciate: ogni coppia di byte (detta word, o parola) è memorizzata in modo che al byte meno significativo corrisponda la locazione di memoria di indirizzo inferiore; quando un dato è formato da due parole (ad esempio un `long` a 32 bit), un analogo criterio è applicato ad ognuna di esse (la word meno significativa precede in

[379] In tal modo esso è particolarmente adatto ad essere elaborato nei registri interni dei coprocessori matematici 80x07. Il `float` e il `double` occupano invece, ripettivamente, 32 e 64 bit.

[380] Quando si abbiano esigenze di portabilità è dunque preferibile evitare il ricorso diffuse prassi quali inizializzare a `-1` un `char` o `int` per utilizzarlo come maschera avente tutti i bit a `1`.

memoria quella più significativa). La conseguenza è che i dati sono memorizzati, in sostanza, a rovescio. Altri processori si comportano in modo differente, e ciò può rendere problematico portare da una macchina all'altra programmi che assumano come scontata una certa modalità di memorizzazione dei byte[381].

DIPENDENZE DAI COMPILATORI

Anche le differenze esistenti tra le molteplici implementazioni di compilatori hanno rilevanti riflessi sulla portabilità. E' del tutto palese che costrutti sintattici ammessi da un compilatore ma non rientranti negli standard (ancora una volta il punto di riferimento è lo standard ANSI) possono non essere ammessi da altri, con ovvie conseguenze. Un macroscopico esempio è rappresentato dallo inline assembly, cioè dalle istruzioni di linguaggio assembler direttamente inserite nel codice C.

Talvolta, problemi di portabilità possono sorgere tra successive release della medesima implementazione di compilatore: la famigerata opzione `-k-` insegna (pag. 173 e seguenti).

Il numero dei caratteri significativi nei simboli (nomi di variabili, di funzioni, di etichette) non è fissato da alcuno standard. Ad esempio, alcuni compilatori sono in grado di distinguere tra loro nomi (ipotetici) di funzioni quali `ConvertToUpper()` e `ConvertToLower()`, mentre altri non lo sono. Inoltre, nonostante il C sia un linguaggio case sensitive (che distingue, cioè, le lettere minuscole da quelle maiuscole), e dunque lo siano tutti i compilatori, possono non esserlo alcuni linker (quantomeno per default): pericoloso, perciò, includere nei sorgenti simboli che differiscono tra loro solo per maiuscole e minuscole (es.: `DOSversion` e `DosVersion`).

Anche in tema di compilatori vi è spazio, comunque, per considerazioni più particolari. Il C riconosce gli operatori unari di incremento `++` e decremento `--` (vedere pag. 64), che possono essere anteposti o, in alternativa, posposti alla variabile che si intende incrementare (decrementare): vale la regola generale che quando esso precede la variabile, questa è incrementata (decrementata) prima di valutare l'espressione di cui fa parte; quando esso la segue, l'incremento (il decremento) avviene dopo la valutazione dell'intera espressione. Consideriamo, però, le due seguenti chiamate a funzione:

```
funz(++a);
funz(a++);
```

Contrariamente a quanto si potrebbe supporre, non è garantito che nella prima la variabile a sia incrementata prima di passarne il valore alla funzione e, viceversa, nella seconda essa sia incrementata successivamente. Lo standard del linguaggio non prevede, al riguardo, vincoli particolari per

---

[381] Vediamo un esempietto. Su una macchina equipaggiata con un Intel 80x86 si può validamente utilizzare il costrutto seguente:

```
union farptr {
    void far *pointer;
    struct ptrparts {
        unsigned offset;
        unsigned segment;
    } ptr;
} point;
```

Esso consente di referenziare un puntatore `far` (`point.pointer`) oppure la sua word segmento (`point.ptr.segment`), oppure ancora la word offset (`point.ptr.offset`) semplicemente utilizzando l'opportuno elemento della `union`: su macchine equipaggiate con il processore Z8000 tale costrutto maniene la propria validità sintattica, ma `point.ptr.offset` contiene il segmento del puntatore e, viceversa, `point.ptr.segment` ne contiene l'offset, in quanto lo Z8000 non utilizza la tecnica backwords.

l'implementazione, coerentemente con la generale libertà che ogni compilatore C può concedersi nella valutazione delle espressioni[382].

A tutti i programmatori C è noto, inoltre, il problema degli effetti collaterali (side-effect) che possono verificarsi quando una variabile in autoincremento (o decremento) costituisce il parametro di una macro, piuttosto che di una funzione. Al riguardo precisiamo che, a seconda dell'implementazione del compilatore, una funzione C può benissimo essere in realtà... una macro: è meglio dunque documentarsi a fondo[383], al minimo curiosando nei file `.H`.

Ancora sulle funzioni: le implementazioni recenti di compilatori ammettono (ed è lo stile di programmazione consigliato) che nelle dichiarazioni di funzione siano specificati tipo e nome dei parametri formali:

```
int funzione(int *ptr,long parm);
```

Ciò consente controlli precisi sul tipo dei parametri attuali e limita le possibilità di errore. I compilatori obsoleti individuano in tali dichiarazioni un errore di sintassi e precisano che i parametri possono essere specificati solo nelle definizioni di funzione (non nelle dichiarazioni).

E' ora il momento di affrontare i puntatori. In C è del tutto lecito e normale calcolare la differenza tra due puntatori: ci si potrebbe aspettare che il risultato sia un integer (o unsigned integer). In realtà esso può essere un `long`: dipende, ancora una volta, dall'implementazione del compilatore. Lo standard ANSI definisce il tipo `ptrdiff_t`[384], il quale garantisce la coerenza della dimensione delle variabili ad esso appartenenti con il comportamento del compilatore nel calcolo di differenze tra puntatori.

Aggiungiamo, per rimanere in tema, che il puntatore nullo `NULL` non è necessariamente uno zero binario in tutte le implementazioni.

Per quanto riguarda i tipi di dato, occorre prestare attenzione al trattamento riservato dal compilatore ai `char`, i quali possono essere considerati `signed` o `unsigned char` per default. Ignorare le convenzioni in base alle quali il compilatore utilizzato si comporta può essere fonte di bug inaspettati e molto difficili da scovare: ci si ricordi che se, per esempio, i `char` sono trattati come segnati,

---

[382] Ovviamente ogni compilatore C è tenuto al rispetto delle priorità algebriche; tuttavia la valutazione di un'espressione può procedere, a parità di priorità algebrica tra gli elementi valutati, indifferentemente da sinistra a destra o viceversa (per alcuni operatori non sono definiti vincoli di associatività: vedere pag. 61), a tutto vantaggio dell'efficienza complessiva del programma.

[383] Esempio, tratto dalla realtà. In tutte le implementazioni standard, `toupper(char c)` converte in maiuscolo, se minuscolo, il carattere contenuto in `c`. Se la libreria utilizzata implementa `toupper()` come funzione, la riga di codice:

```
    a = toupper(++c);
```

può dare luogo alle ambiguità concernenti il momento in cui `c` è incrementata. Ma portando il sorgente ad un compilatore che implementi `toupper()` come macro:

```
#define toupper(c)      (islower(c) ? c - 'a' + 'A' : c)
```

la riga di codice viene espansa dal preprocessore nel modo seguente:

```
    a = (islower(++c) ? ++c - 'a' + 'A' : ++c);
```

con le immaginabili conseguenze sui valori finali di `c` e di `a`.

[384] In `STDDEF.H`.

un'espressione del tipo (a < 0x90), dove a è dichiarata semplicemente char, è sempre vera; analogamente, se i char sono considerati privi di segno, (a < 0x00) è sempre e comunque falsa. Le costanti esadecimali, infatti, sono sempre considerate prive di segno. Altre "stranezze" si verificano nel caso di assegnamento del valore (negativo) di una variabile char ad una variabile int, come risulta dal seguente esempio:

```
....
char c;
int i;
....
c = -1;
i = c;
....
```

Quanto vale i? Se il compilatore considera i char grandezze senza segno, allora i vale 255; in caso contrario assume il valore -1.

DIPENDENZE DAL SISTEMA OPERATIVO

Il sistema operativo mette a disposizione del programmatore un insieme di servizi che possono costituire una comoda interfaccia tra il software applicativo e il ROM BIOS o lo hardware. Sistemi operativi progettati per fornire ambienti differenti sono (è ovvio) interinsecamente diversi; ciononostante è spesso possibile portare programmi dall'uno all'altro senza particolari difficoltà. E' il caso, ad esempio, di Unix e DOS, quando si conoscano entrambi i sistemi appena un poco in profondità e si rinunci ad ottimizzazioni a basso livello del codice. Al riguardo intendiamo ricordare solo alcune delle differenze più notevoli tra DOS e Unix, quale spunto per meditazioni più approfondite.

In primo luogo Unix è, contrariamente al DOS, un sistema multiuser/multitask; un programma scritto sotto DOS con "ambizioni" di portabilità deve essere in grado di gestire la condivisione delle risorse con altri processi.

Un'altra differenza di rilievo consiste nell'assenza, in Unix, del concetto di volume (cioè di disco logico): pertanto un pathname, sotto Unix, non può includere un identificativo di drive. Vi sono poi sistemi operativi (CP/M, antesignano dello stesso DOS) che non gestiscono file systems gerarchici (in altre parole: non consentono l'uso delle directory). Inoltre la backslash ('\') che in DOS separa i nomi di directory e identifica la root è una slash ('/') in Unix, il quale considera il punto ('.'), nei nomi di file, alla stregua di un carattere qualsiasi, mentre in DOS esso ha la funzione di separare nome ed estensione. In ambiente DOS, infine, i nomi di file possono contare un massimo di undici caratteri (otto per il nome e tre per l'estensione); Unix ammette fino a quattordici caratteri; OS/2 (se installato con HPFS[385]) fino a 256.

Unix gestisce le unità periferiche come device[386]: tale caratteristica è stata in parte ripresa dal DOS ed il linguaggio C consente di sfruttarla proficuamente attraverso le funzioni basate su stream (pag. 116). Particolarmente interessanti sono gli stream standard, resi disponibili dal sistema, ed usati da funzioni e macro come printf(), puts(), gets(), etc.: si tratta di stdin (standard input), stdout (standard output), stderr (standard error), stdaux (standard auxiliary) e stdprn (standard

[385] High Performance File System.

[386] Dispositivi hardware pilotati da apposite interfaccia software (device driver: pag. 353), che comunicano con le periferiche attraverso un protocollo dedicato e con il sistema mediante flussi (stream) di dati; i programmi applicativi risultano così, in larga misura, indipendenti dal dispositivo fisico (stampante, disco, console) dal quale provengono (o al quale sono diretti) i dati.

printer). La portabilità tra Unix e DOS del codice che ne fa uso è quasi totale[387], ma vi sono sistemi operativi che gestiscono le periferiche in modo assai differente.

Approfondimenti circa problemi di portabilità tra Unix e DOS di sorgenti che implementano controllo e gestione di processi si trovano a pag. 136.

Se, da una parte, è prevedibile incontrare problemi di portabilità tra sistemi di differente concezione tecnica, dall'altra sarebbe un errore ritenere che lo scrivere codice portabile tra sistemi fortemente analoghi sia privo di ogni difficoltà: si pensi, ad esempio, alle differenze esistenti tra versioni successive del DOS. Il codice che utilizzi servizi DOS non presenti in tutte le versioni non può dirsi completamente portabile neppure nell'ambito del medesimo ambiente operativo. La realizzazione di codice portabile tra ogni release di DOS implica la rinuncia alle funzionalità introdotte via via con le nuove versioni[388]: si tratta, con ogni probabilità, di un prezzo troppo alto e spetta pertanto al programmatore scegliere un opportuno compromesso tra il grado di portabilità da un lato e il livello di efficienza, unitamente al contenuto innovativo del codice, dall'altro[389]. Le funzioni implementate dalle librerie standard dei recenti compilatori C si basano su servizi DOS disponibili a partire dalla versione 2.0 o 3.0.

Va aggiunto che esistono versioni di DOS modificate da produttori di hardware per migliorarne la compatibilità con le macchine da essi commercializzate; possono così riscontrarsi diversità non solo tra release successive, ma anche tra differenti "marchi" nell'ambito della medesima release. Quasi sempre si tratta, in questi casi, di differenze nelle modalità con le quali il DOS interagisce con BIOS e hardware; pertanto difficilmente esse hanno reale influenza sulla portabilità del codice, eccetto i casi in cui questo incorpori o utilizzi servizi implementati a basso livello. Con un approccio forse un po' grossolano si può inoltre osservare che il DOS è costituito da un insieme di routine, ciascuna in grado di svolgere un compito piuttosto elementare (aprire un file, visualizzare un carattere...). Alcune di esse formano l'insieme dei servizi resi disponibili da quella particolare release: sono, pertanto descritte nella documentazione tecnica e la loro permanenza in future versioni è garantita. Altre, realizzate quali routine di supporto alle precedenti, sono riservate ad uso "interno" da parte del sistema operativo stesso: i manuali tecnici non vi fanno cenno e non è possibile contare sulla loro presenza nelle versioni future[390].

[387] "Quasi" significa che comunque vi sono alcune differenze tra il comportamento dei due sistemi operativi in questione. Ad esempio, Unix consente di redirigere lo standard error, al contrario del DOS.

[388] Ogni versione di DOS affianca i nuovi servizi a quelli preesistenti, anche quando questi ultimi siano resi obsoleti dai primi: ne deriva una sostanziale compatibilità verso il basso.

[389] Inoltre è lecito ipotizzare che le nuove versioni del DOS abbiano, nel tempo, rimpiazzato quelle originariamente installate.

[390] Inoltre, qualora esse siano presenti in altre versioni del DOS, nulla assicura che il contenuto dei registri della CPU o la struttura dei dati gestiti rimangano invariati. E' un vero peccato: di solito si rivelano utilissime.

DI TUTTO... DI PIÙ

Il presente capitolo presenta una raccolta di esempi. Si tratta di idee e spunti che non hanno, per le ragioni più svariate, trovato una sistemazione logica altrove nel testo.

DUE FILE SONO IL MEDESIMO FILE?

La domanda è formulata in modo fuorviante. Il problema che intendiamo affrontare è, in realtà, come capire se due pathname, apparentemente differenti, si riferiscono al medesimo file: la questione non è banale, in quanto occorre tenere nella dovuta considerazione i default assunti di volta in volta dal DOS. Vediamo un semplice esempio: "\PROGS\BIN\DOCS\LEGGIMI.TXT" e "C:LEGGIMI.TXT" sono il medesimo file? La risposta è SI, se il drive e la directory di default sono, rispettivamente, "C:" e "\PROGS\BIN\DOCS"; NO altrimenti.

Una prima soluzione del problema consiste nell'utilizzo delle opportune chiamate al DOS per conoscere i defaults attuali: ad esempio, i servizi 19h e 47h dell'int 21h consentono di conoscere drive e, rispettivamente, directory di default.

Una seconda possibilità è affidarsi al sistema operativo per ricavare dalle stringhe di partenza i due pathname completi, ad esempio mediante il servizio 60h dell'int 21h (disponibile a partire dalla versione 3.0 del DOS), non documentata ufficialmente, ma della quale riportiamo, per completezza, la descrizione:

INT 21H, SERV. 60H: RISOLVE UN PATH IN PATHNAME COMPLETO

Input	AH	60h
	DS:SI	indirizzo della stringa contenente il pathname da risolvere
	ES:DI	indirizzo del buffer (80 byte) per il pathname completo
Output	Carry	1 = errore AX = codice dell'errore 02h = stringa non valida 03h = drive non valido 0 = altrimenti AL = 00h AH = '\' se il path in input si riferisce ad un file o a una subdirectory della root; ultima lettera del path in input se esso si riferisce alla directory di default; 00h altrimenti
Note		Il buffer puntato da ES:DI è riempito con il pathname completo (D:\PATH\FILE.EXT); il pathname in input è valido anche se inesistente e può contenere '.' e "..". Tutte le lettere sono convertite in maiuscole. Le '/' sono convertite in '\'.

Entrambi i metodi descritti rivelano un approccio di tipo "umano" al problema; dal momento che il DOS gestisce i file mediante la File Allocation Table e che un cluster di un disco non può

appartenere contemporaneamente a due file[391], possiamo individuare un metodo più diretto: due pathname referenziano il medesimo file se il DOS, mediante la propria routine `FINDFIRST` di scansione della directory (int 21h, servizio 4Eh), evidenzia per entrambi medesimo drive e medesimo cluster iniziale.

La routine che presentiamo si basa sulla funzione di libreria `findfirst()`, la quale utilizza una `struct ffblk` (definita in `DIR.H`) per memorizzare quanto restituito dalla `FINDFIRST` del DOS: `isfsame()` può pertanto essere definita una funzione di alto livello.

```
/********************

    BARNINGA_Z! - 1990

    ISFSAME.C - isfsame()

    int cdecl isfsame(char *fname1,char *fname2);
    char *fname1, *fname2; puntatori ai due pathnames
    Restituisce: !0 se i due pathnames sono il medesimo
                  0 altrimenti

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx isfsame.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dir.h>
#include <dos.h>          /* richiesto solo per FA_LABEL e FA_DIREC */

#define  RESFIELD_LEN        21

int cdecl isfsame(char *fname1,char *fname2)
{
    register int i = 0;
    struct ffblk ff1, ff2;

    if(!findfirst(fname1,&ff1,~(FA_LABEL | FA_DIREC)))
        if(!findfirst(fname2,&ff2,~(FA_LABEL | FA_DIREC)))
            for(; i < RESFIELD_LEN; i++)
                if(ff1.ff_reserved[i] != ff2.ff_reserved[i])
                    break;
    return(i == RESFIELD_LEN);
}
```

La `findfirst()` è invocata due volte, una per ogni pathname: in entrambi i casi, l'attributo di ricerca (il terzo parametro) determina l'esclusione delle etichette di volume e delle directory[392]; i due pathname si riferiscono al medesimo file se i campi `ff_reserved` delle due strutture sono identici. Di `findfirst()` si parla anche a pag. 102.

E' necessario precisare che il contenuto del campo `ff_reserved` non è ufficialmente documentato: tuttavia in esso si trovano le informazioni necessarie al nostro scopo (drive e cluster iniziale

[391] Salvo il caso di anomalie nella gestione della FAT: chi non ha mai usato alemno una volta la utility `CHKDSK`?.

[392] La ricerca delle `FINDFIRST/FINDNEXT` è filtrata confrontando gli attributi dei file con il template fornito alla chiamata: se un bit del template è `0` e il corrispondente bit dell'attribute byte del file è `1`, questo viene escluso dalla ricerca.

del file) ed altre che vengono utilizzate nelle eventuali chiamate alla `findnext()`, la quale, a sua volta, si fonda sull'int 21h, funzione 4Fh (la `FINDNEXT` del DOS). La posizione di tali informazioni all'interno del campo è differente, a seconda della versione del DOS; tuttavia, la `isfsame()` sopravvive sulla constatazione che, se i due pathname si riferiscono al medesimo file, l'algoritmo si risolve nel chiamare due volte la `findfirst()` per quel file, e ciò, ovviamente, sotto la stessa versione di DOS.

DOVE MI TROVO?

Perdersi in un groviglio di drive e directory non è una bella esperienza. Per questo è opportuno che un programma possa sapere, se necessario, qual è l'attuale directory di lavoro. La libreria del C Borland mette a disposizione due funzioni che forniscono informazioni al proposito: `getcurdir()`, che permette di conoscere la directory corrente per un dato drive (compreso quello di default), e `getcwd()`, che consente di conoscere la directory corrente del drive di default. Come si può facilmente constatare, esse sono piuttosto simili e inoltre, dato l'algoritmo utilizzato[393], vengono entrambe influenzate dai comandi `SUBST` e `JOIN` del DOS. Ad esempio, dopo avere assegnato al path `C:\LAVORO\GRAFICA` l'identificativo di drive `K:` mediante `SUBST`, possiamo referenziare la directory `GRAFICA` sia come sottodirectory di `C:\LAVORO`, sia come root del drive fittizio `K:` e posizionarci in essa con il comando `CHDIR` su `C:` oppure cambiando in `K:` il drive di default. In questo caso `getcurdir()` e `getcwd()` indicano che la directory corrente è la root di `K:`. Se il programma necessita conoscere la reale directory di lavoro, ignorando eventuali combinazioni di `SUBST` e `JOIN`, può ricorrere al servizio 60h dell'int 21h (descrizione dettagliata a pag.). Vediamone un esempio di utilizzo:

```
/********************

    BARNINGA_Z! - 1991

    GETRDIR.C - getrealdir()

    int cdecl getrealdir(char *userpath);
    char *userpath; puntatore ad array di caratteri allocato a cura
                    del programmatore e di grandezza sufficiente a
                    contenere il pathname completo della directory
                    corrente
    Restituisce:    0 se tutto o.k.
                    0 se si e' verificato un errore

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -rd -c -mx getrdir.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>
#include <string.h>

#define  MAXPATH  80

int cdecl getrealdir(char *userpath)
{
    int retcode;
    char realpath[MAXPATH];
    struct SREGS sregs;
    union REGS regs;
```

[393] Si basano entrambe sul servizio 47h dell'int 21h.

```
    realpath[0] = '.';
    realpath[1] = (char)0;
    segread(&sregs);
    regs.h.ah = 0x60;
    sregs.ds = sregs.es = sregs.ss;
    regs.x.si = regs.x.di = (unsigned)realpath;
    retcode = intdosx(&regs,&regs,&sregs);
    if(!regs.x.cflag) {
        retcode = 0;
        (void)strcpy(userpath,realpath);
    }
    return(retcode);
}
```

L'array `realpath` funge da buffer sia di input che di output; occorre dunque caricare i registri `DS` e `ES` con il suo indirizzo di segmento, e i registri `SI` e `DI` con il suo offset. In quanto variabile automatica, `realpath` è allocato nello stack: il suo indirizzo di segmento è dunque rappresentato dal registro `SS`; in quanto array, il suo offset è rappresentato dal nome stesso (che è un puntatore all'array). Il valore di `SS` è ricavato mediante la funzione di libreria `segread()`, che copia i valori dei registri di segmento nei campi della struttura `sregs` (di tipo `SREGS`)[394]. `SI` e `DI` sono caricati, attraverso la `union regs` di tipo `REGS`, con l'offset dell'array; il cast ad `unsigned` è necessario in quanto `realpath` è puntatore a `char`, cioè di tipo `char *`. Se la funzione di libreria `intdosx()` non ha riscontrato la restituzione di un errore da parte dell'int 21h, da essa invocato, il campo `.x.cflag` della `union REGS` è nullo[395]: la stringa preparata dal servizio 60h è copiata all'indirizzo passato come parametro a `getrealdir()`, che restituisce `0`. Se `.x.cflag` non è nullo allora `getrealdir()` restituisce il codice di errore riportato dall'int 21h. Si noti che il puntatore `userpath`, parametro della funzione, deve essere allocato a cura della routine che invoca la `getrealdir()` e deve essere in grado di contenere l'intero pathname (la massima lunghezza di un pathname, in DOS, è 80 caratteri compreso il terminatore nullo[396]).

Poche modifiche sono sufficienti per trasformare `getrealdir()` in una funzione in grado di risolvere in pathname completo qualunque nome di file o directory passatole come parametro.

```
/********************

    BARNINGA_Z! - 1991

    RSLVPATH.C - resolvepath()

    int cdecl resolvepath(char *userpath);
    char *userpath; puntatore alla stringa contenente il pathname
                    parziale da risolvere in pathname completo. Lo
                    spazio allocato deve essere sufficiente a
                    contenere quest'ultimo.
    Restituisce:     0 se tutto o.k.
                    !0 se si e' verificato un errore:
                         2 = userpath contiene caratteri non leciti in
                             un pathname
                         3 = drive specificato in userpath non valido
```

[394] Vedere pag. 115 e seguenti.

[395] Esso contiene il valore del `CarryFlag` al ritorno dall'int 21h.

[396] Del resto, `80` è il valore della costante manifesta `MAXPATH`, definita in `DIR.H`.

```
    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -rd -c -mx RSLVPATH.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#include <dos.h>
#include <string.h>

#define  MAXPATH  80

int cdecl resolvepath(char *userpath)
{
    int retcode;
    char realpath[MAXPATH];
    struct SREGS sregs;
    union REGS regs;

    segread(&sregs);
    regs.h.ah = 0x60;
    sregs.es = sregs.ss;
    regs.x.di = (unsigned)realpath;
#if defined(__TINY__) || defined(__SMALL__) || defined(__MEDIUM__)
    regs.x.si = (unsigned)userpath;
#else
    sregs.ds = FP_SEG(userpath);
    regs.x.si = FP_OFF(userpath);
#endif
    retcode = intdosx(&regs,&regs,&sregs);
    if(!regs.x.cflag) {
        retcode = 0;
        (void)strcpy(userpath,realpath);
    }
    return(retcode);
}
```

La differenza sostanziale tra le due funzioni è che `resolvepath()` utilizza `userpath` come buffer di input per il servizio 60h. Esso punta ad un array allocato nel segmento dati; pertanto la sua gestione varia a seconda del modello di memoria prescelto per la compilazione. Se lo heap è limitato a 64Kb (modelli tiny, small e medium), userpath esprime un offset rispetto a `DS`: `segread()` carica dunque `sregs.ds` con il valore effettivamente necessario all'int 21h. Nei modelli compact, large e huge `userpath` è un puntatore `far`: si rende pertanto necessario l'uso delle macro `FP_SEG()` e `FP_OFF()`, definite in DOS.H, che ricavano, rispettivamente, segmento ed offset da puntatori di tipo `far` (pag. 24).

Se `userpath` punta alla stringa `"."`, `resolvepath()` può essere utilizzata in luogo di `getrealdir()`.

Il servizio 60h dell'int 21h non è ufficialmente documentato[397]: ad esempio, non sembra essere implementato nel DR-DOS 5.0, rilasciato alcuni mesi dopo lo sviluppo delle due funzioni descritte. Come accade spesso quando si sfruttano caratteristiche non ufficiali del sistema operativo, è stato indispensabile correre ai ripari, scrivendo una funzione in grado di emulare il servizio 60h.

```
/********************

    BARNINGA_Z! - 1991

    PATHNAME.C - pathname()
```

[397] Guarda caso, i servizi non documentati sono (quasi) sempre molto utili ed interessanti.

```
pathname()          Risolve un pathname in pathname completo

SINTASSI     int cdecl pathname(char *path,char *src,char *badchrs);

PARAMETRI    path           puntatore ad  un'area di  memoria di  almeno
                            MAXPATH (80)  caratteri (MAXPATH  è definita
                            in  fileutil.h)  in  cui  è  costrutio  il
                            pathname completo. La funzione non controlla
                            la   lunghezza   del   buffer;   assume
                            semplicemente che  esso  sia  sufficiente  a
                            contenere tutto il pathname.

             src            puntatore  al  pathname  sorgente.  Se  la
                            stringa è  vuota, in  pcompl viene  posto il
                            pathname della  directory corrente del drive
                            di default, seguita da una backslash.

             badchrs        puntatore  ad  una  stringa  contenente  i
                            caratteri  che  devono  essere  considerati
                            illeciti in  un pathname  oltre ai caratteri
                            da 0x00  a 0x1F. In fileutil.h è definita la
                            stringa BADCHARS  ";,:|><". I  due punti (:)
                            sono comunque  ammessi  nell'indicatore  del
                            drive.

SCOPO        Trasforma un pathname (src) in pathname completo di drive,
             percorso e nome di file, considerando gli attuali defaults
             di sistema.

RESTITUISCE  0         Operazione completata con successo.

             EOF       In caso  di errore. Le variabili globali errno e
                       doserrno contengono  il codice di errore ENOPATH
                       (path non trovato); errno e doserrno  contengono
                       il codice dell'errore.

NOTE         Il path  sorgente può  contenere '*'  e '?'.  Gli '*' sono
             trasformati nell'opportuno  numero di  '?'. Le slashes (/)
             eventualmente presenti sono convertite in backslashes (\).
             Il path  sorgente può  anche  non  esistere,  e  può  fare
             riferimento ad un drive inesistente (nel qual caso il path
             fornito è  assunto quale entry della root). La presenza di
             un carattere  illecito nel  pathname, o di due o più punti
             (.) anche  non consecutivi,  o due  o  più  backslashes  o
             slashes consecutive  determinano errore. Tutti i caratteri
             che seguono  un '*'  e precedono il '.' o la fine del nome
             (se il  '.' è già stato incontrato) sono ignorati. Un nome
             di directory  (o di file) che superi gli 8 caratteri viene
             troncato ad  8. L'estensione  che superi  i 3  caratteri è
             troncata a  3. La funzione interpreta '.' e '..'; tuttavia
             esse devono  (ovviamente)  trovarsi  all'inizio  del  path
             sorgente  o  immediatamente  dopo  i  due  punti  (:)
             dell'indicativo del  drive, onde evitare una condizione di
             errore. Il  pathname costruito  in pcompl  non termina con
             backslash, a meno che si tratti di root o il path in input
             (src) sia una stringa vuota.

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -rd -c -mx PATHNAME.C

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
```

```
#pragma  warn -pia

#include <string.h>
#include <errno.h>
#include <ctype.h>
#include <dir.h>

#define  NAMELEN        (MAXFILE-1)                         /* lunghezza nomi file */
#define  EXTLEN         (MAXEXT-2)                         /* lunghezza estensione */
#define  CURRENT        "."
#define  CURDIR         ".\\"
#define  PARENT         ".."
#define  PARDIR         "..\\"
#define  PARDIR2        "\\.."
#define  ROOTSYM        ":\\"
#define  SLASH          '/'
#define  BSLASH         '\\'
#define  COLON          ':'
#define  POINT          '.'
#define  ASTERISK       '*'
#define  QMARK          '?'
#define  D_FIRST        'A'

#define  islwctl(c)  ((c >= 0 && c < 0x20) ? 1 : 0)          /* macro per chr ctl */
#define  isslash(c)  ((c == SLASH || c == BSLASH) ? 1 : 0)         /* separatore? */

int pascal __IOerror(int dosErr);           /* funzione C per condizione d'errore */


int pathname(char *pcompl,char *src,char *badchrs)
{
    register sx, px, cnt;
    char *ptr, temp[MAXPATH];
    struct {
        unsigned b:1;                            /* il car. precedente e' BSLASH */
        unsigned c:1;                            /* il path in input parte da .\ */
        unsigned p:1;                             /* il car. precedente e' POINT */
        unsigned w:1;                                   /* gia' incontrato QMARK */
    } flag;

    for(cnt = 0; src[cnt]; cnt++)
        temp[cnt] = (src[cnt] == SLASH) ? BSLASH : src[cnt];           /* / --> \ */
    temp[cnt] = NULL;
    if(temp[--cnt] == BSLASH)                       /* il path in input non puo' */
        if(cnt > 0 && strstr(temp,ROOTSYM) != temp+cnt-1)        /* finire in \ a */
            return(__IOerror(ENOPATH));                      /* meno che sia root */
    if(cnt > 0 && temp[1] == COLON) {
        if(!isalpha(*temp))                    /* errore se 1^ car non alfabet. */
            return(__IOerror(ENOPATH));
        *pcompl = toupper(*temp);
        sx = 2;
    }
    else {
        *pcompl = getdisk()+D_FIRST;                          /* drive di default */
        sx = 0;
    }
    pcompl[1] = COLON;
    pcompl[2] = BSLASH;
    px = MAXDRIVE;                                              /* costruito d:\ */
    flag.c = 0;
    if(temp[sx] == BSLASH) {          /* se in temp c'e' BSLASH iniziale o dopo d: */
        ++sx;                    /* il percorso e' completo; BSLASH gia' in pcompl */
        flag.b = 1;
```

```
    }
    else {                                  /* altrimenti bisogna cercare . e .. */
        flag.b = 0;
        cnt = 0;
        if(!strcmp(temp+sx,CURRENT))                              /* temp = "." */
            sx += strlen(CURRENT);
        else {
            if(!strcmp(temp+sx,PARENT)) {                        /* temp = ".." */
                ++cnt;
                sx += strlen(PARENT);
            }
            else {
                ptr = strstr(temp+sx,CURDIR);                      /* c'e' ".\" */
                if(ptr == temp+sx) {                    /* nella posiz. attuale */
                    sx += strlen(CURDIR)-1;
                    flag.c = 1;
                }
                else {
                    ptr = strstr(temp+sx,PARDIR);                 /* c'e' "..\" */
                    if(ptr == temp+sx) {                /* nella posiz. attuale */
                        sx += strlen(PARDIR)-1;
                        cnt = 1;
                    }
                }
                while(ptr = strstr(temp+sx,PARDIR2)) {           /* cerca "\.." */
                    if(ptr == temp+sx) {                     /* seguiti da \ o */
                        sx += strlen(PARDIR);                /* a fine stringa */
                        if(!temp[sx] || (temp[sx] == BSLASH)) {
                            ++cnt;
                            continue;
                        }
                    }
                    return(__IOerror(ENOPATH));       /* "\.." in posiz. errata */
                }
            }
        }
        if(!getcurdir((*pcompl)-D_FIRST+1,pcompl+px))            /* dir di default */
            if((px = strlen(pcompl)) > MAXDRIVE) {
                pcompl[px++] = BSLASH;
                flag.b = 1;
            }
        if(cnt) {
            pcompl[px] = NULL;                /* esclude dal percorso costruito */
            for(++cnt; cnt; cnt--)                 /* le ultime dirs annullate */
                if(!(ptr = strrchr(pcompl,BSLASH)))                /* dai "\.." */
                    return(__IOerror(ENOPATH));         /* errore se "\.." piu' */
                else                                   /* numerosi delle dirs nel */
                    *ptr = NULL;                               /* path costruito */
            px = (unsigned)(ptr-pcompl);
            flag.b = 0;
        }
    }
    for(flag.w = 0, flag.p = 0, cnt = NAMELEN; temp[sx]; ) {               /* copia */
        switch(pcompl[px] = toupper(temp[sx])) {              /* il resto di temp */
            case BSLASH:
                if(flag.c) {                                 /* temp = ".\xxxxxx" */
                    flag.c = 0;
                    if(flag.b)                            /* temp = ".\..\xxxxxx" */
                        --px;
                }
                else
                    if(flag.b || flag.p || flag.w)
                        return(__IOerror(ENOPATH));
                flag.b = 1;
```

```
                cnt = NAMELEN;
                break;
            case POINT:
                if(flag.b || flag.p)
                    return(__IOerror(ENOPATH));
                flag.p = 1;
                cnt = EXTLEN;
                break;
            default:
                if(strchr(badchrs,pcompl[px]) || islwctl(pcompl[px]))
                    return(__IOerror(ENOPATH));
                flag.b = 0;
                if(!cnt) {
                    while(temp[sx] && (temp[sx] != POINT) &&
                                                    !(temp[sx] == BSLASH))
                        ++sx;
                    continue;
                }
                if(pcompl[px] == QMARK)
                    flag.w = 1;
                else
                    if(pcompl[px] == ASTERISK) {            /* risolve asterischi */
                        for(; cnt; cnt--)
                            pcompl[px++] = QMARK;
                        while(temp[sx] && (temp[sx] != POINT)) {
                            if(temp[sx] == BSLASH)
                                return(__IOerror(ENOPATH));
                            ++sx;
                        }
                continue;
                }
                --cnt;
        }
        ++sx;
        ++px;
    }
    pcompl[px] = NULL;
    cnt = strlen(pcompl)-1;
    if((pcompl[cnt] == BSLASH) && strlen(pcompl) != MAXDRIVE && *temp)
        --px;                  /* elimina \ finale a meno che \ sola o temp vuota */
    else
        if(strlen(pcompl) == MAXDRIVE-1)                /* aggiunge \ se path e' */
            pcompl[px++] = BSLASH;            /* d:\ e la \ finale e' stata tolta */
    return(pcompl[px] = NULL);                /* da eliminzaione dirs per "\.." */
}
```

La `pathname()` non è in grado di riconoscere i pathname originali sottostanti ridefinizioni effettuate da `JOIN` e `SUBST`.

Si noti l'uso della funzione (non documentata) di libreria `__IOerror()`, mediante la quale sono valorizzate le variabili globali `errno` e `doserrno`, al fine di rendere la gestione degli errori coerente con lo standard di libreria (per i dettagli su `__IOerror()` vedere pag.).

LA COMMAND LINE

Si intende, per command line, la riga di testo digitata al prompt di sistema: essa contiene un comando DOS (che può essere il nome di un programma), completo degli eventuali parametri da avviamento. Ad esempio, la command line

```
pippo /a /b file0 *.dat
```

lancia il programma PIPPO passandogli quattro parametri, separati tra loro da spazi.

E' noto che il linguaggio C mette a disposizione un metodo semplice e standardizzato per accedere ai parametri della command line: allo scopo è sufficiente dichiarare la funzione main() con due parametri, nell'ordine un intero e un array di stringhe (un array di puntatori a carattere), convenzionalmente chiamati argc e argv (*arguments counter* e *arguments vector*):

```
void main(int argc, char **argv)

    ....
```

La variabile argc contiene il numero di parole, separate da spazi, presenti sulla command line (incluso, dunque, il nome del programma); gli elementi di argv referenziano le parole componenti la command line (argv[0] punta al nome del programma, completo di pathname[398]). Ogni sequenza di caratteri compresa tra due o più spazi (o tabulazioni) è considerata un parametro (se si intende passare al programma una stringa contenente spazi come un unico parametro è sufficiente racchiuderla tra doppi apici, "come questa") e le wildcard non vengono espanse (con riferimento all'esempio sopra riportato, argv[4] in PIPPO contiene "*.dat").

Aggiungendo un terzo parametro a main() è possibile avere a disposizione anche le stringhe che costituiscono l'environment del programma[399]:

```
void main(int argc, char **argv, char **envp)

    ....
```

Come argv, anche envp (environment pointer) è un array di puntatori a stringa, o meglio un array di puntatori a carattere. Vedere anche pag. 105 e seguenti.

Il lavoro di parsing (cioè di scansione) della command line e di valorizzazione di inzializzazione delle stringhe e dei rispettivi puntatori è svolto da due funzioni di libreria, non documentate in quanto implementate per uso interno: nelle librerie del C Borland i loro nomi sono _setargv__() e _setenvp__()[400].

_setargv__() e _setenvp__()

Appare evidente che _setargv__() e _setenvp__() svolgono un compito utile solo se effettivamente il programma ha necessità di conoscere le stringhe che compongono l'environment e, rispettivamente, la command line. Esse vengono tuttavia invocate dal modulo di startup (pag. 105) in ogni caso: è possibile accrescere l'efficienza dei programma che non necessitano di tali informazioni con un semplice stratagemma.

```
#pragma option -k-            /* evita stack frame nelle funzioni senza parametri */

void _setargv__(void)                   /* sostituisce la _setargv__() di libreria */
{
}
```

[398] Solo a partire dalla versione 3.0 del DOS. Se il programma è invocato sotto versioni precedenti argv[0] contiene la stringa "C".

[399] L'environment non c'entra nulla con la command line, ma già che ci siamo...

[400] Nelle librerie del C Microsoft troviamo setenvp() e setargv(): la sostanza non cambia.

```
void _setenvp__(void)                /* sostituisce la _setenvp__() di libreria */
{
}

void main(void)                          /* non usa command line ed environment */
{
    ....
}
```

Naturalmente si può dichiarare una sola funzione fittizia, qualora l'altra sia necessaria:

```
#pragma option -k-          /* evita stack frame nelle funzioni senza parametri */

void _setenvp__(void)                /* sostituisce la _setenvp__() di libreria */
{
}

void main(int argc, char **argv)                        /* non usa l'environment */
{
    ....
}
```

La dichiarazione di entrambe le funzioni `_set...()` riduce di alcune centinaia di byte la dimensione dell'eseguibile[401] (`.COM` o `.EXE`) e ne rende leggermente più rapido il caricamento.

Entrambe `_setargv__()` e `_setenvp__()` sono eseguite prima di `main()`, siano esse le funzioni di libreria o quelle dichiarate nel sorgente: in questo caso nulla vieta al programmatore di crearne versioni personalizzate, non necessariamente "nullafacenti".

WILDARGS.OBJ

Una versione più sofisticata di `_setargv__()` è quella implementata nel modulo `WILDARGS.OBJ` (fornito con il compilatore): essa è in grado di espandere le wildcards e restituire in `argv` l'elenco completo dei file corrispondenti ai nomi "collettivi" specificati sulla command line. Se il file `PIPPO.C` è compilato con:

```
bcc pippo.c wildargs.obj
```

il vettore `argv` ricavato dalla command line dell'esempio di pag. può avere più di quattro elementi, in quanto `*.dat` è espanso in tante stringhe quanti sono i file `.dat` effettivamente presenti nella directory. Solo se non vi è alcun file `.dat` `argv` ha cinque elementi ed `argv[4]` è ancora "`*.dat`".

Esiste un solo object file `WILDARGS.OBJ` per tutti i modelli di memoria (pag. 143).

PSP e command line

Le variabili `argc` e `argv` e il file `WILDARGS.OBJ` costituiscono gli strumenti standard per la gestione della command line: essi rendono disponibili gli elemeti (stringhe) che la compongono. Chi desideri andare "al di là" del C, ed accedere direttamente alla riga di comando digitata al prompt[402], deve vedersela con il Program Segment Prefix (vedere pagina 324).

---

[401] Il compilatore, diligentemente, inserisce un riferimento esterno (che il linker risolve con una ricerca in libreria) solo per quelle funzioni il cui codice non si trova nel sorgente.

[402] Priva, però, del nome del programma e degli eventuali simboli di piping ('|') e redirezione ('<', '>').

Infatti, nel PSP, il byte ad offset `80h` memorizza la lunghezza della command line (escluso il nome del programma ed incluso il `CR` terminale); ad offset `81h` si trova la riga di comando (a partire dal primo spazio o tabulazione seguente il nome del programma), che include il carattere di `CR`.

Il mini-programma dell'esempio seguente, compilato sotto Borland C++ 2.0, stampa la command line, o meglio la sequenza di argomenti passati al programma tramite questa.

```
#include <stdio.h>
#include <dos.h>                                                    // per MK_FP()
#include <string.h>                                             // per _fstrncpy()

void main(void)
{
    int cmdlen;
    char cmdline[127];
    extern unsigned _psp;                          // indirizzo di segmento del PSP

    cmdlen = (int)(*(char far *)MK_FP(_psp,0x80));
    _fstrncpy((char far *)cmdline,(char far *)MK_FP(_psp,0x81),cmdlen);
    cmdline[cmdlen] = NULL;
    puts(cmdline);
}
```

Si noti che `main()` non ha parametri: la command line viene infatti letta direttamente nel PSP. La variabile `cmdlen` viene valorizzata con la lunghezza della command line: il doppio cast è necessario in quanto detto dato è costituito, nel PSP, da un solo byte; il cast di `MK_FP()` (pag. 24) a `(int far *)` avrebbe l'effetto di restituire una word in cui il byte più significativo è, in realtà, il primo carattere della riga di comando (di solito un blank). Il buffer `cmdline` è dimensionato a 127 byte: infatti tale è la massima lunghezza della command line (incluso il `CR` terminale, che deve essere sostituito con un `NULL` per ottenere una stringa stampabile dalle funzioni C). Per copiare in cmdline la porzione di PSP di nostro interesse è utilizzata `_fnstrcpy()`, che è equivalente a `strncpy()` ma accetta puntatori `far`[403]. Se le librerie del compilatore utilizzate non includono una versione `far` della `strncpy()` è necessario copiare la stringa byte per byte, ad esempio con un ciclo while:

```
    ....
    register i = 0;
    ....
    while((cmdline[i] = *(char far *)MK_FP(_psp,i+0x81)) != (char)0x0D)
        i++;
    cmdline[i] = NULL;
    ....
```

Dal momento che la seconda metà (gli ultimi 128 byte) del PSP è utilizzata dal DOS come DTA (Disk Transfer Address) di default per il programma, è indispensabile che le operazioni descritte siano le prime eseguite dal programma stesso, onde evitare che la command line sia sovrascritta.

Una gestione Unix-like

Nei sistemi Unix la command line è spesso utilizzata per fornire al programma invocato direttive, dette opzioni, di modifica degli eventuali default di funzionamento. Nel tempo, la modalità di elencazione delle opzioni ha raggiunto, sia pure in modo non esplicitamente formale, un livello di

[403] Un puntatore al PSP è, per definizione, `far` ed impone l'uso di `_fstrncpy()`, mentre il buffer `cmdline` è `near`: ciò spiega il cast `(char far *)cmdline`. La `_fstrncpy()` è la versione per puntatori a 32 bit della `strncpy()` (vedere pag. 25 e seguenti).

standardizzazione tale da poter parlare di una vera e propria sintassi delle opzioni, in parte ripresa e spesso seguita anche in ambiente DOS.

I parametri di `main()` `argc` e `argv` costituiscono solo una base di partenza per implementare una gestione Unix-like delle opzioni di command line: il resto è lasciato alla buona volontà del programmatore. Proviamo a scendere nel dettaglio: lo scopo è definire una sintassi per le command line option analoga a quella comunemente seguita nei sistemi Unix, e realizzare un insieme di funzioni libreria in grado di implementare in qualunque programma la gestione delle opzioni secondo quella sintassi.

La sintassi

Va innazitutto precisato che per opzione si intende un carattere alfanumerico, detto *optLetter*, preceduto sulla command line da un particolare carattere, detto *switch character*, che ha proprio lo scopo di identificare la optLetter quale opzione. In ambiente DOS lo switch character è, solitamente, la barra '`/`'; nei sistemi Unix è usato il trattino '`-`'. Il programma può prevedere che l'opzione supporti un parametro, detto argomento, rappresentato da una sequenza di caratteri qualsiasi, nel qual caso il carattere alfanumerico che rappresenta l'opzione è detto *argLetter* (e non optLetter).

Il programma riconosce le optLetter e le argLetter in quanto esse sono elencate in una stringa, convenzionalmente denominata `optionS`, in base alle regole elencate di seguito:

1) La stringa `optionS` non deve contenere spazi[404].

2) Le optLetter e le argLetter non possono essere segni di interpunzione.

3) Se il programma deve considerare equivalenti maiuscole e minuscole è necessario inserire in `optionS` sia la maiuscola che la minuscola per ogni optLetter o argLetter.

4) Ogni argLetter è seguita dal carattere '`:`'.

5) Per le argLetter non è data alcuna indicazione sugli argomenti validi: spetta al programma valutarli ed accettarli o segnalare eventuali errori.

Lo switch character e la optLetter (o argLetter seguita dal proprio argomento) costituiscono un *option cluster* (gruppo di opzione), nell'ambito del quale valgono le regole seguenti:

1) Lo switch deve essere preceduto da uno spazio e seguito immediatamente da optLetter o argLetter.

2) Uno switch isolato tra due spazi è un errore di sintassi.

3) Uno switch seguito da un carattere non compreso tra le optLetter e le argLetter riconosciute dal programma è un errore di sintassi.

4) Una optLetter può essere seguita da uno spazio o dalla successiva optLetter o argLetter.

5) Una argLetter deve essere seguita dal proprio argomento, che non può essere uno spazio.

6) Tra una argLetter e l'argomento può esservi uno spazio oppure un carattere '`:`'.

---

[404] Per spazio si intende uno o più blank o tabulazioni.

7) Se un argomento inizia con il carattere ':', questo deve essere ripetuto.

8) Un argomento può contenere (anche quale carattere iniziale) lo switch character; se contiene spazi deve essere compreso tra virgolette.

9) Un argomento deve essere seguito da uno spazio.

10) Maiuscole e minuscole non sono equivalenti, tanto in optLetter e argLetter quanto negli argomenti.

11) Gli option cluster devono essere elencati tutti all'inizio della command line; la prima stringa presente in essa non introdotta dallo switch character è considerata il primo dei parametri non-option.

12) Se il primo parametro non-option inizia con lo switch character, questo deve essere ripetuto due volte.

13) La ripetizione, nella command line, della medesima opzione è sintatticamente lecita: spetta al programma valutare se ciò costituisca un errore.

Spaventati? Non è il caso: tale insieme di regole è complesso solo apparentemente. In effetti esso formalizza una realtà probabilmente già nota. Vediamo un paio di esempi.

Se optionS è "A:F:PuU:wXZ:", nella command line:

```
PROG  /uPFPi /X /AL /f UnFile AltraStringa
```

PROG è il nome del programma e sono individuate le optLetter 'u', 'P' e 'X', nonché le argLetter 'F' (il cui argomento è "Pi") e 'A' (con l'argomento "L"), mentre la lettera 'f' non è un'opzione valida; le stringhe "UnFile" e "AltraStringa" costituiscono validi parametri non-option.

Se optionS è "AT:J:", nella command line

```
PROG -A -T:4 -AJ::k -T2 --123- -w- -  Parola "Due parole"
```

PROG è il nome del programma ed è riconosciuta la optLetter 'A', presente due volte; sono inoltre individuate le argLetter 'J' (con l'argomento ":k") e 'T', anch'essa presente due volte (con gli argomenti, rispettivamente, "4" e "2"). Il primo parametro non-option è la stringa "-123-"; i successivi sono le stringhe "-w-", "-", "Parola" e "Due parole". Il parametro "-" è valido, in quanto non è il primo non-option parameter (nel qual caso dovrebbe essere digitato come due trattini).

Se optionS è, ancora, "AT:J:", nella command line

```
PROG  -J -J "UnaSolaParola" -A
```

PROG è il nome del programma ed è riconosciuta la argLetter 'J', il cui argomento è "-J". Il primo non-option argument è la stringa "UnaSolaParola". La stringa "-A" rappresenta il secondo parametro non-option e non viene riconosciuta quale optLetter in quanto tutto ciò che segue il primo non-option argument è comunque considerato tale a sua volta.

Le funzioni per la libreria

Sulla scorta delle regole descritte è possibile realizzare le funzioni necessarie per una gestione Unix-like[405] della command line. Di seguito è presentato il listato di PARSEOPT.H, header file incluso nel sorgente delle funzioni e necessario anche per la realizzazione di programmi che le richiamano. Esso contiene i prototipi delle funzioni, i templates delle strutture appositamente definiti ed alcune costanti manifeste.

```
/******************************************************************************

    Barninga_Z! - OPTLIB

    filename - PARSEOPT.H

        getswitch()    - legge lo switch character DOS di default
        parseopt()     - analizza e memorizza le opzioni (low level)
        parseoptions() - analizza e memorizza le opzioni
        setswitch()    - stabilisce lo switch character DOS di defaul

******************************************************************************/

#ifndef __PARSEOPT__

#define  ILLEG_S           "Illegal option"                        // : nella cmd line
#define  ERROR_S           "Unknown option"                  // opzione non in optionS
#define  ERRCHAR           ((char)-1)                                // opzione errata

struct OPT {          // usato per le opzioni, gli argomenti e i valori di convalida
    char opt;         // restituiti dalle funzioni. A storeopt() deve essere passato
    char *arg;                  // un puntatore ad una struttura OPT. Se e' invocata
    int val;                  // parseopt() l'array di strutture e' allocato in modo
};                                    // automatico e ne e' restituito il puntatore.

struct VOPT {              // template per array strutture contenenti le opzioni e i
    char opt;                                // puntatori alle funzioni di validazione
    int (*fun)(struct OPT *tmp,int no);
};


int cdecl getswitch(void);
struct OPT *cdecl parseopt(int argc,char **argv,char *optionS,char sw,char
                                                                        *illegalS,
                                               char *errorS,struct VOPT valfuncs[]);
struct OPT *cdecl parseoptions(int argc,char **argv,char *optionS,
                                                            struct VOPT valfuncs[]);
int cdecl setswitch(char sw);

#define __PARSEOPT__
#endif

/****************** FINE DEL FILE PARSEOPT.H ***********************************/
```

L'interfaccia utente di alto livello è costituito dalla `parseoptions()`, che gestisce al proprio interno, secondo la sintassi descritta poco sopra, tutte le operazioni necessarie per processare le opzioni specificate sulla riga di comando del programma.

I suoi primi due parametri sono gli ormai noti `argc` e `argv`, che devono quindi essere dichiarati parametri di `main()`; il terzo parametro è la stringa `optionS`, che, come descritto, contiene

[405] La principale differenza tra dette regole e la consuetudine Unix è che quest'ultima, generalmente, non ammette l'uso del carattere `':'` quale separatore tra una `argLetter` e relativo parametro.

l'elenco delle optLetter e delle argLetter. Il quarto parametro è l'indirizzo di un array di strutture di tipo VOPT, il cui template è definito, come si vede, in PARSEOPT.H. Ogni struttura è formata da due campi:

```
struct VOPT {
    char opt;
    int (*fun)(struct OPT *tmp,int no);
};
```

Il campo opt è un carattere e rappresenta una optLetter o una argLetter, mentre il campo fun, puntatore a funzione, contiene l'indirizzo della funzione di validazione dell'optLetter o argLetter: parseoptions(), quando viene individuata sulla command line la optLetter o argLetter del campo opt, lancia la funzione il cui indirizzo è contenuto in fun, passandole un puntatore a struttura OPT (di cui diremo tra breve) e un intero esprimente la posizione dell'opzione sulla command line. La funzione a cui punta fun è definita dall'utente. Vediamo un esempio: se la optionS contiene la optLetter 'a' e abbiamo definito la funzione valid_a(), che deve essere invocata quando -a è specificata sulla riga di comando, si ha:

```
#include <STOREOPT.H>
....

char *optionS = "abc:";
....

int valid_a(struct OPT *tmp,int no)
{
    ....
}

....

int valid_err(struct OPT *tmp,int no)
{
    ....
}

int valid_glob(struct OPT *tmp,int no)
{
    ....
}

struct VOPT valfuncs[] = {
    ....
    {'a',valid_a},
    ....
    {ERRCHAR,valid_err},
    ....
    {NULL,valid_glob}
};

....

void main(int argc,char **argv)
{
    struct OPT *optArray;

    if(!(optArray = parseoptions(argc,argv,optionS,valfuncs)))
        perror("Problem");
    ....
```

Tralasciamo, per il momento, il contenuto di `valid_a()` e delle altre funzioni di validazione opzioni per analizzare alcune altre importanti caratteristiche dell'array `valfuncs`: il campo `opt` dell'ultimo elemento dell'array deve essere il carattere ASCII `0` (`NULL`), in quanto è proprio questo il "segnale" che consente a `parseoptions()` di capire che nell'array non vi sono altri elementi. Il campo `fun` corrispondente può, a sua volta, contenere `NULL`; se invece è inizializzato con un puntatore a funzione, `parseoptions()` lo utilizza per invocare quella funzione per ogni oggetto non-option incontrato sulla command line dopo l'ultima opzione. E' proprio questo il caso dell'esempio: se ipotizziamo che sulla riga di comando l'ultima opzione sia seguita da tre non-option items, `valid_glob()` è chiamata tre volte.

Inoltre, se il campo `opt` di uno degli elementi di `valfuncs` contiene il valore definito dalla costante manifesta `ERRCHAR`, la funzione indirizzata dal campo `fun` corrispondente, nell'esempio `valid_err()`, è chiamata ogni volta che sulla command line è incontrata una optLetter o argLetter non valida (non inclusa nella `optionS`).

L'array `valfuncs` ha pertanto numero di elementi pari alle optLetter e argLetter in `optionS`, più, opzionalmente, un elemento per la gestione delle opzioni errate, più un elemento finale con campo `opt` inizializzato a `NULL` (e campo `fun` nullo o puntatore alla funzione di validazione dei non-options items).

La `parseoptions()` restituisce `NULL` in caso di errore di sistema[406], nel qual caso può essere utilizzata `perror()` per visualizzare la descrizione dell'errore (vedere pag. 499). Se, al contrario, l'elaborazione termina con successo viene restituito un puntatore ad array di strutture `OPT`, il cui template è definito in `PARSEOPT.H`:

```
struct OPT {
    char opt;
    char *arg;
    int val;
};
```

Vediamo il significato di ogni elemento dell'array (nell'esempio `optArray`) e, per ogni elemento, il significato dei singoli campi. La prima struttura `OPT` presente in `optArray` (`optArray[0]`) contiene informazioni sugli elementi successivi. In particolare, il campo `opt` contiene l'indice[407], nell'array stesso, dell'elemento relativo al primo non-option item presente sulla command line; in altre parole `optArray[optArray[0].opt]` è la struttura `OPT` che, nell'array, si riferisce al primo non-option item. Il campo `arg` contiene sempre `argv[0]`, cioè il puntatore alla stringa che rappresenta il path completo del programma. Il campo `val` contiene il numero dei non-option items presenti sulla riga di comando; se non ve ne sono, entrambi i campi `opt` e `val` sono inizializzati a `0`.

Nell'array `optArray` troviamo poi un elemento per ogni optLetter e argLetter incontrata sulla command line. Nel caso di una optLetter, il campo `opt` contiene l'optLetter stessa, il campo `arg` è `NULL` e il campo val contiene l'intero restituito dalla funzione di validazione (campo `fun` della `struct VOPT`). Nel caso di una argLetter, invece, il campo `arg` punta ad una stringa contenente l'argomento fornito alla argLetter stessa sulla command line. Si noti che `arg` è sempre puntatore a stringa, anche nel caso in cui l'argomento della optLetter sia un numero: in tal caso occorre utilizzare l'appropriata funzione di conversione (ad esempio `atoi()`) per ottenere il valore numerico.

Se la optLetter o argLetter incontrata non si trova in `errorS`, il campo `arg` contiene il puntatore alla stringa `ERROR_S` definita in `PARSEOPT.H`. Nella riga di comando, i due punti ("`:`")

---

[406] Non nel caso di opzioni errate!

[407] Non è un carattere, ma un numero a 8 bit. Ciò significa che per esprimere il numero 4, il campo `opt` non contiene `'4'`, ma il carattere ASCII 4.

isolati sono interpretati come opzione illecita ed il campo `arg` contiene la stringa `ILLEGAL_S` (anch'essa definita in `PARSEOPT.H`). In entrambi i casi appena descritti, il campo `val` assume valore `-1`.

Si noti ancora che `parseoptions()` assume che lo switch character utilizzato sia quello di default per il sistema operativo: in DOS esso è la barra ("`/`"); se si preferisce utilizzare un altro carattere per introdurre le opzioni, ad esempio il trattino ("`-`") secondo la convenzione Unix, occorre modificare il default mediante una chiamata a `setswitch()`.

Ipotizziamo ora che il listato d'esempio faccia parte di un programma chiamato `PROG` ed invocato con la seguente riga di comando:

```
PROG /a /c14 /b pippo "Qui, Quo, Qua" pluto
```

Al ritorno da `parseoptions()` `optArray` contiene 6 elementi, valorizzati come segue:

`optArray[0].opt`	`4`	`4` è l'indice dell'elemento di `optArray` relativo al primo non-option item, se ve ne sono; altrimenti esso esprime il numero di elementi di cui si compone `optArray`.
`optArray[0].arg`	pathname di `PROG`	E' `argv[0]`.
`optArray[0].val`	`3`	Indica che vi sono 3 non-option item. Essi sono referenziati da `optArray[4]`, `optArray[5]` e `optArray[6]`; si osservi inoltre che `6`, ricavabile con `(optArray[0].opt+optArray[0].val-1)`, è l'indice dell'ultimo elemento dell'array. Infine, `argv[argc-optArray[0].val]` è il primo non-option item in `argv`.

`optArray[1].opt`	`'a'`	Prima opzione sulla command line.
`optArray[1].arg`	`NULL`	`'a'` è una optLetter: non ha argomento.
`optArray[1].val`	valore restituito da `valid_a()`	Se `'a'` non fosse un'opzione valida, o fosse specificata in modo errato, il campo conterrebbe il valore della costante manifesta `ERRCHAR` (`-1`).

`optArray[2].opt`	`'c'`	Seconda opzione sulla command line.
`optArray[2].arg`	`"14"`	`'c'` è una argLetter: `"14"` è il suo argomento, sempre sotto forma di stringa.
`optArray[2].val`	valore restituito da `valid_c()`	Se `'c'` non fosse un'opzione valida, o fosse specificata in modo errato, il campo conterrebbe il valore della costante manifesta `ERRCHAR` (`-1`).

optArray[3].opt	'b'	Terza opzione sulla command line.
optArray[3].arg	NULL	'b' è una optLetter: non ha argomento.
optArray[3].val	valore restituito da valid_b()	Se 'b' non fosse un'opzione valida, o fosse specificata in modo errato, il campo conterrebbe il valore della costante manifesta `ERRCHAR` (`-1`).

optArray[4].opt	0	Primo non-option item sulla command line.
optArray[4].arg	"pippo"	Il non-option item stesso, come stringa.
optArray[4].val	valore restituito da valid_glob()	Sempre `0`.

optArray[5].opt	1	Secondo non-option item sulla command line.
optArray[5].arg	"Qui, Quo, Qua"	Il non-option item stesso, come stringa.
optArray[5].val	valore restituito da valid_glob()	

optArray[4].opt	2	Primo non-option item sulla command line.
optArray[4].arg	"pluto"	Il non-option item stesso, come stringa.
optArray[4].val	valore restituito da valid_glob()	

E' il momento di descrivere le funzioni di validazione, quelle, cioè, i cui puntatori sono contenuti nei campi `fun` degli elementi dell'array `valfuncs`. Va precisato subito che il codice di dette funzioni dipende dalle esigenze del programma. Generalmente, ad ogni optLetter e argLetter corrisponde una appropriata funzione di validazione, ma nulla vieta di utilizzare una medesima funzione per controllare più opzioni: è sufficiente che i campi `fun` a queste corrispondenti siano tutti inizializzati con lo stesso puntatore. Analoghe considerazioni valgono a proposito della funzione richiamata in caso di opzione errata, e di quella richiamata una volta per ogni non-option item.

Ad esempio, la funzione `valid_err()` potrebbe visualizzare un messaggio di errore e interrompere il programma, tramite la funzione di libreria `exit()`.

Tutte queste funzioni, comunque, devono restituire un intero al fine di valorizzare secondo le esigenze del programmatore i campi `val` degli elementi dell'array di strutture `OPT`; inoltre tutte ricevono in ingresso, come parametri, il puntatore ad una `struct OPT` (la quale altro non è che l'elemento dell'array `optArray` corrispondente a quell'opzione) in cui i campi `opt` e `arg` sono già valorizzati come mostrato nella tabella sopra esposta, mentre il campo `val` contiene `0`. Può risultare utile, nelle funzioni di validazione delle argLetter, utilizzare il campo `arg` per effettuare i necessari controlli di validità dell'argomento associato all'argLetter stessa. L'intero che le funzioni di validazione ricevono come secondo parametro esprime la posizione dell'argLetter (o optLetter) sulla command line, e può essere utilizzato per le opportune verifiche qualora la posizione dell'opzione sia importante.

Nei programmi si ha spesso la necessità di modificare il comportamento dell'algoritmo a seconda che un'opzione sia stata o meno specificata: in questi casi può essere utile un flag, inizializzato dalla funzione di validazione e controllato laddove occorra nel corso dell'elaborazione. Un metodo efficiente di implementare tale tecnica di gestione delle opzioni è rappresentato da una variabile globale in cui ogni bit è associato ad una opzione. La funzione di validazione pone a `1` il bit; questo è poi verificato da altre funzioni del programma, che possono accedere a quella variabile, proprio in quanto globale.

Tornando al nostro esempio, le funzioni di validazione potrebbero agire su un intero senza segno:

```
unsigned int optBits;

int valid_a(struct OPT *tmp,int no)
{
    ....
    return(optBits |= 1);
}

int valid_b(struct OPT *tmp,int no)
{
    ....
    return(optBits |= 2);
}

int valid_c(struct OPT *tmp,int no)
{
    register int test;

    test = atoi(tmp->arg);
    if(test < 0 || test >= 10)
        valid_err(tmp,no);
    return(optBits |= 4);
}

int valid_glob(struct OPT *tmp,int no)
{
    static int progressiveLen;

    return(progressiveLen += strlen(tmp->arg);
}

int valid_err(struct OPT *tmp,int no)
{
    fprintf(stderr,"Errore: opzione '%c' non valida.\n",tmp->opt);
    exit(-1);
}
```

I bit della variabile globale `optBits` sono associati alle singole opzioni: in particolare, il bit 0 corrisponde all'optLetter '`a`', il bit 1 all'optLetter '`b`' e il bit 2 all'argLetter '`c`'; si noti che le costanti utilizzate per valorizzarli sono potenze di `2` (in particolare, 2 elevato ad esponente pari al numero del bit). In tal modo è possibile con un'operazione di `OR` su bit (composta con l'assegnamento) modificare il singolo bit desiderato. In qualunque altro punto del programma può utilizzare un test analogo al seguente:

```
if(optBits & 2)....
```

per verificare se l'opzione '`b`' è stata specificata. L'uso di costanti manifeste come

```
#define  OPTION_A     1
#define  OPTION_B     2
#define  OPTION_C     4
```

facilita notevolmente la vita.

Si noti che in luogo di una variabile integral si può utilizzare un campo di bit, con il vantaggio, tra l'altro, di non essere costretti ad utilizzare le operazioni di `AND` e `OR` su bit, in quanto ogni campo rappresenta di per sé un'opzione

```
struct OPTBITS optBits {
    unsigned optionA:1;
    unsigned optionB:1;
    unsigned optionC:1;
}

    ....
    optBits.optionA = 1;
    ....
    if(optBits.optionA)
    ....
```

Si noti che `valid_c()`, in caso di errore, chiama `valid_err()` per interrompere il programma. Inoltre, la scelta dei nomi delle funzioni di validazione è, ovviamente, libera: quelli utilizzati nell'esempio non sono vincolanti, né rappresentano un default.

Va ancora osservato che `parseoptions()` è perfettamente compatibile con `WILDARGS.OBJ` (pag. 477): se come non-option item sono specificati uno o più nomi di file, essi verranno trattati nel modo consueto: l'`argv` che `parseoptions()` riceve come secondo parametro contiene già i puntatori ai nomi generati dall'espansione delle wildcard "`*`" e "`?`".

Ed ecco, finalmente (?), il sorgente completo delle funzioni, listate in ordine alfabetico, che realizzano il meccanismo sin qui descritto. Per un esempio pratico di utilizzo vedere pag. 431.

```
/******************************************************************************

    Barninga_Z! - OPTLIB

    file - parseopt.c

    funzioni:

        getswitch    - legge lo switch character DOS di default
        gopError     - ritorna da storeopt() se vi e' un errore nella command line
        parseopt     - analizza e memorizza le opzioni (low level)
        parseoptions - analizza e memorizza le opzioni
        setswitch    - stabilisce lo switch character DOS di default
        storeopt     - analizza e memorizza le opzioni (internal only)

******************************************************************************/

#pragma  warn -pia              // evita warning per assegnamenti con test implicito

#include <alloc.h>
#include <string.h>

#include "parseopt.h"

#define  EINVFNC           1                              // Invalid function number
#define  EINVAL            19                                    // Invalid argument

// i prototipi di storeopt() e gopError() sono qui e non in PARSEOPT.H perche'
// si tratta di funzioni che l'utente non ha bisogno di chiamare direttamente
```

```
// in pratica si tratta di routines di servizio per le funzioni high-level

int cdecl gopError(struct OPT *cmd);
int cdecl storeopt(int argc,char **argv,char *optionS,char sw,struct OPT *cmd,
                                                    char *illegalS,char *errorS);


int pascal __IOerror(int dosErr);           // funzione di libreria; non documentata

static int optind;           // indice della prossima opzione. E' inzializzata a 1 in
                                // ingresso ed utilizzata da storeopt(). Deve essere
                           // riazzerata in uscita per consentire chiamate multiple a
                                                                      // parseopt()


/*----------------------------------------------------------------------------*

getswitch()        Restituisce lo switch character di default

SINTASSI     int cdecl getswitch(void);

INCLUDE      parsopt.h

PARAMETRI    Nessuno.

SCOPO        Ottiene il  carattere che  il DOS  abilita per  default ad
             introdurre gli  switches (opzioni)  sulla command line dei
             programmi. Se  è usata  per valorizzare il parametro sw di
             parseopt() o  storeopt(), tutte  le opzioni  sulla command
             line devono  essere precedute  dal carattere restituito da
             getswitch(). E' inoltre invocata da parseoptions().

RESTITUISCE  -1    funzione non supportata dal DOS.

             altro lo switch character di default per il DOS.

SORGENTE     getdossw.c

NOTE         Nessuna.

*----------------------------------------------------------------------------*/

int cdecl getswitch(void)
{
    asm {
        mov ax,0x3700;
        int 0x21;
        cmp al,0xFF;
        je notsupported;
        mov al,dl;
        xor ah,ah;
        jmp endfunc;
    }
notsupported:
    _AX = __IOerror(EINVFNC);
endfunc:
    return(_AX);
}


/*----------------------------------------------------------------------------*

gopError()         Ritorna da storeopt() in caso di errore - INTERNAL
```

```
SINTASSI     int cdecl gopError(struct OPT *cmd);

INCLUDE      parsopt.h

PARAMETRI    cmd           puntatore   all'elemento   dell'array   di
                           strutture OPT  (template in  storeopt.h) nel
                           quale l'opzione verrebbe memorizzata.

SCOPO        Ritorna da  storeopt() quando  questa individua  un errore
             nella command  line (es.:  opzione sconosciuta). Memorizza
             il valore  -1 nel  campo  val  della  struttura  e  chiama
             __IOerror().

RESTITUISCE  l'opzione (anche se non valida).

SORGENTE     storeopt.c

NOTE         Il programma non deve invocare questa funzione; essa è una
             funzione di servizio per storeopt().

*----------------------------------------------------------------------------*/

static int cdecl gopError(struct OPT *cmd)
{
    cmd->val = __IOerror(EINVAL);
    return((int)cmd->opt);
}


/*----------------------------------------------------------------------------*

parseopt()         Analizza la command line - LOW LEVEL

SINTASSI     int cdecl parseopt(int argc,char **argv,char *optionS,char sw,
                                  char *illegalS,char *errorS,struct VOPT *valfuncs);

INCLUDE      parsopt.h

PARAMETRI    argc          numero di  parametri sulla command line + 1.
                           E' generalmente argc parametro di main().

             argv          array  di   puntatori  ai   parametri  della
                           command line. E' generalmente argv parametro
                           di main().

             optionS       puntatore alla  stringa contenente  tutti  i
                           caratteri    opazione    riconosciuti    dal
                           programma.

             sw            lo  switch   character  che  introduce  ogni
                           opzione (o gruppo di opzioni).

             illegalS      puntatore alla  stringa  rappresentatnte  il
                           messaggio di  errore corrispondente  all'uso
                           non lecito  dei due  punti (:) nella command
                           line.

             errorS        puntatore  alla   stringa  utilizzata   come
                           messaggio  di   errore  per  un'opzione  non
                           compresa in optionS.

             valfuncs      puntatore  ad   un  array   di  struct  VOPT
                           (template definito in parsopt.h). L'array ha
                           un elemento per ogni opzione che si desidera
```

```
                         testare o  convalidare. Ogni  elemento è una
                         struct VOPT, la quale ha due campi: il primo
                         (char opt) contiene il carattere-opzione; il
                         secondo  (int   (*fun)(struct  OPT   *,int))
                         contiene il puntatore alla funzione che deve
                         essere invocata  per testare l'opzione. Tale
                         funzione  deve   essere  di   tipo  int   ed
                         accettare  due  parametri,  un  puntatore  a
                         struct OPT  (template  in  parsopt.h)  e  un
                         int).  Il   primo   punta   alla   struttura
                         valorizzata    da     storeopt()    estraendo
                         l'opzione dalla  command  line,  il  secondo
                         rappresenta la  posizione dell'opzione nella
                         command  line.  L'array  DEVE  avere,  quale
                         ultimo elemento,  una struct  VOPT in cui il
                         campo opt  è NULL:  la funzione  puntata dal
                         campo  fun   viene  invocata   per   ciascun
                         parametro   non-opzione   incontrato   sulla
                         command line.  Inoltre, se  il campo  opt di
                         uno  qualunque   degli  elementi  dell'array
                         contiene  il  carattere  ERRCHAR (parsopt.h)
                         la funzione puntata dal corrispondente campo
                         fun    viene   invocata    quando   l'opzione
                         restituita  da storeopt() è  sconosciuta  o,
                         comunque, in caso di errore.

SCOPO        Scandisce e  memorizza  in  un  array  di  struct  VOPT  i
             parametri incontrati  sulla command  line, per la sintassi
             della quale  si  veda  storeopt().  La  parseopt()  invoca
             storeopt() finché  tutte i  parametri della  command  line
             sono stati  scanditi (opzionalmente testati) e memorizzati
             nell'array. Alloca  automaticamente memoria  per  l'array,
             che al termine contiene una struct OPT per ogni parametro.

RESTITUISCE  NULL  in caso di errore di allocazione.


SORGENTE     parsopll.c

NOTE         I  campi   della  prima   struct  OPT   dell'array   hanno
             significati particolari:  il campo  opt contiene  l'indice
             del  (elemento   dell'array   corrispondente   al)   primo
             parametro non-opzione  nella command  line, se ve ne sono;
             il  campo arg contiene argv[0];  il campo  val contiene il
             numero totale di argomenti non-opzione nella command line.
             Ancora,  il  campo  opt  delle  struct   OPT   relative  a
             parametri  non-opzione sono  valorizzati così:  0  per  il
             primo trovato, 1 per il secondo, etc..

*---------------------------------------------------------------------------*/

struct OPT *cdecl parseopt(int argc,char **argv,char *optionS,char sw,
                                 char *illegalS,char *errorS,struct VOPT valfuncs[])
{
    struct OPT *cmd = NULL, *tmp = NULL;
    int no, i, fl, carg;
    char option;
    extern int optind;

    if(!(cmd = (struct OPT *)realloc(cmd,sizeof(struct OPT))))
        return(NULL);
    optind = 1;
    for(no = 1;(carg = storeopt(argc,argv,optionS,sw,cmd,illegalS,errorS)) > 0;
                                                                           no++) {
```

```
        if(!(tmp = (struct OPT *)realloc(cmd,(no+1)*sizeof(struct OPT)))) {
            optind = 0;
            return(NULL);
        }
        cmd = tmp;
        (tmp += no)->opt = cmd->opt;
        tmp->arg = (*cmd->arg == ':') ? cmd->arg + 1 : cmd->arg;
        option = (!(tmp->val = cmd->val)) ? tmp->opt : ERRCHAR;
        for(i = 0; (valfuncs+i)->opt; i++)
            if((valfuncs+i)->opt == option)
                if((valfuncs+i)->fun)
                    tmp->val = (*((valfuncs+i)->fun))(tmp,no);
    }
    cmd->opt = no;
    cmd->arg = argv[0];
    cmd->val = 0;
    for(i = 0; (valfuncs+i)->opt; i++);
    for(fl = 0, carg = -carg; carg < argc; carg++) {
        if(!(tmp = (struct OPT *)realloc(cmd,(no+1)*sizeof(struct OPT))))
            free(cmd);
        cmd = tmp;
        (tmp += no++)->opt = (char)cmd->val++;
        tmp->arg = (*argv[carg] == sw && !fl) ? fl++, argv[carg] + 1 : argv[carg];
        tmp->val = ((valfuncs+i)->fun) ? (*(valfuncs+i)->fun)(tmp,carg) : 0;
    }
    optind = 0;
    return(cmd);
}


/*-----------------------------------------------------------------------------*

parseoptions()     Analizza i parametri della commnad line

SINTASSI     int cdecl parseoptions(int argc,char **argv,
                                          char *optionS,struct VOPT *valfuncs);

INCLUDE      parsop.c

PARAMETRI    argc          numero di  parametri sulla command line + 1.
                           E' generalmente argc parametro di main().

             argv          array  di   puntatori  ai   parametri  della
                           command line. E' generalmente argv parametro
                           di main().

             optionS       puntatore alla  stringa contenente  tutti  i
                           caratteri    opazione    riconosciuti    dal
                           programma.

             valfuncs      puntatore  ad  un  array  di  struct  VOPT
                           (template definito in parsopt.h). L'array ha
                           un elemento per ogni opzione che si desidera
                           testare o  convalidare. Ogni  elemento è una
                           struct VOPT, la quale ha due campi: il primo
                           (char opt) contiene il carattere-opzione; il
                           secondo  (int   (*fun)(struct  OPT   *,int))
                           contiene il puntatore alla funzione che deve
                           essere invocata  per testare l'opzione. Tale
                           funzione  deve   essere  di   tipo  int   ed
                           accettare  due  parametri,  un  puntatore  a
                           struct OPT  (template  in  parsopt.h)  e  un
                           int).  Il   primo   punta   alla   struttura
                           valorizzata    da    storeopt()    estraendo
```

```
                           l'opzione dalla  command  line,  il  secondo
                           rappresenta la  posizione dell'opzione nella
                           command  line.  L'array  DEVE  avere,  quale
                           ultimo elemento,  una struct  VOPT in cui il
                           campo opt  è NULL:  la funzione  puntata dal
                           campo  fun   viene  invocata   per   ciascun
                           parametro   non-opzione   incontrato   sulla
                           command line.  Inoltre, se  il campo  opt di
                           uno  qualunque   degli  elementi  dell'array
                           contiene  il  carattere  ERRCHAR (parsopt.h)
                           la funzione puntata dal corrispondente campo
                           fun   viene   invocata   quando   l'opzione
                           restituita  da storeopt() è  sconosciuta  o,
                           comunque,  in  caso   di   errore.   Vedere
                           storeopt() circa la struct OPT.

SCOPO        Analizza la  command line e ne memorizza i parametri in un
             array di  struct OPT  dopo averli (opzionalmente) testati.
             Vedere storeopt()  e parseopt().  Invoca  parseopt()  dopo
             avere valorizzato  il parametro sw con lo switch character
             di default  del DOS (getswitch()), illegalS con la stringa
             "Illegal option" e errorS con la stringa "Unknown option".

RESTITUISCE  NULL  in caso di errore di allocazione.


SORGENTE     parsopll.c

NOTE         I  campi   della  prima   struct  OPT   dell'array   hanno
             significati particolari:  il campo  opt contiene  l'indice
             del  (elemento   dell'array   corrispondente   al)   primo
             parametro non-opzione  nella command  line, se ve ne sono;
             il  campo arg contiene argv[0];  il campo  val contiene il
             numero totale di argomenti non-opzione nella command line.
             Ancora,  il  campo  opt  delle  struct   OPT   relative  a
             parametri  non-opzione sono  valorizzati così:  0  per  il
             primo trovato, 1 per il secondo, etc..

*----------------------------------------------------------------------------*/

struct OPT *cdecl parseoptions(int argc,char **argv,char *optionS,
                                                           struct VOPT valfuncs[])
{
    static char *illegalS = ILLEG_S;
    static char *errorS = ERROR_S;
    char sw;

    return(((sw = (char)getswitch()) < 0) ? NULL :
                           parseopt(argc,argv,optionS,sw,illegalS,errorS,valfuncs));
}


/*----------------------------------------------------------------------------*

setswitch()        Imposta lo switch character di default per il DOS

SINTASSI     int cdecl setswitch(char sw);

INCLUDE      parsopt.h

PARAMETRI    sw            il nuovo  switch character  di  default  del
                           DOS.

SCOPO        Imposta il  nuovo switch character di default del DOS. Per
```

```
             sapere semplicemente  qual è lo switch character attuale è
             possibile usare getswitch().

RESTITUISCE  il vecchio switch character, oppure, in caso di errore:
             -1    funzione non  supportata dal  DOS (__IOerror() setta
                   errno e doserrno).


SORGENTE     setdossw.c

NOTE         Nessuna

*-----------------------------------------------------------------------------*/

int cdecl setswitch(char sw)
{
    asm {
        mov ax,0x3700;
        int 0x21;
        cmp al,0xFF;
        je notsupported;
        push dx;
        mov ax,0x3701;
        mov dl,byte ptr sw;
        int 0x21;
        pop dx;
        cmp al,0xFF;
        je notsupported;
        mov al,dl;
        xor ah,ah;
        jmp endfunc;
    }
notsupported:
    _AX = __IOerror(EINVFNC);
endfunc:
    return(_AX);
}


/*-----------------------------------------------------------------------------*

storeopt()         Memorizza in struct gli argomenti della cmd line

SINTASSI     int cdecl storeopt(int argc,char **argv,char *optionS,
                                  char sw,struct OPT *cmd,char *illegalS,
                                                         char *errorS);

INCLUDE      parsopt.h

PARAMETRI    argc          numero di  parametri sulla command line + 1.
                           E', solitamente, argc parametro di main().

             argv          array  di   puntatori  ai   parametri  nella
                           command   line.    E',   solitamente,   argv
                           parametro di main().

             optionS       puntatore  alla   stringa  contenente  tutti
                           caratteri-opzione      riconosciuti       dal
                           programma.

             sw            lo switch character.

             cmd           puntatore  alla   struct  OPT  (template  in
                           parsopt.h)  nella   quale  i  dati  relativi
```

all'opzione correntemente processata devono essere memorizzati. Detta struct OPT è uno delgi elementi dell'array allocato da parseopt(). Una struct OPT è composta di tre elementi: il primo (char opt) conterrà il carattere-opzione; il secondo (char *arg) punterà all'argomento dell'opzione (NULL se l'opzione non accetta argomenti); il terzo (int val) varrà normalmente 0: in caso di errore (opzione sconosciuta o digitata in modo scorretto o con argomento non valido) sarà posto a -1. Esso è inoltre utilizzato per memorizzare il valore restituito dalla funzione di validazione dell'opzione invocata da parseopt().

illegalS — puntatore alla stringa usata come argomento del carattere due punti (:) quando utilizzato come opzione (illecita) nella command line.

errorS — puntatore alla stringa usata come argomento dei caratteri-opzione non presenti in optionS.

SCOPO — memorizza in una struttura le opzioni presenti nella command line. La sintassi, molto vicina a quella adottata come standard nei sistemi Unix, è la seguente:

option : sw optLetter argLetter argument

dove

- sw è lo switch character
- non ci sono spazi tra lo switch character e ogni optLetter o argLetter.
- optLetter e argLetter non sono segni di punteggiatura.
- optLetter e argLetter devono comparire in optionS.
- argLetter, se presenti, devono essere seguite in optionS da ':'.
- argument è una stringa che termina con uno spazio e può essere preceduta da uno spazio. Può includere lo switch character.
- maiuscole e minuscole non sono equivalenti.

Sulla command line possono comparire più clusters (gruppi) di opzioni, ciascuno dei quali è introdotto dallo switch. Tutti i clusters di pozioni devono comparire prima dei parametri non-opzione (qualunque cosa non introdotta dallo switch, ad eccezione delle stringhe argomenti di opzioni). Una argLetter o optLetter può comparire più volte: è compito del programmatore scegliere se ciò vada considerato errore.

La stringa optionS consente il riconoscimento delle optLetter e argLetter valide. In essa ogni argLetter è seguita dai due punti (:). La storeopt() restituisce la

optLetter o argLetter processata; un valore minore di zero se non vi sono più opzioni sulla command line.

Lo switch isolato tra spazi è un errore.

Due switch characters consecutivi (ad es.: -- o //) vengono considerati il primo parametro non-opzione, il primo dei due switch è rimosso e storeopt() restituisce un valore negativo. Se vengono successivamente incontrate altre coppie di switch characters sono lasciate immutate. Pertanto tale combinazione può essere utilizzata se il primo parametro non-opzione inizia con lo switch. Es.: se PROG è il nome del programma, A è un'opzione valida, -5 è il primo argomento, --ABC-- è il secondo, -4 il terzo e - è lo switch, affinché la command line sia interpretata correttamente occorrerà scriverla come segue:

```
PROG -A --5 --ABC-- -4
```

La optind e' inizialmente 1 e rappresenta sempre l'indice del prossimo argomento di argv[], che storeopt() non ha ancora analizzato. Se e' utilizzato SWSW allora optind e' incrementata al successivo argomento prima che getopt() restituisca il suo valore cambiato di segno (fine opzioni)

Il carattere due punti (:) può separare una argLetter dal suo argomento sulla command line: esso viene ignorato. Se l'argomento inizia con il due punti, esso va ripetuto. Esempio:

```
-T:4     --->    argLetter = T,   argomento = 4

-T::4    --->    argLetter = T,   argomento = :4
```

Se è incontrata una lettera non inclusa in optionS, essa è comunque restituita da storeopt(), ma nel campo val della struct OPT viene memorizzato un valore negativo e non 0. Esempio: se il DOS switch è '/' (DOS default) e optionS è "A:F:PuU:wXZ:" allora 'P', 'u', 'w', e 'X' sono optLetter, mentre 'A', 'F', 'U', 'Z' sono argLetter. Una command line può essere:

```
    PROG  /uPFPi /X /A L /f UnFile AltraStringa
```

dove:

- 'u' e 'P' sono restituite come opzioni.
- 'F' è restituita con "Pi" come proprio argomento.
- 'X' è un'altra opzione.
- 'A' è restituita con "L" come argomento.
- 'f' non è presente in optionS, pertanto è restituita memorizzando -1 nel campo val della struct OPT. Essa è testata dalla funzione puntata dal campo fun della struct VOPT che ha il carattere ERRCHAR nel campo opt (se tale struct è definita nell'array).
- "UnFile" non è un'opzione e viene testata con la funzione puntata dall'elemento fun dell'ultima struct VOPT dell'array.

```
                 -  idem dicasi per "AltraStringa".

RESTITUISCE  L'opzione processata, anche se non presente in optionS: in
             questo caso  il campo  val della  struct OPT  è  negativo;
             esso,   cambiato    di    segno,    rappresenta    l'indice
             dell'elemento di  argv[] successivo  a quello  attualmente
             processato.

SORGENTE     storeopt.c

NOTE         I campi della prima struct OPT nell'array di strutture OPT
             hanno un  significato speciale  (essi sono  valorizzati da
             parseopt() o  parseoptions(), non  da storeopt()).  Vedere
             parseopt() o parseoptions() per maggiore dettaglio.

-----------------------------------------------------------------------------*/

int cdecl storeopt(int argc,char **argv,char *optionS,char sw,struct OPT *cmd,
                                                        char *illegalS,char *errorS)
{
    extern int optind;        // numero della prossima opzione. Inizializzata a 1 da
    static char *letP;       // parseopt() in entrata e da essa riazzerata in uscita
    char *optP;

    while(argc > optind) {//while e' usato per evitare una goto; non e' un vero loop
        cmd->val = 0;
        if(!letP) {
            if((!(letP = argv[optind])) || (*letP++ != sw))
                break;
            if(*letP == sw)
                break;
        }
        if(!(cmd->opt = *letP++)) {
            if(argv[++optind]) {
                letP = NULL;
                continue;
            }
            break;
        }
        if(cmd->opt == ':') {
            cmd->arg = illegalS;
            return(gopError(cmd));
        }
        if(!(optP = strchr(optionS,cmd->opt))) {
            cmd->arg = errorS;
            return(gopError(cmd));
        }
        if(*(optP+1) == ':') {
            optind++;
            if(!(*letP)) {
                if (argc <= optind)
                    return(gopError(cmd));
                letP = argv[optind++];
            }
            cmd->arg = letP;
            letP = NULL;
        }
        else {
            if(!(*letP)) {
                optind++;
                letP = NULL;
            }
            cmd->arg = NULL;
```

```
        }
        return((int)cmd->opt);
    }
    cmd->arg = letP = NULL;
    return((int)(cmd->opt = -optind));
}
```

La funzione `storeopt()` è il cuore del meccanismo. Essa è progettata[408] per scandire una stringa alla ricerca di optLetter e argLetter e, per queste ultime, isolare l'argomento fornito. La `storeopt()` analizza una sola stringa ad ogni chiamata, perciò deve essere utilizzata all'interno di un loop che provveda a gestire opportunamente i puntatori contenuti in `argv`. A ciò provvede `parseopt()`, che, al ritorno da `storeopt()` si occupa di lanciare la funzione di validazione dell'opzione mediante l'indirezione del puntatore contenuto nel campo `fun` della struttura di template `VOPT`:

```
(*(valfuncs+i)->fun)(tmp,carg)
```

I parametri `tmp` e `carg`, coerentemente con il prototipo delle funzioni di validazione, sono il puntatore alla `struct OPT` e l'intero rappresentante la posizione dell'opzione sulla command line.

Se `storeopt()` restituisce una condizione di errore (tramite `gopError()`), `parseopt()` ricerca nell'array `valfuncs` un elemento il cui campo `opt` sia inizializzato con il valore della costante manifesta `ERRCHAR` e, se questo esiste, lancia la funzione di validazione corrispondente (`valid_err()` nell'esempio di poco fa).

Quando `storeopt()` segnala, tramite la restituzione di un valore negativo, che non vi sono più optLetter e argLetter, `parseopt()` considera i restanti elementi di `argv` come non-option items: se il campo `fun` dell'ultimo elemento dell'array `valfuncs` non è `NULL` viene lanciata la funzione da esso indirizzata una volta per ogni non-option item (la funzione è sempre la stessa, ma cambiano i valori dei campi della `struct OPT` di cui essa riceve il puntatore.

Ad ogni iterazione `parseopt()` alloca la memoria necessaria per aggiungere all'array di strutture `OPT` quella corrispondente all'item della command line attualmente processato; al termine dell'elaborazione essa restituisce l'indirizzo dell'array oppure `NULL` in caso di errore (fallita allocazione della memoria).

Oltre ai parametri richiesti dalla `parseoptions()`, la `parseopt()` necessita dello switch character (il carattere che introduce le optLetter e argLetter) e delle due stringhe da utilizzare come campi `arg` per le opzioni errate e illecite. Risulta evidente, a questo punto, che `parseoption()` è semplicemente un "guscio" di alto livello per `parseopt()`, alla quale passa, oltre ai parametri ricevuti, anche quelli appena elencati, fornendone valori di default. In particolare, per le due stringhe sono utilizzate le costanti manifeste `ERROR_S` e `ILLEGAL_S` definite in `PARSEOPT.H`, mentre per lo switch character è utilizzato il valore restituito dalla `getswitch()`, che richiede al DOS il carattere di default.

Al riguardo, sono necessarie alcune precisazioni. Come si è detto, lo switch character di default è la barra in DOS e il trattino in Unix: ciò implica che se si desidera realizzare in ambiente DOS un'interfaccia il più possibile Unix-like occorre dimenticarsi di `parseoptions()` e chiamare direttamente `parseopt()`, avendo cura di fornirle come parametro lo switch character desiderato. In alternativa è possibile modificare l'impostazione di default del DOS mediante la `setswitch()`, prima di chiamare `parseoptions()`.

La `setswitch()` richiede come parametro il carattere che si desidera impostare come nuovo default e restuisce il precedente default (`-1` in caso di errore). La `getswitch()` non richiede parametri

[408]In realtà, `storeopt()` deriva dalla funzione `getopt()`, che nei sistemi Unix rappresenta l'algoritmo standard (fornita come sorgente con molti compilatori proprio per consentire agli sviluppatori di scrivere applicazioni con interfaccia coerente) per la scansione della command line.

e restituisce il default attuale. Le due funzioni si basano sull'int 21h, servizio 37h, subfunzioni 00h (`GetSwitchChar`) e 01h (`SetSwitchChar`); va sottolineato che detto servizio non è ufficialmente documentato e, pertanto, potrebbe non essere disponibile in tutte le versioni di DOS: in particolare, a partire dal DOS 5.0, la subfunzione 01h è ignorata (non determina la restituzione di un errore, ma non ha comunque alcun effetto) ed è perciò necessario utilizzare direttamente `parseopt()` se si desidera utilizzare il trattino come switch character.

INT 21H, SERV. 37H, SUBF. 00H: GET SWITCH CHARACTER

Input	`AH` `AL`	`37h` `00h`
Output	`AL` `DL`	`FFh` in caso di errore (funzione non supportata). Attuale switch character, se `AL` non è `FFh`.
Note		Se non vi è errore, le versioni di DOS fino alla 4.x restituiscono `00h` in `AL`; dalla 5.0 in poi `AL` = `2Fh`.

INT 21H, SERV. 37H, SUBF. 01H: SET SWITCH CHARACTER

Input	`AH`
	`AL`
	`DL`
		`37h`
		`01h`
		Nuovo switch character

Output	`AL`	`00h` se OK, `FFh` in caso di errore (funzione non supportata).
Note		Questa chiamata è ignorata dal DOS a partire dalla versione 5.0.

Va infine osservato che il listato potrebbe essere suddiviso in più sorgenti, uno per ogni funzione (con la sola eccezione di `gopError()` e `storeopt()`, da riunire in un unico file[409]); dalla compilazione si otterrebbero così più file `.OBJ`, da inserire in una libreria. Se ne avvantaggerebbero gli eseguibili incorporanti le funzionalità descritte, dal momento che in essi bverrebbero inclusi dal linker solo i moduli necessari.

LA GESTIONE DEGLI ERRORI

Le librerie della maggior parte dei compilatori implementano una modalità standard, derivata da Unix, di gestione degli errori restituiti dai servizi DOS utilizzati dalle funzioni di libreria. Concentriamo la nostra attenzione su alcune variabili globali, dichiarate nel file `STDLIB.H`:

[409] Dal momento che `gopError()` è una funzione di servizio per `storeopt()`, entrambi i moduli `.OBJ` verrebbero comunque inclusi dal linker nell'eseguibile.

```
extern char *sys_errlist[];
extern int errno;
extern int _doserrno;
```

L'array `sys_errlist` contiene i puntatori alle stringhe che descrivono i diversi errori[410]. Vi è una funzione dedicata alla gestione dei messaggi d'errore, `perror()`, che richiede una stringa quale parametro, la scrive sullo standard error seguita da `": "`, dalla stringa che costituisce l'elemento di `sys_errlist` corrispondente all'errore verificatosi e dal carattere `"\n"`. L'utilizzo di `perror()` non presenta difficoltà:

```
#include <stdio.h>                          // prototipi di fopen() e perror()

#define   PRG_NAME     "MYPROG"
....
    if(!(stream = fopen(filename,"r"))) {
        perror(PRG_NAME);
        return(0);
    }
```

Nell'ipotesi che `filename` contenga un pathname errato, `fopen()` non riesce ad aprire il file e restituisce `NULL`; la `perror()` produce il seguente output:

```
MYPROG: path not found
```

Come riesce `perror()` a individuare il messaggio? Semplice: si basa sul valore assunto dalla variabile `errno`, che può essere validamente utilizzata come indice. In altre parole, `sys_errlist[errno]` è la stringa che descrive l'errore. Dietro le quinte, tutte le funzioni di libreria che per svolgere il proprio compito utilizzano servizi DOS[411], se ricevono da questo un codice di errore lo passano ad una funzione (generalmente non documentata), la `__IOerror()`[412], che lo assegna a `_doserrno`, la quale contiene perciò il codice di errore DOS, e, tramite un'apposita tabella di conversione, ricava il valore appropriato da assegnare ad `errno`.

Può essere utile chiamare direttamente la `__IOerror()` nei propri sorgenti, soprattutto quando si scrivano funzioni destinate ad essere inserite in una libreria: risulta è immediato implementare la gestione degli errori in modo del tutto coerente con le librerie standard del compilatore utilizzato. E' sufficiente dichiarare nel sorgente il prototipo di `__IOerror()`:

```
int pascal __IOerror(int dosErr);
```

La parola riservata `pascal` ha lo scopo di modificare lo stile di chiamata della funzione: a tutte le funzioni dichiarate `pascal`, secondo lo standard di questo linguaggio, i parametri attuali sono passati in ordine diretto, cioè dal primo a sinistra all'ultimo a destra, e non viceversa, secondo quanto previsto invece, per default, dalle regole del C (pag. 92). Lo scopo è ottenere una chiamata più efficiente; la perdita della possibilità di gestire un numero variabile di parametri, in questo caso, non ha alcuna

---

[410] Sono, anche questi, piuttosto uniformi nelle diverse implementazioni. Ad esempio `"Not enough memory"`, `"Path not found"`, e così via.

[411] Si tratta di chiamate ad interrupt, per lo più all'int 21h.

[412] Avvertenza: i nomi di questa funzione e delle variabili globali, eccetto `errno`, possono variare da compilatore a compilatore, ma la sostanza non muta. In caso di dubbi, basta esplorare un po' i file `.H`. I nomi qui indicati sono validi per il compilatore Borland.

importanza, in quanto `__IOerror()` ne riceve uno solo, `dosErr`, che rappresenta il codice di errore restituito dalla routine DOS (solitamente nel registro `AX`). La `__IOerror()` restituisce sempre `-1`.

Vediamo un esempio. Vogliamo scrivere una funzione in grado di dirci, in un ambiente di rete, se un drive è locale o remoto: allo scopo si può utilizzare l'int 21h, servizio 44h, subfunzione 09h:

INT 21H, SERV. 44H, SUBF. 09H: DETERMINA SE IL DRIVE È REMOTO

Input	`AX`	`4409h`
	`BL`	Drive (`00h` = default, `01h` = `A:`, `02h` = `B:`, etc.)
Output	`DX`	Se `CarryFlag` = `0`, il bit 12 di `DX` a `1` indica che il drive è remoto; Se `CarryFlag` = `1` allora `AX` contiene il codice di errore.

Ed ecco il listato:

```
/********************

    BARNINGA_Z! - 1991

    ISREMOTE.C - isDriveRemote()

    int cdecl isDriveRemote(int driveNum);
    int driveNum;   drive da testare (0 = default, 1 = A:, ...)
    Restituisce: 0 = drive locale
                 1 = drive remoto
                -1 = errore (errno e _doserrno gestite come standard)

    COMPILABILE CON BORLAND C++ 3.0

        bcc -O -d -c -mx isremote.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline

#include <dos.h>                                           // per geninterrupt()

int pascal __IOerror(int dosErr);

int cdecl isDriveRemote(int driveNum)
{
    _BL = (char)driveNum;
    _AX = 0x4409;
    geninterrupt(0x21);
    asm jc JOB_ERROR;                       // Se CarryFlag = 1 c'e' stato un errore
    asm and dx,01000000000000b;
    asm mov ax,dx;
    return(_AX);
JOB_ERROR:
    return(__IOerror(_AX));
}
```

La funzione `isDriveRemote()` restituisce `-1` in caso di errore, `0` se il drive è locale, `1` se è remoto. Il parametro è un intero che esprime il drive su cui effettuare il test (`0` indica il drive di default, `1` indica il drive `A:`, `2` il drive `B:`, etc.). Quando la funzione restituisce `-1` è possibile chiamare `perror()` per scrivere su stderr la descrizione dell'errore verificatosi:

```
    if(isDriveRemote(0) == -1)
        perror("What a pity");
```

E' immediato constatare che il comportamento di `isDriveRemote()` è conforme a quello delle altre funzioni di libreria che interagiscono con il DOS. Per altri esempi di utilizzo della `__IOerror()` in funzioni di libreria, vedere pag. 472.

UN ESEMPIO DI... DISINFESTANTE

Chi non si è ancora imbattuto in qualche mutante di... virus informatico? A titolo di esempio riportiamo e commentiamo il listato C di una semplice utility in grado di individuare nei programmi il virus Vienna B e di "risanarli".

Un programma colpito dal Vienna B è riconoscibile in base alle seguenti caratteristiche:

1. è un file `.COM`
2. la sua dimensione è maggiore per 648 byte (codice del virus) rispetto al normale
3. in questa "appendice" vi è la stringa "`*.COM`" preceduta da una sequenza di 3 byte identici ai primi 3 byte del file
4. i 3 byte che precedono tale sequenza erano, prima del contagio, i primi 3 byte del file stesso

L'anti-Vienna deve pertanto aprire il file sospetto, leggerne i primi 3 byte e concatenare a questi la stringa "`*.COM`" per ricostruire l'identificatore del virus. Esso deve poi leggere gli ultimi 648 byte del file in un buffer appositamente allocato e scandirli alla ricerca della stringa identificatrice composta in precedenza. Se essa viene localizzata, allora il file analizzato ha subìto il contagio. Per risanarlo è sufficiente sovrascrivere i suoi primi 3 byte con i 3 byte che, nel buffer, precedono la stringa identificatrice e troncarlo ad una dimensione pari alla lunghezza attuale diminuita di 648.

Vediamone il listato.

```
/********************

    BARNINGA_Z! - 1990

    DISINFES.C - Elimina il virus Vienna (versione B) dai file contagiati

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d disinfes.c wildargs.obj

********************/
#pragma warn -pia               /* no warnings per assegnazioni all'interno di if */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <io.h>

#define  _VIRUSDIM_  ((size_t)648)                         /*  dimensione del virus */
#define  _S_STRING_  ((unsigned char *)"   *.COM")                  /* da cercare */
#define  _S_BYTES_   3                             /* n. bytes da sostituire */
#define  _O_BYTES_   -3                /* offset dei bytes rispetto alla stringa */

unsigned char *sstring = _S_STRING_;
```

```
char *msg[] = {
    "%s non è contaminato.\n",
    "%s decontaminato.\n",
    "\nDISINFES : Elimina virus Vienna B - Barninga_Z!, 1990.\n",
    "USO: disinfes nome_file(s) (sono consentiti * e ?)\n\n",
    "Impossibile aprire %s.\n",
    "Errore di allocazione della memoria.\n",
    "Impossibile decontaminare %s.\n",
    "Errore nella gestione di %s.\n",
    "Impossibile chiudere %s dopo la decontaminazione.\n",
    "Impossibile chiudere %s.\n",
    "ATTENZIONE: è contaminato.\a\n",
    "\nFile: %s:\n"
};

unsigned char *srchstr(unsigned char *bufptr,unsigned char *sstring)
{
    register int i, sl;
    unsigned char *stop;

    stop = bufptr+_VIRUSDIM_-(sl = strlen((char *)sstring));
    for(; bufptr < stop; bufptr++) {
        for(i = 0; i < sl; i++)
            if(*(bufptr+i) != *(sstring+i))
                break;
        if(i == sl)
            return(bufptr);
    }
    return(NULL);
}

int tronca(FILE *file,unsigned char *bufptr,int flen)
{
    if(fread(sstring,(size_t)1,_S_BYTES_,file) != _S_BYTES_)
        return(7);
    if(fseek(file,(long)flen,SEEK_SET) != 0)
        return(7);
    if(fread(bufptr,(size_t)1,_VIRUSDIM_,file) != _VIRUSDIM_)
        return(7);
    if(bufptr = srchstr(bufptr,sstring)) {
        bufptr += _O_BYTES_;
        printf(msg[10]);
        rewind(file);
        if(fwrite(bufptr,(size_t)1,_S_BYTES_,file) < _S_BYTES_)
            return(7);
        if(chsize(fileno(file),(long)flen) != 0)
            return(6);
        return(1);
    }
    return(0);
}

int disinfesta(char *fname,unsigned char *bufptr)
{
    register int ret = 0;
    int flen;
    FILE *file;

    printf(msg[11],fname);
    if(!(file = fopen(fname,"rb+")))
        return(4);
    if((flen = (int)filelength(fileno(file))-_VIRUSDIM_) > 0)
        ret = tronca(file,bufptr,flen);
```

```
    if(fclose(file))
        if(!ret)
            ret = 9;
        else
            ret = 8;
    return(ret);
}

void main(int argc,char **argv)
{
    unsigned char *bufptr;

    printf(msg[2]);
    if(argc < 2)
        printf(msg[3]);
    else
        if(!(bufptr = (unsigned char *)malloc(_VIRUSDIM_)))
            printf(msg[5]);
        else
            for(--argc; argc; argc--)
                printf(msg[disinfesta(argv[argc],bufptr)],argv[argc]);
}
```

La funzione `main()` provvede a visualizzare un opportuno messaggio qualora il programma sia lanciato senza specificare sulla command line alcun nome di file. Se `DISINFES` è consolidato con `WILDARGS.OBJ` è possibile utilizzare le wildcard "`?`" e "`*`" nei nomi di file (vedere pag. 477); `main()` provvede inoltre ad allocare il buffer necessario ai 648 byte letti dal file analizzato e a gestire il ciclo di chiamate (una per ogni file specificato sulla command line) alla funzione `disinfesta()`. Tale ciclo può, a prima vista, apparire di difficile interpretazione; l'algoritmo risulta tuttavia banale se si tiene conto di alcune particolarità. In primo luogo, è noto che la variabile `argc` contiene il numero di elementi presenti nell'array `argv`: per ottenere l'indice massimo di `argv` occorre dunque, inizialmente, decrementare di uno `argc`. Inoltre, dal momento che `argv[0]` è, per default, il pathname completo di `DISINFES`, è necessario escludere detto elemento dal ciclo, il quale si arresta quando `argc` assume valore nullo[413]; al compimento di ogni iterazione `argc` è decrementata, dunque l'analisi procede dall'ultimo file specificato sulla command line di `DISINFES` sino al primo[414]. All'interno del ciclo vi è solamente una chiamata a `printf()`: per ogni file analizzato viene dunque stampato un messaggio individuato, nell'array `msg`, dal valore restituito dalla funzione `disinfesta()`.

Questa apre il file e ne calcola la lunghezza[415], decrementandola di `648` (se esso è contaminato, la variabile `flen` contiene dunque la sua dimensione originaria) e, dopo avere invocato `tronca()`, restituisce un valore atto a individuare l'opportuno messaggio d'errore nell'array `msg` qualora non fosse possibile chiudere il file al ritorno da `tronca()`; in caso contrario, il valore restituito da quest'ultima viene a sua volta restituito a `main()`.

La funzione `tronca()` provvede a leggere i primi 3 byte del file nel buffer `sstring`, che è una variabile (puntatore) globale inizializzata a `_S_STRING_`. L'esame della direttiva

---

[413] Si ricordi che un test del tipo `if(argc)` equivale a `if(argc != 0)`.

[414] I nomi di file sono posti in ordine alfabetico crescente dalle routine contenute in `WILDARGS.OBJ`. L'utilizzo, un po' particolare, di `argc` elimina la necessità di definire una variabile intera con la funzione di contatore.

[415] Dal momento che la dimensione dei files `.COM` non può superare i 64Kb, la variabile `flen` è dichiarata `int`, con il vantaggio di rendere più efficiente la gestione dello stack: tuttavia ciò ha reso necessaria l'operazione di cast nelle chiamate alle funzioni `fseek()` e `chsize()`.

```
#define _S_STRING_ "   *.COM"
```

rivela che sstring contiene, dopo l'operazione di lettura, i primi 3 byte del file seguiti immediatamente da "*.COM": essa può dunque essere utilizzata per identificare il virus. Vengono poi letti nel buffer allocato in main() gli ultimi 648 byte del file analizzato: la funzione srchstr()[416] li scandisce alla ricerca della stringa di identificazione.

L'algoritmo utilizzato da srchstr() è di carattere generale: all'interno di un primo ciclo, che incrementa ad ogni iterazione il puntatore al buffer e valuta, mediante un confronto con un altro puntatore, se il buffer è stato interamente scandito, si trova un secondo ciclo, il quale confronta tra loro i caratteri di buffer e stringa che si trovano all'offset corrispondente all'iterazione attuale e si interrompe se essi sono differenti. Se all'uscita da questo ciclo il contatore è uguale alla lunghezza della stringa ricercata, essa è stata individuata (perché il ciclo non è stato interrotto prima del termine), e il puntatore al buffer referenzia ora, in realtà, la stringa all'interno del buffer medesimo: tale puntatore è restituito a tronca().

Un valore non nullo restituito da srchstr() è altresì, per tronca(), il segnale che il virus è presente nel file analizzato: basta sommare al puntatore bufptr l'offset, rispetto alla stringa, della sequenza di byte che dovrebbe trovarsi in testa al file (attenzione: _O_BYTES_ è definito pari a -3, perciò i conti tornano), riscriverla "al suo posto" e troncare la dimensione del file per completare il ripristino delle sue caratteristiche originarie; l'elaborazione descritta è ripetuta sul successivo file eventualmente specificato sulla riga di comando.

DISINFES.C, il cui impianto logico e tecnico è, come si vede, assai semplice, appare suscettibile di perfezionamenti[417]. Un esame più attento del comportamento del Vienna B, effettuato disassemblando un file "infetto", rivela che i 3 byte da esso scritti in testa ad ogni programma contagiato rappresentano una istruzione di salto JMP (1° byte) seguita (2° e 3° byte) dall'offset al quale viene in effetti trasferita l'elaborazione. In tutti i casi esaminati tale offset è risultato pari alla lunghezza originaria del programma diminuita di 3: se ne trae che il programma contagiato, non appena è eseguito, salta all'indirizzo immediatamente successivo alla fine del proprio codice, cioè all'inizio del codice del virus, il quale, in tal modo, ha la garanzia di poter assumere per primo il controllo della situazione. L'indirizzo ricavato dai 3 byte che originariamente erano in testa al file (JMP e near offset) sono utilizzati per riprendere la normale esecuzione del programma quando Vienna B termina le proprie operazioni.

L'affidabilità di DISINFES potrebbe essere allora accresciuta dotandolo di una funzione che confronti l'integer costituito dal 2° e 3° byte del file con la sua presunta lunghezza originale diminuita di 3:

```
int jmp_test(char *sstring,int flen)
{
    return((*((int *)(sstring+1)))-(flen-_S_BYTES_));
}
```

Si consideri l'espressione (*((int *)(sstring+1))): sommando 1 al puntatore alla stringa utilizzata per identificare il virus se ne "scavalca" il primo byte (l'istruzione JMP); l'operazione di cast forza il compilatore C a considerare puntatore a int l'espressione (sstring+1); l'indirezione restituisce quindi l'intero rappresentato dai 2 byte di cui si è detto. Pertanto, se la funzione jmp_test()

[416] E' stato necessario implementare una funzione apposita, rinunciando ad utilizzare una delle funzioni di libreria per la scansione di stringhe, ad es. strstr(), in quanto queste interrompono l'operazione al primo byte nullo incontrato (terminatore di stringa): si ricordi che il codice del virus non è, di per sé, una stringa, ma una sequenza di byte derivata dall'assemblaggio di un sorgente; è possibile che esso contenga byte nulli, i quali potrebbero interrompere anzitempo la ricerca.

[417] Il programma, scritto in tutta fretta per fronteggiare una situazione di reale emergenza, ha comunque ottenuto buoni risultati, debellando definitivamente il Vienna B in tutti i casi per i quali è stato impiegato.

restituisce un valore non nullo si può concludere che il file esaminato non contiene il virus, pur contenendo la stringa sospetta.

Concludiamo il presente paragrafo presentando la modifica da apportare alla funzione `tronca()` per inserire `jmp_test()` in `DISINFES.C`:

```
....
if((bufptr = srchstr(bufptr,sstring)) && !jmp_test(sstring,flen)) {
    bufptr += _O_BYTES_;
....
```

Il codice di `jmp_test()`, qualora non si intenda dichiararne il prototipo, deve essere inserito nel listato prima della definizione di `tronca()`.

UN ESEMPIO DI... PIRATERIA

Il presente paragrafo intende rappresentare esclusivamente un esempio di come i debugger e il linguaggio C possano essere utilizzati per gestire in profondità l'interazione tra hardware e software; il caso pratico descritto[418] va interpretato esclusivamente come un espediente didattico: non abbiamo alcuna intenzione di incitare, neppure indirettamente, il lettore all'illecito.

Inoltre non è questa la sede per addentrarsi in una descrizione dettagliata delle tecniche utilizzate per proteggere il software dalla duplicazione; basta ricordare che esse si basano spesso su modalità particolari di formattazione del dischetto da proteggere, tali da rendere il medesimo non riproducibile dai programmi `FORMAT` e `DISKCOPY`. Il programma protetto effettua uno o più controlli, di norma accessi in lettura e/o scrittura alle tracce formattate in modo non standard, dai risultati dei quali è in grado di "capire" se il proprio supporto fisico (in altre parole, il disco) è la copia originale oppure ne è un duplicato.

Individuare la strategia di protezione

Veniamo ora al nostro esempio. Il programma in questione, che per comodità indichiamo col nome fittizio di `PROG.COM`, è protetto contro la duplicazione non autorizzata; il comando `DISKCOPY` è in grado di copiare il disco sul quale esso si trova, ma con risultati scadenti: la copia di `PROG.COM`, non appena invocata, visualizza un messaggio di protesta e restitusce il controllo al DOS. Come si può facilmente prevedere, neppure il comando `COPY` è in grado di superare l'ostacolo: copiando il contenuto del disco originale sul disco fisso (o su altro floppy disk) si ottiene un risultato analogo al precedente. Quando viene invocata la copia su hard-disk di `PROG.COM` la spia del drive `A:` si illumina per qualche istante, viene visualizzato il solito messaggio e l'esecuzione si interrompe.

Proviamo a studiare un interessante frammento tratto dal disassemblato di `PROG.COM`, ottenuto mediante il solito `DEBUG` del DOS (i commenti sono stati aggiunti in seguito):

```
CS:0100 EB50          JMP    0136
....
CS:0199 3C20          XOR    AL,AL                                  ; azzera AL
CS:019B 8AD0          MOV    DL,AL                              ; muove AL in DL
CS:019D 32F6          XOR    DH,DH                                  ; azzera DH
CS:019F 8CDB          MOV    BX,DS                              ; muove DS in ES
CS:01A1 8EC3          MOV    ES,BX                                ; attraverso BX
CS:01A3 BBF007        MOV    BX,18FB                                ; carica BX
```

[418] Non si tratta di un caso reale. Il frammento di codice disassemblato qui riportato è stato costruito appositamente ai fini dell'esempio.

```
CS:01A6 B90102        MOV    CX,0201                              ; carica CX
CS:01A9 B80402        MOV    AX,0204                              ; carica AX
CS:01AC CD13          INT    13                               ; chiama int 13h
CS:01AE 720A          JC     01BA                            ; salta se CF = 1
CS:01B0 BABA00        MOV    DX,154F                              ; carica DX
CS:01B3 B409          MOV    AH,09                           ; stampa stringa
CS:01B5 CD21          INT    21                              ;    tramite DOS
CS:01B7 E97DFF        JMP    0156                            ; salta indietro
CS:01BA 80FC04        CMP    AH,04                          ; confr. AH con 4
CS:01BD 75F1          JNE    01B0                               ; salta se !=
CS:01BF B82144        MOV    AX,4421
....
```

Come per tutti i programmi `.COM`, l'entry point[419] si trova alla locazione `CS:0100`: dopo l'istruzione `JMP 0152` l'esecuzione prosegue a `CS:0136`, presumibilmente con le opportune operazioni di inizializzazione, ininfluenti ai nostri fini. Le istruzioni commentate sul listato meritano particolare attenzione. A `CS:01AC` viene richiesto l'int 13h (che gestisce i servizi BIOS relativi ai dischi; vedere pag. 115).

I valori caricati nei registri della CPU rivelano che `PROG.COM`, mediante l'int 13h, legge in memoria i settori `1`, `2`, `3` e `4` della seconda traccia del lato `0` del disco che si trova nel drive `A:`. Al ritorno dall'int 13h (`CS:01AE`) viene effettuato un test sul `CarryFlag`: se questo è nullo, cioè se in fase di lettura non si è verificato alcun errore (comportamento "normale" se il disco è formattato secondo lo standard DOS) l'esecuzione prosegue a `CS:01B1`, è stampata una stringa (il messaggio di protesta) e viene effettuato un salto a ritroso alla locazione `CS:0156`, ove sono effettuate, con ogni probabilità, le operazioni di cleanup e uscita dal programma (infatti il programma termina l'esecuzione subito dopo avere visualizzato il messaggio). Se, al contrario, il `CarryFlag` vale `1` (condizione di errore), l'esecuzione salta a `CS:01BA`, dove `PROG.COM` effettua un controllo sul valore che la chiamata all'int 13h ha restituito in `AH`. Se esso è `04h` (codice di errore per "settore non trovato") l'esecuzione prosegue a `CS:01BF` (controlli superati: il disco è la copia originale); in caso contrario avviene il salto a `CS:01B0`, con le conseguenze già evidenziate.

La strategia è ormai chiara: la traccia `2` del lato `0` della copia originale è formattata in modo non standard. Un tentativo di leggerne alcuni settori determina il verificarsi di una condizione di errore, ed in particolare di "settore non trovato". Se l'int 13h non riporta entrambi questi risultati, allora `PROG.COM` "conclude" che il disco è una copia non autorizzata. Si tratta ora, semplicemente, di trarlo in inganno.

Superare la barriera

Smascherata la strategia di `PROG.COM`, occorre resistere alla tentazione di modificare il codice disassemblato, ad esempio sostituendo una serie di `NOP`[420] alle istruzioni comprese tra `CS:01AD` e `CS:01BE` (significherebbe eliminare l'accesso al disco e tutti i controlli), e riassemblarlo per ottenerne una versione meno "agguerrita": si rischierebbe di incappare in altri trabocchetti[421] e rendersi la vita

---

[419] La prima istruzione che viene eseguita dopo il caricamento in memoria del programma.

[420] *Null OPeration*; istruzione assembler priva di qualunque effetto ed avente il solo scopo di occupare un byte nel file eseguibile.

[421] In effetti, `PROG.COM` potrebbe incorporare una routine per l'effettuazione di una sorta di checksum sul file stesso, al fine di verificare, ad esempio, che la somma dei valori esadecimali dei byte che compongono il codice binario (o almeno la parte di esso che comprende il frammento esaminato) corrisponda ad un valore predeterminato, scoprendo così eventuali modifiche al codice originale.

difficile senza alcuna utilità. E' sicuramente più opportuno simulare il verificarsi delle condizioni di errore ricercate in fase di test: per ottenere tale risultato è sufficiente un programmino in grado di installare un gestore personalizzato dell'int 13h.

Questo deve scoprire se la chiamata proviene da PROG.COM: in tal caso occorre restituire le ormai note condizioni di errore senza neppure accedere al disco; altrimenti è sufficiente concatenare la routine originale di interrupt.

Intercettare la chiamata di PROG.COM è semplice: basta un'occhiata alla tabellina riportata poc'anzi per capire che un test sui registri AX, CX e DX può costituire una "trappola" (quasi) infallibile.

Ecco il listato:

```
/********************

   LOADPROG.C - Barninga_Z! - 1990

       Lancia PROG.COM gestendo opportunamente l'int 13h

   COMPILABILE SOTTO BORLAND C++ 2.0

       bcc -O -d -mt -lt loadprog.c

********************/

#pragma  inline
#pragma  option  -k-                        // fondamentale!!! Evita std stack frame

#include <stdio.h>
#include <process.h>
#include <dos.h>

#define  CHILD_NAME     "PROG.COM"                          // programma da lanciare

char *credit =
"GO-PROG.EXE - Cracking loader for "CHILD_NAME" - Barninga_Z!, 1992\n\n\
Press a key when ready...\a\n";

char *errorP =
"Error while executing "CHILD_NAME"\a";

void oldint13h(void)                // dummy function: puntatore a int 13h originale
{
    asm dd 0;                         // riserva 4 bytes per l'idirizzo dell'int 13h
}

void far newint13h(void)
{
    if(_AX == 0x0204 && _CX == 0x0201 && _DX == 0) {               // PROG.COM chiama
        asm {
            stc;                       // setta il carry per simulare errore lettura
            mov ax,0400h;                        // simula errore "sector not found"
            ret 2;                                   // esce e toglie flags da stack
        }
    }
    else
        asm jmp dword ptr oldint13h;                 // concatena l'int 13h originale
}

void cdecl kbdclear(void)
{
    asm {
        mov ax,0C07h;
        int 21h;                            // vuota buffer tastiera e attende tasto
    }
```

```
}

void cdecl main(void)
{
    puts(credit);
    kbdclear();
    asm cli;
    (void(interrupt *)())*(long far *)oldint13h = getvect(0x13);
    setvect(0x13,(void(interrupt *)())newint13h);
    asm sti;
    if(spawnl(P_WAIT,CHILD_NAME,CHILD_NAME,NULL))                 // esegue PROG.COM
        perror(errorP);                                  // errore load/exec di PROG.COM
    asm cli;
    setvect(0x13,(void(interrupt *)())*(long far *)oldint13h);
    asm sti;
}
```

La struttura di LOADPROG.C è semplice: esso si compone di tre routine, a ciascuna delle quali è affidato un compito particolare.

La funzione main() installa il nuovo vettore dell'int 13h, lancia PROG.COM mediante spawnl() e, al termine dell'esecuzione di PROG.COM, ripristina il vettore originale dell'int 13h. Si noti che spawnl() viene invocata con la costante manifesta (definita in PROCESS.H) P_WAIT: ciò significa che LOADPROG.COM rimane in RAM durante l'esecuzione di PROG.COM (che ne costituisce un child process) in attesa di riprendere il controllo al termine di questo, in quanto è necessario effettuare il ripristino dell'int 13h originale. Se non si prendesse tale precauzione, una chiamata all'int 13h da parte di un programma eseguito successivamente avrebbe conseguenze imprevedibili (e quasi sicuramente disastrose, come al solito).

La funzione newint13h() è il gestore truffaldino dell'int 13h. Essa, in ingresso, controlla se i registri AX, BX e DX contengono i valori utilizzati da PROG.COM nella chiamata all'interrupt: in tal caso viene posto uguale a 1 il CarryFlag, AX a 4 e AL a 0; il controllo ritorna alla routine chiamante senza che sia effettuato alcun accesso al disco. Si noti l'istruzione RET 2, che sostituisce la più consueta IRET. Una chiamata ad interrupt salva automaticamente sullo stack i flag: se newint13h() restituisse il controllo a PROG.COM con una IRET, questa ripristinerebbe in modo altrettanto automatico i flag prelevandoli dallo stack e l'istruzione STC non avrebbe alcun effetto; d'altra parte una RET senza parametro "dimenticherebbe" una word sullo stack, causando probabilmente un crash di sistema[422] (vedere pag. 251 e seguenti). Se, al contrario, i valori di AX, BX e DX non sono quelli cercati, newint13h() concatena il vettore originale saltando all'indirizzo salvato da getvect() nei byte riservati dalla funzione jolly oldint13h()[423], lasciando che tutto proceda come se LOADPROG non esistesse.

La funzione kbdclear() pulisce il buffer della tastiera e attende la pressione di un tasto: maggiori particolari sull'argomento a pag. .

Un'ultima osservazione: la direttiva

```
#pragma option -k-
```

evita che il compilatore generi il codice necessario al mantenimento della standard stack frame (vedere pag.). In assenza di tale opzione sarebbe indispensabile aggiungere l'struzione POP BP prima della RET e della JMP in newint13h(); inoltre si potrebbe eliminare l'istruzione DD 0 in oldint13h() in quanto il codice della funzione occuperebbe di per sé 5 byte (gli opcode corrispondenti alle istruzioni

[422] Il parametro della RET esprime l'incremento di SP in uscita alla funzione: 2 equivale a una word.

[423] Vedere pag. e seguenti.

necessarie alla standard stack frame stessa). La direttiva può essere eliminata qualora si specifichi `-k-` tra le opzioni sulla riga di comando del compilatore.

Sulla retta via...

A cosa può servire un esempio di pirateria? Ovviamente, a fornire spunti utili in situazioni reali (e lecite!). Vi sono programmi, piuttosto datati, che alla partenza modificano vettori di interrupt senza poi ripristinarli in uscita. Se tra i vettori modificati ve ne sono alcuni utilizzati da più recenti software di sistema, il risultato è quasi certamente la necessità di un reset della macchina. Un loader analogo a quello testè presentato può validamente ovviare.

INSODDISFATTI DELLA VOSTRA TASTIERA?

Possiamo almeno tentare di evitare la sostituzione fisica, dal momento che qui si tratta di eliminare un problema spesso fastidioso: nessuna tastiera è configurata per gestire tutti i caratteri di cui si può avere necessità e pertanto, più o meno spesso, si è costretti a ricorrere alle scomode sequenze `ALT+nnn` sul tastierino numerico.

Ridefinire la tastiera

Il programma presentato in queste pagine è in grado di assegnare ai tasti nuovi scan code e ascii code, in sostituzione o in aggiunta a quelli standard. Nella maggior parte dei casi sperimentati, `KBDPLUS` si è rivelato di grande utilità per aggiungere le parentesi graffe e la tilde, care ai programmatori C, alle tastiere italiane e le lettere accentate alle tastiere americane; in un caso ha egregiamente trasformato in tedesca una normale tastiera italiana.

La tabella di ridefinizione dei tasti è contenuta in un normale file ASCII, ed è pertanto riconfigurabile a piacere. La sintassi della tabella è semplice: ogni riga del file ASCII rappresenta una ridefinizione di tasto[424] e deve essere strutturata come descritto di seguito.

```
SPAZI SHIFTS SPAZI SCAN SPAZI NUOVO_SCAN SPAZI ASCII SPAZI COMMENTO
```

[424] Ogni tasto può essere ridefinito più volte (ad ogni ridefinizione corrisponde una riga della tabella): in tal modo si ha la possibilità di associare ad ogni tasto più caratteri, ciascuno dei quali in corrispondenza di una certa combinazione di shift.

spazi	uno o più spazi o tabulazioni. Possono anche non essere presenti.
shifts	uno dei seguenti caratteri o una loro combinazione (senza spazi tra un carattere e l'altro): `A` ALT `C` CTRL `L` LEFT SHIFT `R` RIGHT SHIFT `N` Nessuno shift Lo shift `N` è ignorato se non e' il solo presente.
spazi	uno o più spazi o tabulazioni.
scan	scan code del tasto da ridefinire (2 cifre esadecimali).
spazi	uno o più spazi o tabulazioni.
nuovo_scan	nuovo scan code per il tasto (2 cifre esadecimali). Può essere uguale a SCAN.
spazi	uno o più spazi o tabulazioni.
ascii	ASCII code per il tasto (2 cifre esadecimali).
spazi	uno o più spazi o tabulazioni.
commento	è opzionale e non ha formato particolare.

Si noti che utilizzando un comune editor per scrivere il file, una coppia `CR LF` è aggiunta al termine di ogni riga quando è premuto il tasto `RETURN`. La lunghezza della riga, `CR_LF` compreso, non può superare gli 80 caratteri. Ecco un esempio di riga valida, che ridefinisce la combinazione `CTRL ALT ESC` come `ENTER`:

```
CA 01 1C 0D - Il tasto ESC, premuto con CTRL e ALT, equivale a ENTER.
```

Gli shift sono indicati all'inizio della riga: `CA` significa `CTRL ALT`. Uno spazio separa gli shift dallo scan code del tasto: `01` è il tasto `ESC`. Il nuovo scan code (cioè lo scan code che viene inserito nel buffer di tastiera alla pressione di `CTRL ALT ESC` è `1C` (lo scan code del tasto `RETURN`). Il nuovo codice ASCII da inserire nel buffer è `0D`, corrispondente anch'esso al tasto `RETURN`. Tutto ciò che segue `0D`, sino alla fine della riga, rappresenta un commento ed è ignorato dal programma.

`KBDPLUS` è un programma TSR, che viene installato fornendogli come parametro (sulla command line) il nome del file contenente le ridefinizioni dei tasti; la command line è scandita a partire dall'ultimo parametro ed ognuno di essi (se ve n'è più di uno) è considerato un nome di file. Se il tentativo di apertura fallisce il processo è ripetuto con il parametro che lo precede; se nessun parametro è un nome di file esistente o non vi sono parametri, `KBDPLUS` cerca, nella directory in cui esso stesso si trova, il file `KBDPLUS.DEF`[425]. Se anche questo tentativo fallisce, il programma non si installa e visualizza un testo di aiuto.

[425] Per la precisione, il nome del file di dati è costruito con `argv[0]`: se si rinomina l'eseguibile, il file di dati ricercato ha ancora estensione `.DEF` e nome uguale a quello dell'eseguibile stesso.

KBDPLUS accetta inoltre, dalla command line, due parametri particolari: il carattere '?', che provoca la visualizzazione del testo di help senza tentativo di installazione[426], e il carattere '*', che provoca la disinstallazione del programma (se già installato) e il ripristino delle funzionalità originali della tastiera.

KBDPLUS è in grado di individuare se stesso in RAM ed evita quindi installazioni multiple.

Segue il listato.

```
/************************************************************************

    KBDPLUS.C - KBDPLUS - Barninga_Z! 1991

        Ridefinitore di tastiera.

    Modalita' di funzionamento.
        Legge un file in formato ASCII e in base al suo contenuto ridefinisce
        la tastiera. Ogni riga del file deve essere lunga al massimo 80 caratteri
        ed e' strutturata come segue:

              shf spc scc spc nsc spc asc spc commento crlf

        dove:

        spc.......uno o piu' blanks o tabs. Possono anche essere presenti
                  prima di shifts; in tal caso saranno ignorati.
        shf.......uno dei seguenti caratteri o una loro combinazione (senza
                  spazi tra un carattere e l'altro):
                      A  ALT
                      C  CTRL
                      L  LEFT SHIFT
                      R  RIGHT SHIFT
                      N  Nessuno shift (ignorato se non e' il solo presente).
        scc.......scan code in esadecimale (2 cifre) del tasto da
                  ridefinire.
        nsc.......nuovo scan code in esadecimale (2 cifre) per il tasto.
                  Puo' essere uguale a scancode.
        asc.......codice ascii in esadecimale (2 cifre) per il tasto.
        commento..e' opzionale e non ha formato particolare.
        crlf......sequenza CR LF che chiude ogni riga di testo.

        Esempio di riga che ridefinisce CTRL ALT ESC:

     CA 01 1C 0D - Il tasto ESC, premuto con CTRL e ALT, equivale a ENTER.

        KBDPLUS accetta, alternativamente al nome di file, due parametri:

        KBDPLUS ?   produce la visualizzazione di uno schermo di aiuto;
        KBDPLUS *   provoca la disinstallazione del programma (purche'
                    esso sia gia' presente in RAM).

    Compilato sotto BORLAND C++ 2.0:

        tcc -O -d -rd -k- -Tm2 kbdplus.C

************************************************************************/
#pragma  inline
#pragma  warn -pia
#pragma  option -k-                  /* attenzione: non serve la std stack frame */

#include <dos.h>
```

[426] Il testo è visualizzato anche se KBDPLUS è già residente in RAM.

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

#define  PRG            "KBDPLUS"                           /* nome del programma */
#define  YEAR           "1991"                                           /* anno */
#define  PSPENVOFF      0x2C          /* offset, nel PSP, del ptr all'environment */
#define  ALT_V          ((char)8)
#define  CTRL_V         ((char)4)
#define  LSHF_V         ((char)2)
#define  RSHF_V         ((char)1)
#define  NONE_V         ((char)0)
#define  ALT_C          'A'
#define  CTRL_C         'C'
#define  LSHF_C         'L'
#define  RSHF_C         'R'
#define  NONE_C         'N'
#define  SHFMASK        0x0F         /* shift status mask (elimina NumLock, etc.) */
#define  INKEYPORT      0x60                    /* porta di input della tastiera */
#define  CTRKEYPORT     0x61                /* porta di controllo della tastiera */
#define  ENABLEKBBIT    0x80               /* bit di abilitazione della tastiera */
#define  E_O_I          0x20                        /* segnale di fine interrupt */
#define  I_CTRPORT      0x20               /* porta di controllo degli interrupt */
#define  KBDOFF         0x3FE          /* off di seg implicito nei puntatori - 2 */
#define  MAXLEN         83          /* n. max di carat. in ogni riga del file dati */
#define  BLANK          ' '                           /* spazio bianco (ascii 32) */
#define  ARRDIM         360         /* dimensione della func-array Dummy() (90*4)*/
#define  TSR_TEST       0xAD                                /* HANDSHAKE per TSR */
#define  TSR_YES        0xEDAF           /* se installato risponde cosi' al test */
#define  TSR_INST       0x00             /* serv. 0 int 2Fh. Testa se installato */
#define  TSR_FREE       0x01              /* serv. 1 int 2Fh. Disinstalla il TSR */
#define  DEFEXT         ".DEF"              /* il file di default e' KBDPLUS.DEF */
#define  HELP_REQUEST   '?'              /* opzione cmd line per richiedere help */
#define  FREE_REQUEST   '*'              /* opzione cmd line per disinstall. TSR */

#define  int09h    ((void(interrupt *)())*(((long *)Dummy)+0))           /* off 0 */
#define  int2Fh    ((void(interrupt *)())*(((long *)Dummy)+1))           /* off 4 */
#define  ShiftFlag (*((char far *)*(((int *)Dummy)+4)))               /* offset 8 */
#define  HeadPtr   (*((int far *)*(((int *)Dummy)+5)))               /* offset 10 */
#define  TailPtr   (*((int far *)*(((int *)Dummy)+6)))               /* offset 12 */
#define  StartPtr  (*((int far *)*(((int *)Dummy)+7)))               /* offset 14 */
#define  EndPtr    (*((int far *)*(((int *)Dummy)+8)))               /* offset 16 */
#define  nk        (*(((int *)Dummy)+9))                             /* offset 18 */
#define  ResPSP    (*(((unsigned *)Dummy)+10))                       /* offset 20 */
#define  keys      (((struct KbDef far *)Dummy)+6)                   /* offset 24 */

#define int09h_asm   Dummy                         /* usata nell'inline assembly */
#define int2Fh_asm   Dummy+4                       /* usata nell'inline assembly */
#define ResPSP_asm   Dummy+20                      /* usata nell'inline assembly */

extern unsigned cdecl _heaplen = 8000;          /* riduce ingombro al caricamento */

struct KbDef {                           /* struttura per la gestione della tastiera */
    char shf;                               /* shifts usati per ridefinire il tasto */
    char scan;                                     /* scan code originale del tasto */
    char nwscn;                                        /* nuovo scan code del tasto */
    char nwasc;                                     /* nuovo codice ASCII del tasto */
};

struct ShfFlag {                   /* struttura di definizione dei possibili shifts */
    char kc;                          /* carattere usato per rappresentare lo shift */
    char kv;                                      /* valore dello shift status byte */
} shf[] = {
    {ALT_C,ALT_V},
    {CTRL_C,CTRL_V},
```

```
    {LSHF_C,LSHF_V},
    {RSHF_C,RSHF_V},
    {NONE_C,NONE_V}            /* NONE deve essere l'ultimo item (ultimo .kv = 0) */
};

void Dummy(void);              /* necessaria per JMP in newint09h() e newint16h() */

void far newint09h(void)                                       /* handler int 09h */
{
    int i;

    asm {
        push ax;
        push bx;
        push cx;
        push dx;
        push es;
    }
    for(i = 0; i < nk; i++) {                    /* scandisce tabella ridefinizioni */
        if((ShiftFlag & SHFMASK) == keys[i].shf) {
            asm in al,INKEYPORT;
            asm mov cl,al;              /* salva al; ax e' usato per i puntatori */
            if(_CL == keys[i].scan) {
                TailPtr += 2;
                if(TailPtr == HeadPtr ||
                              (TailPtr == EndPtr && HeadPtr == StartPtr)) {
                    TailPtr -= 2;
                    break;                /* se il buffer e' pieno lascia al BIOS */
                }
                _CH = keys[i].nwscn;
                _CL = keys[i].nwasc;
                *((int far *)((TailPtr)+KBDOFF)) = _CX;          /* tasto --> buf */
                if(TailPtr == EndPtr)                    /* se è in fondo al buffer */
                    TailPtr = StartPtr;                     /* aggiorna i puntatori */
                asm {
                    in al,CTRKEYPORT;            /* legge lo stato della tastiera */
                    mov ah,al;                                      /* lo salva */
                    or al,ENABLEKBBIT;              /* setta il bit "enable kbd" */
                    out CTRKEYPORT,al;           /* lo scrive sulla porta di ctrl */
                    xchg ah,al;        /* riprende lo stato originale della tast. */
                    out CTRKEYPORT,al;                           /* e lo riscrive */
                    mov al,E_O_I;          /* manda il segnale di fine Interrupt */
                    out I_CTRPORT,al;                /* al controllore dell'8259 */
                    jmp _EXIT;
                }
            }
        }
    }
    asm {
        pop es;
        pop dx;
        pop cx;
        pop bx;
        pop ax;
        mov sp,bp;
        pop bp;
        jmp dword ptr int09h_asm;                  /* concatena handler originale */
    }
_EXIT:
    asm {
        pop es;
        pop dx;
        pop cx;
        pop bx;
```

```
        pop ax;
        mov sp,bp;
        pop bp;
        iret;                                    /* ritorna al processo interrotto */
    }
}

void far newint2Fh(void)                                     /* gestore dell'int 2Fh */
{
    asm {
        cmp ah,TSR_TEST;                         /* la chiamata viene da KBDPLUS ? */
        jne _CHAIN;                                                  /* no: salta */
        cmp al,TSR_INST;         /* si: e' richiesto il test di gia' installato ? */
        jne _FREETSR;                          /* no: salta al servizio successivo */
        mov ax,TSR_YES;          /* si: carica AX con la "passwd" per restituirla */
        iret;
    }
_FREETSR:
    asm {
        cmp al,TSR_FREE;                      /* e' richiesta la disinstallazione ? */
        jne _CHAIN;                                                  /* no: salta */
        mov ax,offset Dummy;                     /* carica AX con l'offset di Dummy */
        mov dx,seg Dummy;                     /* carica DX con il segmento di Dummy */
        iret;
    }
_CHAIN:
    asm jmp dword ptr int2Fh_asm;                     /* concatena gestore originale */
}

void Dummy(void)                          /* spazio dati globali e strutture KbDef */
{
    asm {
        dd 0;                                                /* punta all'int 09h */
        dd 0;                                      /* punta all'int 2Fh originale */
        dw 0417h;                                 /* punta allo shift status byte */
        dw 041Ah;                                 /* punta alla testa del kbd buf */
        dw 041Ch;                                  /* punta alla coda del kbd buf */
        dw 0480h;                                 /* punta all'inizio del kbd buf */
        dw 0482h;                                  /* punta alla fine del kbd buf */
        dw 0;                                         /* n. di ridefinizioni (nk) */
        dw 0;                                                     /* PSP del TSR */
        dw 0;                               /*  tappo per sincronizzare gli offset */
        db ARRDIM dup (0);               /* spazio per strutture di template KbDef */
    }
}

char *hlpmsg = "\n\
Type  "PRG" ?  for help; "PRG" *  to uninstall.\n\
DATA FILE DESCRIPTION (if no name given, "PRG" tries for "PRG""DEFEXT"):\n\
Each line in the file redefines a key and has the following format:\n\
\n\
      shifts spaces scan spaces newscan spaces ascii spaces comment crlf\n\
\n\
spaces...one or more blanks (or tabs).\n\
shifts...the combination of shift keys: one or more of the following:\n\
            A  (the ALT key)\n\
            C  (the CTRL key)\n\
            L  (the LEFT SHIFT key)\n\
            R  (the RIGHT SHIFT key)\n\
            N  (no shift has to be pressed with the redefined key)\n\
scan.....the hex scan code of the redefined key.\n\
newscan..the new hex scan code for the redefined key. Can be = <scan>.\n\
ascii....the hex ASCII code of the char for the redefined key.\n\
comment..optional entry; useful to scribble some remarks.\n\
```

```
crlf.....a CR LF seq. (End-of-line; max 80 chrs).\n\
\n\
CA 01 1C 0D   Example that makes CTRL ALT ESC same as the ENTER key.\
";                                                          /* stringa di help */

void release_env(void)                              /* libera environment del TSR */
{
    extern unsigned _envseg;

    asm {
        mov es,_envseg;            /* ...carica in ES l'ind. di segmento dell'env. */
        mov ah,0x49;          /* chiede al DOS di liberare il MCB dell'environment */
        int 0x21;
    }
}

int nibble(char c)                                 /* 2 hex digit string ===> int */
{
    if(c > (char)0x29 && c < (char)0x40)
        return(c-'0');
    return(((c <= 'Z') ? c : c-BLANK)-('A'-10));
}

char *setcodes(char *bufptr,char far *code)           /* legge i codici dal file */
{
    for(; *bufptr <= BLANK;)
        ++bufptr;
    *code = (char)((nibble(*bufptr) << 4) + nibble(*(++bufptr)));
    return(++bufptr);
}

void readdata(FILE *in)            /* legge i codici e prepara tabella in memoria */
{
    char shift[MAXLEN], *bufptr;
    register int i, lim;

    lim = ARRDIM / sizeof(struct KbDef);
    for(; fgets(shift,MAXLEN,in) && nk < lim; nk++) {
        for(bufptr = shift; *bufptr <= BLANK;)
            ++bufptr;
        for(; *bufptr > BLANK;)
            ++bufptr;
        *bufptr = (char)NULL;
        for(i = 0; shf[i].kv; i++)
            if(strchr(shift,shf[i].kc))
                keys[nk].shf |= shf[i].kv;
        bufptr = setcodes(bufptr,&(keys[nk].scan));
        bufptr = setcodes(bufptr,&(keys[nk].nwscn));
        setcodes(bufptr,&(keys[nk].nwasc));
        }
}

int readfile(int argc,char **argv)                        /* apre e chiude il file */
{
    FILE *in;

    strcpy(strrchr(argv[0],'.'),DEFEXT);
    for(; argc;)                  /* ipotesi: ogni parm in cmdline = nome di file */
        if(in = fopen(argv[--argc],"rb")) {
            printf(PRG": found %s. Reading...\n",argv[argc]);
            readdata(in);
            fclose(in);
            break;
        }
```

```
        else
            printf(PRG": %s not found.\n",argv[argc]);
    return(nk);
}

int resparas(void)                  /* calcola paragrafi residenti indispensabili */
{
    return(FP_SEG(keys)+((FP_OFF(keys)+nk*sizeof(struct KbDef))
                                                              >> 4)+1-_psp);
}

int tsrtest(void)                                   /* controlla se gia' residente */
{
    asm {
        mov ah,TSR_TEST;
        mov al,TSR_INST;
        int 0x2F;
        cmp ax,TSR_YES;
        je _RESIDENT;
        xor ax,ax;
    }
_RESIDENT:
    return(_AX);
}

void uninstall(void)                              /* disinstalla KBDPLUS residente */
{
    void far * ResDataPtr;

    asm {
        mov ah,TSR_TEST;
        mov al,TSR_FREE;
        int 0x2F;
        mov word ptr ResDataPtr,ax;         /* DX:AX e' l'ind. della Dummy resid. */
        mov word ptr ResDataPtr+2,dx;               /* e va gestito in back-words */
    }
    setvect(0x09,(void(interrupt *)())(*((long far *)ResDataPtr)));
    setvect(0x2F,(void(interrupt *)())(*(((long far *)ResDataPtr)+1)));
    _ES = *(((unsigned far *)ResDataPtr)+10);
    asm {
        mov ah,0x49;           /* chiede al DOS di liberare il MCB del PSP del TSR */
        int 0x21;
    }
}

void install(void)                                          /* installa TSR KBDPLUS */
{
    asm cli;
    int09h = getvect(0x09);
    int2Fh = getvect(0x2F);
    setvect(0x09,(void(interrupt *)())newint09h);
    setvect(0x2F,(void(interrupt *)())newint2Fh);
    asm sti;
    ResPSP = _psp;
    release_env();
    printf(PRG": Installed... %d key redefinition(s).\n",nk);
    keep(0,resparas());                                                 /* TSR ! */
}

void main(int argc,char **argv)
{
    printf(PRG": Keyboard ReDef Utility - Barninga_Z! "YEAR"\n");
    if(argv[1][0] == HELP_REQUEST)
        printf(PRG": help screen.\n%s",hlpmsg);
```

```
    else
        if(tsrtest())
            if(argv[1][0] == FREE_REQUEST) {
                uninstall();
                printf(PRG": Uninstalled: original keys restored.\n");
            }
            else
                printf(PRG": Already installed.\n%s",hlpmsg);
        else
            if(readfile(argc,argv))
                install();
            else
                printf(PRG": Invalid data file name or lines.\n%s",hlpmsg);
}
```

Il programma installa i gestori dell'int 09h e dell'int 2Fh. L'int 2Fh ha il compito di determinare se `KBDPLUS` è già presente in memoria e collabora alla disintallazione: esso controlla se in ingresso `AH` contiene il valore della costante manifesta `TSR_TEST`, nel qual caso valuta `AL`. Se questo contiene `TSR_INST` (il valore è `0` come suggerito da Microsoft) la routine restituisce in `AX` il valore `TSR_YES`, riconosciuto da `tsrtest()`.

Se invece `AL` contiene `TSR_FREE`, il gestore dell'int 2Fh restituisce a `uninstall()` l'indirizzo far della `Dummy()` residente: `uninstall()` può così ripristinare i vettori e liberare la RAM allocata al PSP del codice residente.

Se `AL` contiene un altro valore, il gestore concatena l'int 2Fh originale.

Se `KBDPLUS` non è residente, `main()` invoca `readfile()`, che costruisce il nome di default per il file dati sostituendo all'estensione del nome del programma (ricavato da `argv[0]`) la stringa `DEFEXT` (`".DEF"`) e tenta, comunque, di aprire tutti i file i cui nomi sono specificati sulla command line, a partire dall'ultimo (ogni parametro è interpretato come nome di file). Se nessuno di essi esiste o non sono stati passati parametri viene aperto il file di default. Se neppure questo esiste `KBDPLUS` visualizza il testo di help e termina.

Se invece un file è stato trovato ed aperto, vengono lette in memoria, una ad una, le righe che lo compongono (o, al massimo, tante righe quante sono le ridefinizioni consentite[427]). La decodifica di ogni riga avviene non appena questa è letta, per evitare di allocare un buffer in grado di contenere tutto il file. Sono scartati gli spazi eventualmente presenti ad inizio riga; tutti i caratteri rappresentanti gli shift sono considerati un'unica sequenza interrotta dal primo blank; gli scan code e il codice ascii sono interpretati come numeri di due cifre esadecimali e convertiti in `int` da `nibble()`, in luogo della quale sarebbe possibile utilizzare `sscanf()`, analoga alla `fscanf()` descritta a pag. 122.

I dati decodificati sono scritti nella tabella alla quale `Dummy()` riserva spazio. La `Dummy()`, funzione fittizia (vedere pag.), contiene anche tutti i dati globali; il suo nome (che è in pratica il puntatore alla funzione) viene all'occorrenza forzato a puntatore ai vari tipi di dato in essa contenuti: alcune macro consentono di "nascondere" dietro a pseudo-nomi di variabili le indirezioni e i cast necessari, a volte complessi.

Lo stratagemma attuato con la `Dummy()` consente di lasciare residente in RAM solo ciò che è indispensabile: `newint09h()`, `newint16h()` e i dati globali (di questi, l'array che contiene le ridefinizioni è l'ultimo, per poterne troncare la parte non utilizzata). In tal modo la funzione `resparas()` può calcolare quanti paragrafi devono essere residenti basandosi sulla differenza tra l'indirizzo della tabella e quello del PSP, alla quale va sommato lo spazio occupato dalle ridefinizioni dei tasti (il tutto arrotondato per eccesso).

I gestori `newint09h()` e `newint2Fh()` sono dichiarati `far` e non `interrupt`: ciò consente di evitare inutili salvataggi automatici di registri non utilizzati e semplifica, nel caso di `newint2Fh()`, la restituzione di un valore alla routine chiamante (vedere pag. 253 e seguenti). Nella

[427] Il loro numero dipende dallo spazio riservato nella `Dummy()`.

`newint09h()` la variabile `i` non può essere dichiarata `register`: ne risulterebbe complicata la gestione dello stack (se `DI` e `SI` sono referenziati nella funzione il compilatore li spinge sullo stack in entrata); di qui l'utilizzo dell'opzione `-rd` in compilazione.

Quando `KBDPLUS` è residente, `newint09h()` è attivata ad ogni pressione o rilascio di tasto: viene scandita la tabella delle ridefinizioni alla ricerca di una combinazione di shift e scan code corrispondenti a quella rilevata sulla tastiera. Se la ricerca ha successo, il nuovo codice ASCII e il nuovo scan code sono inseriti nel buffer della tastiera (con conseguente aggiornamento del puntatore alla coda del buffer[428]) e vengono resettati la tastiera e il controllore degli interrupt. In caso contrario, o se il buffer della tastiera è pieno, viene concatenato l'int 09h originale.

Segue il listato di una versione leggermente più sofisticata del programma `KBDPLUS`; si tratta di una successiva release, che rimuove uno dei principali limiti della versione appena presentata: l'incapacità di distinguere tra loro i due ALT (ALT sinistro e ALT-GR) ed i due CTRL (sinistro e destro) presenti sulla tastiera. Le modifiche al listato sono minime: compaiono le definizioni delle costanti manifeste `ALTGR_V` e `CTRLR_V` (maschere dei bit di shift), nonché `ALTGR_C` e `CTRLR_C` (caratteri da utilizzare per ridefinire un tasto mediante ALT-GR e CTRL destro). Ne risulta, ovviamente, ampliato l'array `shf` di strutture `ShfFlag`. Il riconoscimento del tasto ALT-GR è effettuato nella `newint09h()` mediante un controllo basato sul Keyboard Status Byte (pag. 302), che si trova all'indirizzo `0:0496` (nuova costante manifesta `KBSTBYTE`).

```
/******************************************************************************

    KBDPLUS2.C - KBDPLUS 2.5 - Barninga_Z! 14-06-93

        Ridefinitore di tastiera.

    Modalita' di funzionamento.
        Legge un file in formato ASCII e in base al suo contenuto ridefinisce
        la tastiera. Ogni riga del file deve essere lunga max. 80 caratteri
        ed e' strutturata come segue:

              shf spc scc spc nsc spc asc spc commento crlf

        dove:

        spc.......uno o piu' blanks o tabs. Possono anche essere presenti
                  prima di shifts; in tal caso saranno ignorati.
        shf.......uno dei seguenti caratteri o una loro combinazione (senza
                  spazi tra un carattere e l'altro):
                      A  ALT
                      G  ALT GR
                      C  CTRL
                      T  CTRL RIGHT
                      L  LEFT SHIFT
                      R  RIGHT SHIFT
                      N  Nessuno shift (ignorato se non e' il solo presente).
        scc.......scan code in esadecimale (2 cifre) del tasto da ridefinire.
        nsc.......nuovo scan code in esadecimale (2 cifre) per il tasto. Puo'
                  essere uguale a scancode.
        asc.......codice ascii in esadecimale (2 cifre) per il tasto.
        commento..e' opzionale e non ha formato particolare.
        crlf......sequenza CR LF che chiude ogni riga di testo.

        Esempio di riga che ridefinisce CTRL ALT ESC:
```

[428] La `newint09h()` non inserisce i codici ascii e di scansione nel buffer di tastiera mediante il servizio 05h dell'int 16h in quanto esso è disponibile solo sulle macchine di categoria 80286 o superiore. Sul buffer di tastiera e sull'int 16h si veda a pag. .

```
     CA 01 1C 0D - Il tasto ESC, premuto con CTRL e ALT, equivale a ENTER.

        Se gia' presente in RAM, KBDPLUS si riconosce e non si installa una
        seconda volta.
        KBDPLUS2 accetta, alternativamente al nome di file, due parametri:
        ?: KBDPLUS2 ?   produce la visualizzazione di uno schermo di aiuto;
        *: KBDPLUS2 *   provoca la disinstallazione del programma (purche' esso
                        sia gia' presente in RAM).

    Compilato sotto TURBO C++ 3.1:

        tcc -O -d -rd -k- -Tm2 kbdplus2.c

******************************************************************************/
#pragma  inline
#pragma  warn -pia
#pragma  option -k-                     // no std stack frame!! serve per assembly!

#include <dos.h>
#include <stdio.h>
#include <string.h>

#define  PRG            "KBDPLUS2"                    /* nome del programma */
#define  VER            "2.5"                                   /* release */
#define  YEAR           "1993"                                     /* anno */
#define  PSPENVOFF      0x2C     /* offset, nel PSP, del ptr all'environment */
#define  ALT_V          ((int)520)     // 512 + 8
#define  ALTGR_V        ((int)8)       // gestito con kbd st. byte (0x496)
#define  CTRL_V         ((int)260)     // 256 + 4
#define  CTRLR_V        ((int)4)
#define  LSHF_V         ((int)2)
#define  RSHF_V         ((int)1)
#define  NONE_V         ((int)0)
#define  ALT_C          'A'
#define  ALTGR_C        'G'
#define  CTRL_C         'C'
#define  CTRLR_C        'T'
#define  LSHF_C         'L'
#define  RSHF_C         'R'
#define  NONE_C         'N'
#define  SHFMASK        0x30F      /* shift status mask per entrambi i bytes */
#define  INKEYPORT      0x60                /* porta di input della tastiera */
#define  CTRKEYPORT     0x61            /* porta di controllo della tastiera */
#define  ENABLEKBBIT    0x80           /* bit di abilitazione della tastiera */
#define  E_O_I          0x20                    /* segnale di fine interrupt */
#define  I_CTRPORT      0x20           /* porta di controllo degli interrupt */
#define  KBSTBYTE       0x496           /* puntatore al keyboard status byte */
#define  KBDOFF         0x3FE   /* offset di seg implicito nei puntatori - 2 */
#define  MAXLEN         83    /* n. max di carat. in ogni riga del file dati */
#define  BLANK          ' '                      /* spazio bianco (ascii 32) */
#define  ARRDIM         1280  /* dimensione della func-array Dummy() (256*5) */
#define  TSR_TEST       0xAD                            /* HANDSHAKE per TSR */
#define  TSR_YES        0xEDAF  /* se instal., int 16 risponde cosi' al test */
#define  TSR_INST       0x00         /* serv. 0 int 2Fh. Testa se installato */
#define  TSR_FREE       0x01          /* serv. 1 int 2Fh. Disinstalla il TSR */
#define  DEFEXT         ".DEF"          /* il file di default e' KBDPLUS.DEF */
#define  HELP_REQUEST   '?'          /* opzione cmd line per richiedere help */
#define  FREE_REQUEST   '*'          /* opzione cmd line per disinstall. TSR */

#define  int09h     ((void (interrupt *)())*(((long *)Dummy)+0)) /* offset 0 */
#define  int2Fh     ((void (interrupt *)())*(((long *)Dummy)+1)) /* offset 4 */
#define  ShiftFlag  (*((int far *)*(((int *)Dummy)+4)))          /* offset 8 */
#define  HeadPtr    (*((int far *)*(((int *)Dummy)+5)))         /* offset 10 */
```

```
#define  TailPtr    (*((int far *)*(((int *)Dummy)+6)))         /* offset 12 */
#define  StartPtr   (*((int far *)*(((int *)Dummy)+7)))         /* offset 14 */
#define  EndPtr     (*((int far *)*(((int *)Dummy)+8)))         /* offset 16 */
#define  nk         (*(((int *)Dummy)+9))                       /* offset 18 */
#define  ResPSP     (*(((unsigned *)Dummy)+10))                 /* offset 20 */
#define  keys       ((struct KbDef far *)DummyFkeys)

#define int09h_asm   Dummy                       /* usata nell'inline assembly */
#define int2Fh_asm   Dummy+4                     /* usata nell'inline assembly */
#define ResPSP_asm   Dummy+20                    /* usata nell'inline assembly */

extern unsigned cdecl _heaplen = 8000;

struct KbDef {
    int  shf;
    char scan;
    char nwscn;
    char nwasc;
};

struct ShfFlag {
    char kc;
    int  kv;
} shf[] = {
    {ALT_C,ALT_V},
    {ALTGR_C,ALTGR_V},          // aggiunta 14-06-93
    {CTRL_C,CTRL_V},
    {CTRLR_C,CTRLR_V},          // aggiunta 14-06-93
    {LSHF_C,LSHF_V},
    {RSHF_C,RSHF_V},
    {NONE_C,NONE_V}       /* NONE deve essere l'ultimo item (ultimo .kv = 0) */
};

void Dummy(void);                 /* necessaria per newint09h() e newint16h() */
void DummyFkeys(void);            /* necessaria per newint09h() e newint16h() */

void far newint09h(void)
{
    asm {
        push ax;
        push bx;
        push cx;
        push dx;
        push es;
    }
    for(_SI = 0; _SI < nk; _SI++) {
        if(((*(char far *)KBSTBYTE & keys[_SI].shf) == keys[_SI].shf) ||
                                ((ShiftFlag & SHFMASK) == keys[_SI].shf)) {
            asm in al,INKEYPORT;
            asm mov cl,al;          /* salva al; ax e' usato per i puntatori */
            if(_CL == keys[_SI].scan) {
                TailPtr += 2;
                if(TailPtr == HeadPtr ||
                              (TailPtr == EndPtr && HeadPtr == StartPtr)) {
                    TailPtr -= 2;
                    break;   /* se il buffer e' pieno scarica barile al BIOS */
                }
                _CH = keys[_SI].nwscn;
                _CL = keys[_SI].nwasc;
                *((int far *)((TailPtr)+KBDOFF)) = _CX;
                if(TailPtr == EndPtr)
                    TailPtr = StartPtr;
                asm {
                    in al,CTRKEYPORT;       /* legge lo stato della tastiera */
```

```
                    mov ah,al;                                  /* lo salva */
                    or al,ENABLEKBBIT;          /* setta il bit "enable kbd" */
                    out CTRKEYPORT,al; /* lo scrive sulla porta di controllo */
                    xchg ah,al;   /* riprende lo stato originale della tast. */
                    out CTRKEYPORT,al;                     /* e lo riscrive */
                    mov al,E_O_I;      /* manda il segnale di fine Interrupt */
                    out I_CTRPORT,al;           /* al controllore dell'8259 */
                    jmp _EXIT;
                }
            }
        }
    }
    asm {
        pop es;
        pop dx;
        pop cx;
        pop bx;
        pop ax;
        pop si;
        jmp dword ptr int09h_asm;
    }
_EXIT:
    asm {
        pop es;
        pop dx;
        pop cx;
        pop bx;
        pop ax;
        pop si;
        iret;
    }
}

void far newint2Fh(void)
{
    asm {
        cmp ah,TSR_TEST;                  /* la chiamata viene da KBDPLUS ? */
        jne _CHAIN;                                           /* no: salta */
        cmp al,TSR_INST;    /* si: e' richiesto il test di gia' installato ? */
        jne _FREETSR;                 /* no: salta al servizio successivo */
        mov ax,TSR_YES;   /* si: carica AX con la "password" per restituirla */
        iret;
    }
_FREETSR:
    asm {
        cmp al,TSR_FREE;              /* e' richiesta la disinstallazione ? */
        jne _CHAIN;                                           /* no: salta */
        mov ax,offset Dummy;             /* carica AX con l'offset di Dummy */
        mov dx,seg Dummy;              /* carica DX con il segmento di Dummy */
        iret;
    }
_CHAIN:
    asm jmp dword ptr int2Fh_asm;             /* concatena gestore originale */
}

void Dummy(void)                    /* spazio dati globali e strutture KbDef */
{
    asm {
        dd 0;                                                /* ptr int 09 */
        dd 0;                                /* punta all'int 2Fh originale */
        dw 0417h;                           /* punta allo shift status byte */
        dw 041Ah;                                      /* punta alla testa */
        dw 041Ch;                                       /* punta alla coda */
        dw 0480h;                                      /* punta all'inizio */
```

```
        dw 0482h;                                          /* punta alla fine */
        dw 0;                                   /* n. di ridefinizioni (nk) */
        dw 0;                                                  /* PSP del TSR */
    }
}

void DummyFkeys(void)
{
    asm {
        db ARRDIM dup (0);          /* spazio per strutture di template KbDef */
    }
}

char *hlpmsg = "\n\
Type  KBDPLUS ?  for help; KBDPLUS *  to uninstall.\n\
DATA FILE DESCRIPTION (if no name given, "PRG" searches for "PRG""DEFEXT"):\n\
Each line in the file redefines a key and has the following format:\n\
\n\
        shifts spaces scan spaces newscan spaces ascii spaces comment\n\
\n\
spaces...one or more blanks (or tabs).\n\
shifts...the combination of shift keys: one or more of the following:\n\
            A  (the ALT key)\n\
            G  (the ALT GR key)\n\
            C  (the LEFT CTRL key)\n\
            T  (the RIGHT CTRL key)\n\
            L  (the LEFT SHIFT key)\n\
            R  (the RIGHT SHIFT key)\n\
            N  (no shift has to be pressed with the redefined key)\n\
scan.....the hex scan code of the redefined key.\n\
newscan..the new hex scan code for the redefined key. It can be = <scan>.\n\
ascii....the hex ASCII code of the char for the redefined key.\n\
comment..optional entry; useful to scribble some remarks.\n\
\n\
CA 01 1C 0D   Example that makes CTRL ALT ESC equivalent to the ENTER key.";

void release_env(void)
{
    extern unsigned _envseg;

    asm {
        mov es,_envseg; /* ...carica in ES l'ind. di segmento dell'environm. */
        mov ah,0x49;  /* richiede al DOS di liberare il MCB dell'environment */
        int 0x21;
    }
}

int nibble(char c)                                /* hex digit string ===> int */
{
    if(c > (char)0x29 && c < (char)0x40)
        return(c-'0');
    return(((c <= 'Z') ? c : c-BLANK)-('A'-10));
}

char *setcodes(char *bufptr,char far *code)
{
    for(; *bufptr <= BLANK;)
        ++bufptr;
    *code = (char)((nibble(*bufptr) << 4) + nibble(*(++bufptr)));
    return(++bufptr);
}

void readdata(FILE *in)
{
```

```
    char shift[MAXLEN], *bufptr;
    register int i, lim;

    lim = ARRDIM / sizeof(struct KbDef);
    for(; fgets(shift,MAXLEN,in) && nk < lim; nk++) {
        for(bufptr = shift; *bufptr <= BLANK;)
            ++bufptr;
        for(; *bufptr > BLANK;)
            ++bufptr;
        *bufptr = (char)NULL;
        for(i = 0; shf[i].kv; i++)
            if(strchr(shift,shf[i].kc))
                keys[nk].shf |= shf[i].kv;
        bufptr = setcodes(bufptr,&(keys[nk].scan));
        bufptr = setcodes(bufptr,&(keys[nk].nwscn));
        (void)setcodes(bufptr,&(keys[nk].nwasc));
        }
}

int readfile(int argc,char **argv)
{
    FILE *in;

    strcpy(strrchr(argv[0],'.'),DEFEXT);
    for(; argc;)
        if(in = fopen(argv[--argc],"rb")) {
            (void)printf(PRG": found %s. Reading...\n",argv[argc]);
            readdata(in);
            (void)fclose(in);
            break;
        }
        else
            (void)printf(PRG": %s not found.\n",argv[argc]);
    return(nk);
}

int resparas(void)
{
    return(FP_SEG(keys)+((FP_OFF(keys)+nk*sizeof(struct KbDef)) >> 4)+1-_psp);
}

int tsrtest(void)
{
    asm {
        mov ah,TSR_TEST;
        mov al,TSR_INST;
        int 0x2F;
        cmp ax,TSR_YES;
        je _RESIDENT;
        xor ax,ax;
    }
_RESIDENT:
    return(_AX);
}

void uninstall(void)
{
    void far * ResDataPtr;

    asm {
        mov ah,TSR_TEST;
        mov al,TSR_FREE;
        int 0x2F;
        mov word ptr ResDataPtr,ax; /* DX:AX e' l'ind. della Dummy residente */
```

```
        mov word ptr ResDataPtr+2,dx;         /* e va trattato in back-words */
    }
    setvect(0x09,(void(interrupt *)())(*((long far *)ResDataPtr)));
    setvect(0x2F,(void(interrupt *)())(*(((long far *)ResDataPtr)+1)));
    _ES = *(((unsigned far *)ResDataPtr)+10);
    asm {
        mov ah,0x49;   /* richiede al DOS di liberare il MCB del PSP del TSR */
        int 0x21;
    }
}

void install(void)
{
    asm cli;
    int09h = getvect(0x09);
    int2Fh = getvect(0x2F);
    setvect(0x09,(void(interrupt *)())newint09h);
    setvect(0x2F,(void(interrupt *)())newint2Fh);
    asm sti;
    ResPSP = _psp;
    release_env();
    (void)printf(PRG": Installed... %d key redefinition(s).\n",nk);
    keep(0,resparas());                                         /* TSR ! */
}

void main(int argc,char **argv)
{
    (void)printf(PRG" "VER": Keyboard ReDef Utility - Barninga_Z! "YEAR"\n");
    if(argv[1][0] == HELP_REQUEST)
        (void)printf(PRG": help screen.\n%s",hlpmsg);
    else
        if(tsrtest())
            if(argv[1][0] == FREE_REQUEST) {
                uninstall();
                (void)printf(PRG": Uninstalled: original keys restored.\n");
            }
            else
                (void)printf(PRG": Already installed.\n%s",hlpmsg);
        else
            if(readfile(argc,argv))
                install();
            else
                (void)printf(PRG": Invalid data file name or lines.\n%s",hlpmsg);
}
```

Sono state introdotte anche alcune modifiche volte ad incrementare l'efficienza complessiva del programma: in particolare, nella `newint09h()` il ciclo di scansione della tabella dei tasti ridefiniti è gestito esplicitamente mediante il registro `SI` (nella precedente versione era utilizzata una variabile automatica allocata nello stack). Detta tabella, infine, è stata scorporata dalla funzione jolly `Dummy()` e lo spazio ad essa necessario è ora riservato mediante una seconda funzione fittizia, la `dummyFkeys()`; è stata inoltre coerentemente modificata la definizione della costante manifesta `keys`.

Una utility

A corredo di `KBDPLUS` presentiamo un semplice programma in grado di visualizzare, nei formati decimale ed esadecimale, lo scan code ed il codice ASCII di ogni tasto premuto. `KBDCODES` può risultare utile nella preparazione della tabella di ridefinizioni da utilizzare con `KBDPLUS`.

```
/************************************************************************

    KBDCODES.C - Barninga_Z! - 1991

        Visualizza scan code e ascii code del tasto premuto. Per uscire
        basta premere ESC due volte di seguito.

    Compilato sotto BORLAND C++ 2.0

        tcc -O -d -mt -lt kbdcodes.c

************************************************************************/
#include <stdio.h>

#define  ESC      ((char)0x1B)
#define  NORMAL   ((char)0x00)
#define  EXTENDED ((char)0x10)
#define  NORMOPT  '-'
#define  EXTOPT   '+'
#define  YRLIMIT  ((char)0x05)
#define  YRADDR   ((char far *)0xF000FFFCL)

char *msg[] ={"\
KEYBCODE 1.0 : Keyboard Codes : Barninga_Z! : Torino, 14/04/1991\n\n\
Current setting is %s bios service:\n\
KBDCODES +   forces extended bios service;\n\
KBDCODES -   forces normal bios service.\n\n\
Press keys to see ScanCodes and AsciiCodes;\n\
Press ESC twice to exit.\n\n",
"\
ScanCode  = %02Xh (%3d);  AsciiCode = %02Xh (%3d)\n",
"\
normal",
"\
extended"
};

void main(int argc,char **argv)
{
    unsigned char scan, ascii, service = NORMAL;
    register count = 2;
    char far *BiosYear = YRADDR;

    if(*BiosYear > YRLIMIT) {
        service = EXTENDED;
        count = 3;
    }
    if(argc > 1)
        if(*argv[1] == NORMOPT) {
            service = NORMAL;
            count = 2;
        }
        else
            if(*argv[1] == EXTOPT) {
                service = EXTENDED;
                count = 3;
            }
    printf(msg[0],msg[count]);
    for(count = 0; ; count < 2) {
        _AH = service;
        asm int 16h;
        scan = _AH;
        ascii = _AL;
        printf(msg[1],(int)scan,(int)scan,(int)ascii,(int)ascii);
```

```
        if(ascii == ESC)
            ++count;
        else
            count = 0;
    }
}
```

KBDCODES si basa sull'int 16h, di cui invoca il servizio 10h (tastiera estesa) se il bios della macchina è datato 1986 o piu' recente, altrimenti usa il servizio 00h. E' possibile forzare l'uso del servizio 10h invocando il programma con il parametro '+' sulla command line; il parametro '-', al contrario, forza l'utilizzo del servizio 00h. Per uscire dal programma è sufficiente premere il tasto ESC due volte consecutive oppure la sequenza CTRL-BREAK.

VUOTARE IL BUFFER DELLA TASTIERA

Molte applicazioni che richiedono all'utilizzatore l'immissione di dati dalla tastiera necessitano che il buffer della tastiera venga vuotato prima della richiesta di input: si evita in tal modo che tasti battuti, ad esempio, durante durante una fase elaborativa e interpretati al momento sbagliato producano risultati errati o un comportamento imprevedibile del programma. Lo svuotamento del buffer di tastiera può essere effettuato in diversi modi: uno di questi è implementare un ciclo utilizzante il servizio 01h dell'int 16h (vedere pag. e seguenti) per controllare se nel buffer c'è un tasto "in attesa"; in caso affermativo questo viene rimosso mediante il servizio 00h. L'uscita dal ciclo avviene quando il servizio 01h segnala che il buffer è vuoto.

Un altro metodo consiste nell'utilizzare la funzione 0Ch dell'int 21h, appositamente prevista per pulire il buffer della tastiera e invocare una delle funzioni 01h, 06h, 07h, 08h, 0Ah: se si desidera esclusivamente vuotare il buffer è sufficiente indicare 06h quale valore di AL (funzione) e FFh quale valore di DL (input per la funzione)[429]. Le caratteristiche della funzione 0Ch dell'in 21h sono le seguenti:

[429] La funzione di libreria bdos() (vedere pag. 115) è quel che ci vuole per applicare il metodo descritto:

```
....
bdos(0x0C,0xFF,0x06);
....
```

Per completezza aggiungiamo che il servizio 06h legge un carattere dallo standard input se il registro DL contiene FFh, altrimenti essa scrive sullo standard output il carattere rappresentato dal valore di DL; con la funzione 07h, che attende la pressione di un tasto, è possibile realizzare una rudimentale imitazione di getch() (che, d'altra parte, la utilizza) in grado di pulire il buffer della tastiera:

```
....
(char)bdos(0x0C,0x00,0x07);
....
```

La bdos() restituisce 0 se è stato premuto un tasto speciale (tasti funzione, etc.). Per conoscerne il codice è sufficiente invocare una seconda volta la funzione 07h.

```
....
(char)bdos(0x07,0x00,0x00);
....
```

INT 21H, SERV. 0CH: PULISCE IL BUFFER DI TASTIERA E INVOCA UN SERVIZIO

Input	AH	0Ch
	AL	servizio da invocare (può essere 01h, 06h, 07h, 08, 0Ah)
	Altri	Altri registri: valori eventualmente richiesti dalla funzione indicata in AL.
Output	AL	il carattere in input (eccetto funzione 0Ah)

Il terzo metodo utilizza i puntatori alla testa e alla coda del buffer (vedere, anche in questo caso, pag. e seguenti):

```
/********************

    BARNINGA_Z! - 1990

    CLEARKBD.C - clearkbd()

    void cdecl clearkbd(void);

    COMPILABILE CON TURBO C++ 1.0

        tcc -O -d -c -mx clearkbd.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

void cdecl clearkbd(void)
{
    *((int far *)0x41A) = *((int far *)0x41C);
}
```

Il meccanismo è banale: il puntatore alla testa e quello alla coda del buffer si trovano agli indirizzi `0:041A` e `0:041C`, rispettivamente. Tali indirizzi possono pertanto essere considerati i puntatori a detti puntatori: la loro indirezione ne restituisce i valori. La funzione `clearkbd()`, quindi, non fa altro che rendere uguali il puntatore alla testa e quello alla coda, forzando così la condizione di buffer vuoto.

CATTURARE IL CONTENUTO DEL VIDEO

In questo esempio presentiamo un programma TSR che consente di scrivere in un file specificato dall'utente il contenuto del video (in modo testo) quando vengano premuti contemporaneamente i due tasti di shift. Il testo del buffer video è aggiunto ai dati eventualmente già presenti nel file indicato; al termine di ogni riga (e prima di quella iniziale) è aggiunta una sequenza `CR LF`; in coda al buffer è inserito un carattere ASCII `12` (`FormFeed` o salto pagina), rendendo in tal modo il tutto particolarmente idoneo al successivo editing mediante programmi di videoscrittura.

```
/********************

   Barninga_Z! - 1991

   SHFVWRIT.C - TSR che scrive su un file il buffer video CGA, EGA,
```

```
                VGA in modo testo formattato mediante l'aggiunta di
                CR+LF in testa e a fine riga, e di FF in coda. Il nome
                del file deve essere specificato sulla command line.

                I bit del byte attributo (eccetto il bit del blink)
                sono invertiti per segnalare lo svolgimento
                dell'operazione e sono nuovamente invertiti al termine
                della scrittura su file (effettuata in append). Se
                l'applicazione interrotta "ridisegna" una parte del
                video durante la scrittura, i bit dei bytes attibuto
                di tale porzione del video non vengono invertiti al
                termine dell'operazione. Il TSR si attiva premendo
                contemporaneamente i due SHIFT.

        Compilato sotto TURBO C++ 1.01

                    tcc -O -d -rd -k- -Tm2 shfvwrit.c

********************/
#pragma  inline
#pragma  option -k-

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

#define PRG                   "SHFVWRIT"
#define YEAR                  "1991"
#define _BLANK_               ((char)32)
#define _FF_                  ((char)12)
#define _LF_                  ((char)10)
#define _CR_                  ((char)13)
#define _NROW_                25
#define _NCOL_                80
#define _VBYTES_              _NROW_*_NCOL_
#define _NCR_                 (_NROW_+1)
#define _NLF_                 _NCR_
#define _NFF_                 1
#define _BUFDIM_              (_VBYTES_+_NCR_+_NLF_+_NFF_)
#define _MONO_VIDADDR_        ((char far *)0xB0000000L)
#define _COLR_VIDADDR_        ((char far *)0xB8000000L)
#define _SHFMASK_             ((char)3)
#define _MASK_                ((char)127)
#define _TSR_TEST_            0x97                      /* shfvwrit e' residente ? */
#define _TSR_YES_             0xABFE                /* risposta = si, e' residente */
#define _PSPENVOFF_           0x2C               /* off del seg ptr all'env.in psp */
#define _FNMLEN_              80                /* lungh. max path con NULL finale */

#define  int09h   ((void (interrupt *)())(*((long *)TSRdata)))                /*dd*/
#define  int28h   ((void (interrupt *)())(*(((long *)TSRdata)+1)))            /*dd*/
#define  int2Fh   ((void (interrupt *)())(*(((long *)TSRdata)+2)))            /*dd*/
#define  fnameptr ((char far *)(*(((long *)TSRdata)+3)))                      /*dd*/
#define  strptr   ((char far *)(*(((long *)TSRdata)+4)))                      /*dd*/
#define  shfflag  (*(((int *)TSRdata)+10))                                    /*dw*/
#define  exeflag  (*(((int *)TSRdata)+11))                                    /*dw*/
#define  shfstat  (*((char far *)(*(((int *)TSRdata)+12))))                   /*dw*/
#define  vidstr   (((char *)TSRdata)+26)                                     /*arr*/
#define  vidatr   (((char *)TSRdata)+26+_BUFDIM_)                            /*arr*/
#define  fname    (((char far *)TSRdata)+26+_BUFDIM_+_VBYTES_)               /*arr*/

#define  int09h_asm    TSRdata
#define  int28h_asm    TSRdata+4
#define  int2Fh_asm    TSRdata+8
#define  fnameptr_asm  TSRdata+12
```

```
#define  strptr_asm    TSRdata+16

#define  clear_stack()asm {pop di;pop si;pop ds;pop es;pop dx;pop cx;pop bx;pop ax;}

void TSRdata(void);              /* consente l'uso del nome prima della definiz. */

void filewrit(void)
{
    strptr = vidstr;
    asm {
        push ds;
        mov ah,0x5B;
        xor cx,cx;                                          /* attributo normale */
        lds dx,dword ptr fnameptr_asm;
        int 0x21;                               /* tenta di aprire un nuovo file */
        pop ds;
        mov bx,ax;
        cmp ax,0x50;                            /* se ax=50h il file esiste gia' */
        jne _OPENED;
        push ds;
        mov ax,0x3D01;
        lds dx,dword ptr fnameptr_asm;
        int 0x21;                          /* il file esiste: puo' essere aperto */
        pop ds;
        mov bx,ax;
        mov ax,0x4202;
        xor cx,cx;
        xor dx,dx;
        int 0x21;                             /* va a EOF per effettuare l'append */
    }
_OPENED:
    asm {
        mov cx,_BUFDIM_;
        mov ah,0x40;
        push ds;
        lds dx,dword ptr strptr_asm;
        int 0x21;                                  /* scrive il buffer nel file */
        pop ds;
        mov ah,0x3E;
        int 0x21;                                            /* chiude il file */
    }
}


void vidstrset(void)
{
    register j, i;
    char far *vidptr, far *vidstop;
    char *atrptr;

    atrptr = vidatr;
    asm {
        push bp;
        push sp;
        mov ah,0x0F;
        int 0x10;
        pop sp;
        pop bp;
    }
    switch(_AL) {
        case 2:
        case 3:
            vidptr = _COLR_VIDADDR_;
            break;
```

```
        case 7:
            vidptr = _MONO_VIDADDR_;
            break;
        default:
            return;
    }
    vidstop = (vidptr += (_BH*_VBYTES_));
    strptr = vidstr;
    *strptr++ = _CR_;
    *strptr++ = _LF_;
    for(i = 0;i < _NROW_;i++) {
        for(j = 0;j < _NCOL_;j++,vidptr++) {
            *strptr++ = (*vidptr) ? *vidptr : _BLANK_;
            *atrptr++ = (*(++vidptr) ^= _MASK_);
        }
        *strptr++ = _CR_;
        *strptr++ = _LF_;
    }
    *strptr++ = _FF_;
    filewrit();
    for(;vidptr > vidstop;vidptr--)
        *vidptr = (*(--vidptr) == (*(--atrptr) ^= _MASK_)) ? *vidptr :
                                                                *atrptr;
}


void interrupt newint28h(void)
{
    if(shfflag & (!exeflag)) {
        exeflag = 1;
        vidstrset();
        shfflag = exeflag = 0;
    }
    clear_stack();                    /* macro per uscire da funzione interrupt */
    asm jmp dword ptr int28h_asm;
}


void interrupt newint09h(void)
{
    if(((shfstat & _SHFMASK_) == _SHFMASK_) && (!(shfflag | exeflag)))
        shfflag = 1;
    clear_stack();                    /* macro per uscire da funzione interrupt */
    asm jmp dword ptr int09h_asm;
}


void far newint2Fh(void)               /* non interrupt: non serve salvare regs */
{
    asm {
        cmp al,0;
        jne _CHAIN;
        cmp ah,_TSR_TEST_;
        je _ANSWER;
    }
_CHAIN:
    asm jmp dword ptr int2Fh_asm;
_ANSWER:
    asm {
        mov ax,_TSR_YES_;
        iret;
    }
}
```

```
void TSRdata(void)
{
    asm {
        dd 0;                                                    /* ptr int 09h */
        dd 0;                                                    /* ptr int 28h */
        dd 0;                                                    /* ptr int 2Fh */
        dd 0;                                           /* ptr al nome del file */
        dd 0;                         /* ptr all'array contenente copia del video */
        dw 0;                                           /* flag richiesta popup */
        dw 0;                                              /* flag popup attivo */
        dw 0x417;                                     /* ptr shift status byte */
        db _BUFDIM_ dup (0);                   /* buffer dati dal video al file */
        db _VBYTES_ dup (0);                /* buffer bytes-attributo dal video */
        db _FNMLEN_ dup (0);                     /* pathname del file di output */
    }
}


int tsrtest(void)
{
    asm {
        mov ah,_TSR_TEST_;
        xor al,al;
        int 0x2F;
        cmp ax,_TSR_YES_;
        je _RESIDENT;
        xor ax,ax;
    }
_RESIDENT:
    return(_AX);
}


void release_env(void)
{
    asm {
        mov ax,_psp;                                           /* tramite AX... */
        mov es,ax;                /* ...carica in ES l'ind. di segmento del PSP */
        mov bx,_PSPENVOFF_;              /* ES:BX punta all'indirizzo dell'Env. */
        mov ax,es:[bx];             /* salva in AX l'ind di segmento dell'Env. */
        mov es,ax;                                 /* per poi caricarlo in ES */
        mov ah,0x49;              /* libera il blocco di memoria tramite il DOS */
        int 0x21;                                      /* int 21h servizio 49h */
    }
}


void install(void)
{
    asm cli;
    int09h = getvect(0x09);
    int28h = getvect(0x28);
    int2Fh = getvect(0x2F);
    setvect(0x09,newint09h);
    setvect(0x28,newint28h);
    setvect(0x2F,(void (interrupt *)())newint2Fh);
    asm sti;
    release_env();
    _DX = FP_SEG(fnameptr)-_psp+1+((FP_OFF(fnameptr)+_FNMLEN_) >> 4);
    asm {
        mov ax,0x3100;
        int 0x21;
    }
```

```
}

int filetest(char *argv1)
{
    register i;

    fnameptr = fname;
    for(i = 0; argv1[i]; i++)
        fname[i] = argv1[i];
    asm {
        push ds;
        lds si,dword ptr fnameptr_asm;
        les di,dword ptr fnameptr_asm;
        mov ah,0x60;                              /* ricava il pathname completo */
        int 0x21;             /* ATTENZIONE: e' un servizio DOS non documentato!! */
        lds dx,dword ptr fnameptr_asm;
        xor cx,cx;                                          /* attributo normale */
        mov ah,0x5B;
        int 0x21;                                /* tenta di aprire un nuovo file */
        jnc _OPENED_NEW;                            /* file aperto: il nome e' ok */
        cmp ax,0x50;                        /* se AX=50h il file esiste: nome ok */
        je _FNAME_OK;
        jmp _EXITFUNC:
    }
_OPENED_NEW:
    asm {
        mov bx,ax;
        mov ah,0x3E;
        int 0x21;                                              /* chiude il file */
        mov ah,0x41;
        int 0x21;                                              /* e lo cancella */
    }
_FNAME_OK:
    asm xor ax,ax;
_EXITFUNC:
    asm pop ds;
    return(_AX);
}

void main(int argc,char **argv)
{
    (void)puts(PRG" - Barninga_Z! - Torino - "YEAR".");
    if(argc != 2)
        (void)puts(PRG": sintassi: shfvwrit [d:][path]file[.ext]");
    else
        if(tsrtest())
            (void)puts(PRG": già residente in RAM.");
        else
            if(filetest(argv[1]))
                (void)puts(
                      PRG": impossibile aprire il file specificato.");
            else {
                (void)puts(
                 PRG": per attivare premere LeftShift e RightShift.");
                install();
            }
}
```

La struttura del programma non è particolarmente complessa. La `main()` controlla che sia stato fornito, via command line, un nome di file per l'output: in caso negativo l'elaborazione viene interrotta. La funzione `tsrtest()` verifica l'eventuale presenza in RAM del TSR, utilizzando l'int 2Fh

(la tecnica è descritta a pag.). Se il programma non è già residente, `filetest()` controlla la validità del nome di file specificato dall'utente: il servizio 60h dell'int 21h (vedere anche pag.) è utilizzato per ricavarne il pathname completo, onde evitare che variazioni dei default relativi a drive e directory di lavoro determinino la scrittura dei dati in luoghi imprevisti. La `main()` invoca poi `install()`, che completa la fase di installazione: essa, in primo luogo, salva i vettori degli interrupt utilizzati dal programma e sostituisce ad essi quelli dei nuovi gestori; in seguito invoca `release_env()`, che libera la RAM occupata dall'environment fornito dal DOS al TSR (vedere pag.), dal momento che questo non ne fa uso. Infine `install()` calcola il numero di paragrafi che devono rimanere residenti (l'algoritmo di calcolo è descritto a pag.) e installa `SHFVWRIT` mediante l'int 21h, servizio 31h, "cuore" della funzione di libreria `keep()` (pag. 276), qui invocato direttamente per rendere il codice più compatto.

Tutti i dati globali necessari al programma sono gestiti nello spazio ad essi riservato, all'interno del code segment, dalla funzione jolly `TSRdata()`, che chiude il gruppo delle routine residenti. Le macro atte a facilitare i riferimenti a detti dati sono definite, nel sorgente, al termine delle direttive `#define` relative alle costanti manifeste. Disorientati? Niente paura: vedere a pag. .

La funzione `newint2Fh()` è il nuovo gestore dell'int 2Fh: essa è dichiarata `far` in quanto la semplicità della sua struttura rende inutile il salvataggio automatico dei registri. Il concatenamento al gestore originale (nel caso in cui `SHFVWRIT` non sia ancora installato) è effettuato mediante una istruzione `JMP`; il ritorno al processo chiamante (`SHFVWRIT` installato) mediante una `IRET`. In entrambi i casi è indispensabile ripristinare lo stack quale esso appare al momento dell'ingresso nella funzione: a seconda della versione del compilatore può essere necessario, a tal fine, effettuare una `POP` del registro `BP`.

La funzione `newint09h()` è il nuovo gestore dell'int 09h: esso si occupa semplicemente di analizzare il byte di stato degli shift per intercettare lo hotkey. Se l'utente preme i due shift e, al tempo stesso, il TSR non è attivo (`exeflag` = `0`) viene posto a `1` il flag indicante la richiesta di popup, cioè di attivazione. Anche in questo caso il concatenamento al gestore originale è effettuato con una `JMP`; lo stack è ripristinato dalla macro `clear_stack()`, precedentemente definita. La `newint09h()` è dichiarata interrupt per sicurezza. Essa, infatti, gestisce un interrupt hardware che viene invocato da un evento asincrono; inoltre contiene quasi esclusivamente codice C, con la conseguenza che non è possibile sapere a priori quali registri vengano da essa modificati. In questo caso appare prudente (e comodo) lasciare al compilatore il compito di salvare in modo opportuno i registri della CPU.

La `newint28h()` gestisce l'int 28h: essa è pertanto invocata dal DOS quando questo è in attesa di un input dalla tastiera. Se è stato richiesto il popup e il TSR non è attivo, viene posto a `1` il flag che ne indica l'attivazione ed è invocata `vidstrset()`. Al rientro sono azzerati i flag e viene concatenato il gestore originale. Anche `newint28h()` è scritta quasi interamente in linguaggio C, pertanto è dichiarata `interrupt`.

La funzione `vidstrset()` pilota le operazioni di lettura del buffer video e di preparazione alla scrittura nel file. Essa verifica la modalità video attuale mediante il servizio 0Fh dell'int 10h: se è grafica o testo a 40 colonne il controllo è restituito a `newint28h()` senza intraprendere alcuna azione, altrimenti viene opportunamente determinato l'indirizzo del buffer video.

Mediante indirezioni e incrementi di puntatori, il testo contenuto nel buffer è copiato nello spazio ad esso riservato nella `TSRdata()`, inserendo al tempo stesso i caratteri necessari per dare al testo il formato voluto; inoltre gli ASCII `0` sono trasformati in ASCII `32` (spazio). I byte attributo del buffer, e dunque i colori del testo visualizzato, dopo essere stati anch'essi salvati in un array collocato nella `TSRdata()`, sono modificati tramite un'operazione di `XOR` con un byte che funge da maschera: lo scopo è segnalare visivamente che `SHFVWRIT` è in azione. Al termine di queste operazioni è invocata `filewrit()` e al rientro da questa una nuova operazione di `XOR`, identica alla precedente, ripristina i colori originali del testo nelle sole aree di video non modificate da altre applicazioni durante l'attività di `filewrit()`.

Quest'ultima si occupa delle operazioni di output nel file specificato dall'utente: dal momento che il testo deve essere aggiunto al contenuto del file, è necessario che esso sia creato se non esiste, ma non distrutto se è già presente. Il servizio 5Bh dell'int 21h tenta l'apertura di un nuovo file: se questo

esiste l'operazione fallisce (il servizio 3Ch ne avrebbe troncato a `0` la lunghezza) e il file può essere aperto con il servizio 3Dh. Dopo l'operazione di scrittura il file viene chiuso: in tal modo il DOS aggiorna correttamente FAT e directory. La `filewrit()` è scritta quasi per intero in assmbly inline non solo per ragioni di velocità e compattezza del codice, ma anche (e soprattutto) per evitare l'uso di funzioni di libreria[430] e, di conseguenza, i problemi descritti a pag. .

Concludiamo il commento al codice di `SHFVWRIT` evidenziandone le carenze, dovute alla necessità di presentare un esempio di facile comprensione. Le routine residenti non segnalano all'utente il verificarsi di eventuali errori (modalità video non prevista, disco pieno, etc.) e mancano gestori per gli int 23h e 1Bh (`CTRL-C`, `CTRL-BREAK`) e per l'int 24h (errore critico), la gestione dei quali è quindi lasciata all'applicazione interrotta. Il limite più evidente è però l'utilizzo dell'int 28h come punto di attivazione: detto interrupt è infatti generato dal DOS nei loop interni alle funzioni 01h-0Ch dell'int 21h, cioè, in prima approssimazione, quando esso è in attesa di un input da tastiera. Se l'applicazione interrotta non si avvale del DOS per gestire la tastiera ma, piuttosto, del BIOS (int 16h), `newint09()` può ancora registrare nell'apposito flag la richiesta di popup, ma questo non avviene sino al termine dell'applicazione stessa.

Presentiamo di seguito il listato del programma `VIDEOCAP`, che rappresenta una evoluzione di `SHFVWRIT`. In particolare `VIDEOCAP` utilizza ancora quale punto di attivazione l'int 28h, ma incorpora un gestore dell'int 16h (`newint16h()`), il quale simula con un loop sul servizio 01h/11h le richieste di servizio 00h/10h; in tale loop viene invocato un int 28h (se non sono in corso servizi DOS) consentendo così l'attivazione anche sotto programmi che gestiscono la tastiera via int 16h. La routine di copia del buffer video individua il modo video attuale (ammessi testo 80 colonne colore o mono), il numero di righe video attuali e la pagina corrente, risultando così più flessibile e completa di quella incorporata da `SHFVWRIT`. L'offset della pagina video corrente nel buffer video è determinato moltiplicando 160 (80 % 2) per il numero di righe correnti e sommando un correttivo (individuato empiricamente) descritto in una tabella hard-coded, alla quale è riservato spazio dalla funzione fittizia `rowFactors()`.

Si noti che le funzioni fittizie sono dichiarate `DUMMY`, così come il tipo di parametro richiesto. L'identificatore di tipo `DUMMY` è, in realtà, definito e reso equivalente a `void` dalla riga

```
typedef void DUMMY;
```

Si tratta dunque di uno stratagemma volto ad aumentare la leggibilità del programma (lo specificatore `typedef` definisce sinonimi per i tipi di dato; vedere pag. 118).

`VIDEOCAP`, a differenza di `SHFVWRIT`, impiega una funzione fittizia per ogni variabile globale residente: in tal modo è più semplice referenziarle mediante operazioni di cast e si evitano alcune `#define`.

Anche la modalità in cui il programma segnala di avere eseguito il proprio compito è radicalmente diverso: invece di modificare il colore dei caratteri a video durante l'operazione, `VIDEOCAP` emette un breve beep a 2000 Hz non appena chiuso il file. La funzione `beep2000Hz()` esemplifica come pilotare l'altoparlante del PC in una routine residente.

```
/******************************************************************************

    VIDEOCAP.C - Barninga_Z! - 09/11/92

        Utility TSR per salvataggio del video su file. Invocare con un nome
        di file per installare in memoria. Il dump su file si ottiene
        premendo contemporaneamente i due tasti di shift. Per disinstallare
        invocare con un asterisco come parametro sulla command line.

        Compilato sotto BORLAND C++ 3.1:
```

[430] Scrivendo la funzione in puro linguaggio C sarebbe stato necessario utilizzare, al minimo, la `int86x()` e la `segread()`.

```
            bcc -Tm2 -O -d -rd -k- videocap.c

******************************************************************************/
#pragma inline
#pragma warn -pia

#include <stdio.h>                    // la #include <io.h> deve essere DOPO tutte le
#include <dos.h>                        // funzioni DUMMY contenenti asm ... dup(...)
#include <string.h>                       // perche' in io.h e' definita int dup(int)

#define PRG             "VIDEOCAP"
#define VER             "1.0"
#define YEAR            "1992"
#define CRLF            0A0Dh                                          // backwords
#define UNINSTALL       '*'                              // opzione disinstallazione
#define BLANK           32
#define FORMFEED        12
#define _NCOL_          80
#define _MAXROW_        50
#define _BUFDIM_        ((_NCOL_*_MAXROW_)+(2*_MAXROW_)+3)  // b*h+b*CRLF+CRLFFF
#define _MONO_V_SEG_    0B000h
#define _COLOR_V_SEG_   0B800h
#define _SHFMASK_       3
#define _TSR_TEST_      0xA1                        // videocap e' residente ?
#define _TSR_YES_       0xFF16                       // risposta = si, e' residente
#define _FNAMELEN_      81                // lungh. max pathname compreso NULL finale
#define  BADCHARS       ";,:|><"                // caratteri illeciti nei nomi di file

typedef     void       DUMMY;

int pathname(char *path,char *src,char *badchrs);
char far *getInDOSaddr(void);

DUMMY resPSP(DUMMY)
{
    asm dw 0;
}

DUMMY inDosFlagPtr(DUMMY)
{
    asm dd 0;
}

DUMMY old09h(DUMMY)
{
    asm dd 0;
}

DUMMY old16h(DUMMY)
{
    asm dd 0;
}

DUMMY old28h(DUMMY)
{
    asm dd 0;
}

DUMMY old2Fh(DUMMY)
{
    asm dd 0;
}
```

```
DUMMY opReq(DUMMY)
{
    asm db 0;
}

DUMMY inOp(DUMMY)
{
    asm db 0;
}

DUMMY rowFactors(DUMMY)                  // fattori di offset per pagine video su VGA
{
    asm db  48, 12;                        // fattore offset, numero righe (AL,AH) da
    asm db  48, 25;                        // utilizzare per calcolare l'offset della
    asm db 112, 29;                                 // pagina attiva nel buffer video
    asm db  16, 43;
    asm db  96, 50;
    asm db   0,  0;                          // tappo (segnala la fine della tabella)
}

DUMMY fileName(DUMMY)
{
    asm db _FNAMELEN_ dup(0);
}

DUMMY bufVid(DUMMY)
{
    asm db _BUFDIM_ dup(BLANK);
}

#include <io.h>   // definisce dup(int); va incluso DOPO tutte le asm XX dup(Y)

void beep2000Hz(void)
{
    asm in al,61h;                                              // prepara PC speaker
    asm or al,3;
    asm out 61h,al;
    asm mov al,0B6h;
    asm out 43h,al;
    asm mov al,054h;
    asm out 42h,al;
    asm mov al,2;
    asm out 42h,al;                                           // suona a 2000 Hz
    asm mov cx,0FFFFh;
  DELAY:
    asm jmp $ + 2;
    asm loop DELAY;
    asm in al,61h;
    asm and al,0FCh;
    asm out 61h,al;                                             // esclude PC speaker
}

void writebufVid(int rows)
{
    asm push ds;
    asm xor cx,cx;                                               // attributo normale
    asm mov ax,seg fileName;
    asm mov ds,ax;
    asm mov dx,offset fileName;
    asm mov ah,0x5B;
    asm int 0x21;                                   // tenta di aprire un nuovo file
    asm pop ds;
    asm mov bx,ax;
    asm cmp ax,0x50;                                // se ax=50h il file esiste gia'
```

```
    asm jne OPENED;
    asm push ds;
    asm mov ax,seg fileName;
    asm mov ds,ax;
    asm mov dx,offset fileName;
    asm mov ax,0x3D01;
    asm int 0x21;                           // il file esiste: puo' essere aperto
    asm pop ds;
    asm mov bx,ax;
    asm mov ax,0x4202;
    asm xor cx,cx;
    asm xor dx,dx;
    asm int 0x21;                             // va a EOF per effettuare l'append
  OPENED:
    asm push ds;
    asm mov ax,_NCOL_;
    asm mov cx,rows;
    asm mul cl;                                   // AX = AL*CL (colonne * righe)
    asm push ax;
    asm mov ax,2;
    asm mul cl;                              // AX = AL*CL (righe*2; spazio CRLF)
    asm pop cx;
    asm add cx,ax;                          // CX = spazio totale righe * colonne
    asm add cx,3;             // CX = spazio totale con CRLF 1^ riga + FF finale
    asm mov ax,seg bufVid;
    asm mov ds,ax;
    asm mov dx,offset bufVid;
    asm mov ah,0x40;
    asm int 0x21;                                  // scrive il buffer nel file
    asm pop ds;
    asm mov ah,0x3E;
    asm int 0x21;                                              // chiude il file
    asm call _beep2000Hz;
}

int getOffsetByRow(void)
{
    asm push ds;
    asm mov ax,seg rowFactors;
    asm mov ds,ax;
    asm mov si,offset rowFactors;
  NEXT_FACTOR:
    asm lodsw;
    asm cmp ax,0;
    asm je END_OF_TABLE;            // non trovato n.righe in tabella -> offset = 0
    asm cmp ah,cl;
    asm jne NEXT_FACTOR;
    asm xor ah,ah;
    asm mov bx,2;
    asm mul bx;                     // raddoppia offset per contare attributi video
  END_OF_TABLE:
    asm pop ds;
    return(_AX);                                       // AX = offset correttivo
}

int setbufVid(void)
{
    asm push ds;
    asm xor dl,dl;                                    // valore restituito: righe
    asm mov ax,_COLOR_V_SEG_;                          // video se modo video ok;
    asm mov ds,ax;                                                // altrimenti 0
    asm mov ah,0Fh;
    asm int 10h;
    asm push bx;                                      // BH = pagina video attiva
```

```
    asm cmp al,2;
    asm je GETROWS;
    asm cmp al,3;
    asm je GETROWS;
    asm cmp al,7;
    asm je MONO;
    asm pop bx;
    asm jmp EXIT_FUNC;
  MONO:
    asm mov ax,_MONO_V_SEG_;
    asm mov ds,ax;
  GETROWS:
    asm mov ax,1130h;
    asm xor bh,bh;
    asm push bp;
    asm int 10h;
    asm pop bp;
    asm inc dl;                                        // numero righe display
    asm xor ch,ch;
    asm mov cl,dl;
    asm pop bx;
    asm mov bl,bh;
    asm xor bh,bh;
    asm mov ax,_NCOL_ * 2;
    asm push dx;
    asm mul cx;
    asm mul bx;                  // AX = offset in buf. video della pagina attiva
    asm push ax;
    asm push bx;                                                // salva AX e BX
    asm call _getOffsetByRow;              // restituisce AX = offset correttivo
    asm pop bx;                                                // BX = num. pag.
    asm mul bx;                 // offset * num. pagina; AX = totale correttivo
    asm pop bx;                                              // BX = offset base
    asm add ax,bx;                           // AX = offset totale in buf. video
    asm pop dx;
    asm mov si,ax;                                             // DS:SI -> video
    asm mov ax,seg bufVid;
    asm mov es,ax;
    asm mov di,offset bufVid;                                 // ES:DI -> buffer
  NEXTROW:
    asm mov word ptr es:[di],CRLF;
    asm add di,2;
    asm push cx;
    asm mov cx,_NCOL_;
  ROWCOPY:
    asm lodsb;
    asm cmp al,0;                                               // NULL -> BLANK
    asm jne NOT_NULL;
    asm mov al,BLANK;
  NOT_NULL:
    asm stosb;
    asm inc si;                                              // salta attributo
    asm loop ROWCOPY;
    asm pop cx;
    asm loop NEXTROW;
    asm mov word ptr es:[di],CRLF;
    asm add di,2;
    asm mov byte ptr es:[di],FORMFEED;
  EXIT_FUNC:
    asm pop ds;
    return(_DL);
}

void far new09h(void)
```

```
{
    asm push ax;
    asm push bx;
    asm push ds;
    asm cmp byte ptr opReq,0;
    asm jne EXIT_INT;
    asm cmp byte ptr inOp,0;
    asm jne EXIT_INT;
    asm xor ax,ax;
    asm mov ds,ax;
    asm mov bx,0417h;                                   // indir. shift status byte
    asm mov al,byte ptr [bx];
    asm and al,_SHFMASK_;
    asm cmp al,_SHFMASK_;
    asm jne EXIT_INT;
    asm mov byte ptr opReq,1;
  EXIT_INT:
    asm pop ds;
    asm pop bx;
    asm pop ax;
    asm jmp dword ptr old09h;
}

void far new16h(void)
{
    asm sti;
    asm cmp ah,0;
    asm je SERV_0;
    asm cmp ah,10h;
    asm je SERV_0;
    asm cmp ah,1;
    asm je SERV_1;
    asm cmp ah,11h;
    asm je SERV_1:
    asm jmp EXIT_INT;
  SERV_1:
    asm int 28h;
    asm jmp EXIT_INT;
  SERV_0:
    asm push dx;
    asm mov dx,ax;
    asm inc dh;
  LOOP_0:
    asm mov ax,dx;
    asm cli;
    asm pushf;
    asm call dword ptr old16h;
    asm jnz KEY_READY;
    asm push ds;
    asm push bx;
    asm lds bx,dword ptr inDosFlagPtr;
    asm cmp byte ptr [bx],0;
    asm pop bx;
    asm pop ds;
    asm jne LOOP_0;
    asm sti;
    asm int 28h;
    asm jmp LOOP_0;
  KEY_READY:
    asm mov ax,dx;
    asm dec ah;
    asm pop dx;
  EXIT_INT:
    asm jmp dword ptr old16h;
```

```
}

void interrupt new28h(void)
{
    asm cmp byte ptr opReq,0;
    asm je CALL_OLDINT;
    asm cmp byte ptr inOp,0;
    asm jne CALL_OLDINT;
    asm mov byte ptr inOp,1;
    asm call _setbufVid;
    asm cmp ax,0;
    asm je DONE;
    asm push ax;
    asm call _writebufVid;
    asm pop cx;
  DONE:
    asm mov byte ptr inOp,0;
    asm mov byte ptr opReq,0;
  CALL_OLDINT:
    asm pushf;
    asm call dword ptr old28h;
}

void s2Funinstall(void)
{
    asm push ds;
    asm xor ax,ax;
    asm mov es,ax;
    asm mov ax,seg old09h;
    asm mov ds,ax;
    asm mov si,offset old09h;
    asm mov di,0x09 * 4;
    asm mov cx,2;
    asm rep movsw;
    asm mov ax,seg old16h;
    asm mov ds,ax;
    asm mov si,offset old16h;
    asm mov di,0x16 * 4;
    asm mov cx,2;
    asm rep movsw;
    asm mov ax,seg old28h;
    asm mov ds,ax;
    asm mov si,offset old28h;
    asm mov di,0x28 * 4;
    asm mov cx,2;
    asm rep movsw;
    asm mov ax,seg old2Fh;
    asm mov ds,ax;
    asm mov si,offset old2Fh;
    asm mov di,0x2F * 4;
    asm mov cx,2;
    asm rep movsw;
    asm pop ds;
}

void far new2Fh(void)
{
    asm cmp ah,_TSR_TEST_;
    asm je NEXT0;
    asm jmp CHAIN_INT;
  NEXT0:
    asm cmp al,0;
    asm jne NEXT1;
    asm mov ax,_TSR_YES_;
```

```
    asm jmp EXIT_INT;
  NEXT1:
    asm cmp al,UNINSTALL;
    asm jne NEXT2;
    asm call _s2Funinstall;
    asm mov ax,word ptr resPSP;
    asm jmp EXIT_INT;
  NEXT2:
  CHAIN_INT:
    asm jmp dword ptr old2Fh;
  EXIT_INT:
    asm iret;
}

void releaseEnv(void)
{
    extern unsigned _envseg;

    if(freemem(_envseg))
        puts(PRG": impossibile liberare l'environment.");
}

void install(void)
{
    extern unsigned _psp;

    (char far *)*(long far *)inDosFlagPtr = getInDOSaddr();
    *(unsigned far *)resPSP = _psp;
    releaseEnv();
    asm cli;
    (void(interrupt *)(void))*((long far *)old09h) = getvect(0x09);
    (void(interrupt *)(void))*((long far *)old16h) = getvect(0x16);
    (void(interrupt *)(void))*((long far *)old28h) = getvect(0x28);
    (void(interrupt *)(void))*((long far *)old2Fh) = getvect(0x2F);
    setvect(0x09,(void (interrupt *)(void))new09h);
    setvect(0x16,(void (interrupt *)(void))new16h);
    setvect(0x28,new28h);
    setvect(0x2F,(void (interrupt *)(void))new2Fh);
    asm sti;
    puts(PRG": per attivare premere LShift e RShift. "PRG" *  disinstalla.");
    keep(0,FP_SEG(releaseEnv) + (FP_OFF(releaseEnv) / 16) - _psp + 1);
}

int tsrtest(void)
{
    _AL = 0;
    _AH = _TSR_TEST_;
    geninterrupt(0x2F);
    return(_AX == _TSR_YES_);
}

int filetest(char *fname)
{
    FILE *testFile;
    int newFile;
    char path[_FNAMELEN_];

    if(pathname(path,fname,BADCHARS))
        return(1);
    newFile = access(path,0);
    if(!(testFile = fopen(path,"a")))
        return(1);
    fclose(testFile);
    _fstrcpy((char far *)fileName,(char far *)path);
```

```
    if(newFile)
        return(unlink(path));
    return(0);
}

void uninstall(void)
{
    _AH = _TSR_TEST_;
    _AL = UNINSTALL;
    geninterrupt(0x2F);
    if(freemem(_AX))
        puts(PRG": impossibile liberare la memoria allocata.");
    else
        puts(PRG": disinstallato. Vettori ripristinati e RAM liberata.");
}

void main(int argc,char **argv)
{
    puts(PRG" "VER" - Barninga_Z! - Torino - "YEAR".");
    if(argc != 2)
        puts(PRG": sintassi: "PRG" [d:][path]file[.ext] | *");
    else
        if(tsrtest())
            if(*argv[1] == UNINSTALL)
                uninstall();
            else
                puts(PRG": già residente in RAM.");
        else
            if(filetest(argv[1]))
                puts(PRG": impossibile aprire il file specificato.");
            else
                install();
}
```

`VIDEOCAP` chiama due funzioni delle quali, per brvità, il codice comprende esclusivamente i prototipi: si tratta di `pathname()`, utilizzata per ricavare il path completo del file fornito come parametro sulla riga di comando, e di `getInDOSaddr()`, che restituisce l'indirizzo dell'`InDOS Flag` (pag. 192). I listati completi e commentati delle due funzioni si trovano, rispettivamente, a pag. 472 e pag. 295.

Infine, si noti che lo header file `IO.H` è incluso dopo la definizione di tutte le funzioni fittizie, per evitare che il compilatore interpreti come chiamate alla funzione `dup()`, in esso dichiarata, le direttive assembly `DUP` utilizzate per riservare spazio alle variabili globali residenti (vedere pag. 157).

DISINSTALLARE I TSR

Sappiamo che, per un programma TSR, la capacità di disinstallarsi (liberando la RAM allocata e rilasciando i vettori di interrupt agganciati) è una caratteristica utile.

Sappiamo anche, per esperienza, che non tutti i TSR ne sono dotati e pertanto possono essere rimossi dalla memoria esclusivamente con un nuovo bootstrap.

Ecco allora una utility di qualche aiuto: essa è, a sua volta, un TSR, in grado di disinstallare tutti i TSR caricati successivamente.

```
/********************

    Barninga_Z! - 1991

        ZAPTSR.C - TSR in grado di rimuovere dalla ram tutti i TSR
            caricati dopo la propria installazione (nonche' se' medesimo),
```

```
            ripristinando i vettori di interrupt attivi al momento del
            proprio caricamento e liberando tutti i blocchi di memoria
            allocati ad un PSP maggiore del proprio. E' possibile
            installarne piu' copie in RAM; quando si richiede la
            disinstallazione (invocare con un asterisco come parametro
            sulla command line) viene disinstallata l'ultima copia
            installata (criterio L.I.F.O.) e, con essa, tutti i TSR
            caricati successivamente. Passando un piu' (+) sulla command
            line, ZAPTSR si installa allocando a se' stesso tutti i
            blocchi di RAM liberi ad indirizzi minori del proprio.

            STRATEGIA: il numero di copia installata, l'indirizzo di
                segmento del MCB della porzione residente e la tavola
                dei vettori sono conservati nello spazio riservato dalla
                funzione dummy GData(). La comunicazione tra porzione
                residente e porzione transiente e' gestita mediante
                l'int 2Fh. La main() effettua la scelta tra
                disinstallazione e installazione; se ZAPTSR e' gia'
                attivo in RAM viene invocata confirm() per richiedere
                conferma. La func install() gestisce le operazioni
                relative all'installazione; uninstall() quelle relative
                alla disinstallazione.

        Compilato sotto BORLAND C++ 1.01:

            bcc -O -d -rd zaptsr.c

********************/

#pragma  inline
#pragma  warn -pia
#pragma  warn -rvl
#pragma  -k+            // il codice e' scritto per TURBO C++ 1.01 (vedere pag. 173)

#include <stdio.h>
#include <conio.h>
#include <dos.h>

#define  PRG            "ZAPTSR"                          /* nome del programma */
#define  REL            "1.0"                                       /* versione */
#define  YR             "1991"                           /* anno di compilazione */
#define  HEY_YOU        0xB3          /* byte riconoscimento chiamata all'int 2Fh */
#define  HERE_I_AM      0xA1C9               /* risposta per gia' presente in RAM */
#define  HANDSHAKE      0x00                 /* richiesta se gia' presente in RAM */
#define  UNINSTALL      0x01           /* richiesta disinstallazione all'int 2Fh */
#define  UNINST_OPT     '*'                  /* opzione richiesta disinstallazione */
#define  ALLOC_OPT      '+'                /* opz. allocazione blocchi precedenti */
#define  CNT_SPC        2           /* bytes occupati dal numero di copia install. */
#define  MCB_SPC        2           /* bytes occupati dall'ind.seg. del MCB resid. */
#define  VEC_SPC        1024            /* bytes occupati dalla tavola dei vettori */
#define  FALSE          NULL                               /* booleano per falso */
#define  TRUE           (!FALSE)                            /* booleano per vero */
#define  LASTBLK        'Z'                /* segnala ultimo blocco DOS di memoria */

const char *CopyRight = \
PRG": TSRs controller -"REL"- Barninga_Z! "YR".\n\
";
const char *ConfirmMsg = \
PRG": Already active; install copy #%u (Y/N)? \
";
const char *EnvErrMsg = \
PRG": Couldn't release environment block; memory error.\n\
";
const char *InstMsg = \
```

```
PRG": Copy #%u installed; RAM status and vectors table saved.\n\
        NOTE: Installing "PRG" with a plus sign ('+') as command line\n\
              parameter will protect lower MCBs.\n\
        NOTE: All TSRs loaded after this copy of "PRG" (and "PRG" also)\n\
              will be unloaded by invoking "PRG" with an asterisk ('*')\n\
              as command line parameter.\n\
";
const char *UnInstMsg = \
PRG": Copy #%u uninstalled; vectors restored and RAM freed up.\n\
";
const char *UnInstErrMsg = \
PRG": Cannot uninstall; not active in RAM.\n\
        DOS errorlevel set to 1.\n\
";

struct MCB {               /* struttura per la gestione dei Memory Control Blocks */
    char     pos;                         /* 'Z' = ultimo blocco; 'M' = non ultimo */
    unsigned psp;                     /* PSP del programma proprietario del blocco */
    unsigned dim;                            /* dimensione del blocco in paragrafi */
    char     res[3];                                          /* riservato al DOS */
    char     name[8];          /* nome progr. se DOS >= 4.0; altrimenti non usato */
};

void _restorezero(void);                /* non e' dichiarata negli include di bcc */

/********************
    GData(): funzione dummy; lo spazio riservato nel code segment dalle
        db e' utilizzato per memorizzare il numero della copia, l'indir.
        di seg. del MCB della porzione residente e la tavola dei vettori.
        L'opcode dell'istruzione RETF rappresenta un byte (in eccesso) in
        coda allo spazio riservato ai dati globali.
        Se si compilasse (sempre con TC++ 1.0 o successivi) ma senza la
        opzione -k- lo spazio disponibile comprenderebbe 5 bytes in eccesso
        (gli opcodes per la gestione dello stack piu' la RETF) di cui 3 in
        testa. Questi ultimi non rappresenterebbero un problema in quanto i
        dati memorizzati in GData() non devono essere inizializzati dal
        compilatore, ma lo sono a run-time.
********************/

void far GData(void)
{
    asm {
        db CNT_SPC dup(?);                      /* numero della copia installata */
        db MCB_SPC dup(?);          /* ind. di seg. del MCB della parte residente */
        db VEC_SPC dup(?);                                  /* tavola dei vettori */
    }
}

/********************
    new2Fh(): gestore dell'int 2Fh. Utilizzato per le comunicazioni tra
        parte transiente e parte residente. Se non riconosce in AH il
        segnale della parte transiente (HEY_YOU) concatena il gestore
        precedente il cui indirizzo e' prelevato direttamente nella copia
        di tavola dei vettori contenuta in GData(). Altrimenti analizza AL
        per determinare quale servizio e' richiesto dalla porzione
        transiente. Sono riconosciuti due servizi:
        1) HANDSHAKE: la parte transiente richiede se vi e' una copia di
           ZAPTSR gia' attiva in RAM; new2Fh() restituisce in AX la parola
           d'ordine quale risposta affermativa e in DX il numero della
           copia.
        2) UNINSTALL: la parte transiente richiede la disinstallazione
           dell'ultima copia caricata; new2Fh() restituisce in AX:DX
           l'indirizzo della GData() residente. La convenzione generale
           per la restituzione di double words da funzioni è DX:AX,
```

```
           sacrificata in questo caso per ottenere codice piu' compatto
           efficiente.
********************/

void far new2Fh(void)
{
    asm {
        pop bp;
        cmp ah,HEY_YOU;
        jne CHAIN;
        cmp al,HANDSHAKE;
        jne NO_HANDSHAKE;
        push ds;
        mov ax,seg GData;
        mov ds,ax;
        mov bx,offset GData;
        mov dx,ds:word ptr[bx];                           /* DX = numero di copia */
        mov ax,HERE_I_AM;                                 /* AX = parola d'ordine */
        pop ds;
        iret;
    }
NO_HANDSHAKE:
    asm {
        cmp al,UNINSTALL;
        jne NO_UNINSTALL;
        mov ax,seg GData;
        mov dx,offset GData;            /* AX:DX = indirizzo di GData() residente */
        iret;
    }
NO_UNINSTALL:
CHAIN:
    asm jmp dword ptr GData+MCB_SPC+CNT_SPC+(4*2Fh);
}

/********************
    releaseEnv(): libera il blocco di memoria allocato dal DOS alla copia
        di environment creata per ZAPTSR.
********************/

void releaseEnv(void)
{
    extern unsigned _envseg;

    if(freemem(_envseg))
         printf(EnvErrMsg);
}

/********************
    restoredata(): ripristina i vettori di interrupt copiandoli dalla
        GData() residente (di cui ottiene l'indirizzo via int 2Fh) alla
        tavola dei vettori. Restituisce l'indirizzo di segmento del MCB
        della parte residente.
********************/

unsigned restoredata(void)
{
    asm {
        push ds;
        mov al,UNINSTALL;
        mov ah,HEY_YOU;
        int 2Fh;
        cli;
        mov ds,ax;
        mov si,dx;                               /* DS:SI punta a GData residente */
```

```
        add si,CNT_SPC;                                    /* salta il n. di copia */
        mov bx,ds:word ptr [si];
        add si,MCB_SPC;
        xor ax,ax;
        mov es,ax;
        xor di,di;                                /* ES:DI punta a tavola vettori */
        mov cx,512;
        rep movsw;                                   /* ripristina tavola vettori */
        sti;
        pop ds;
    }
    return(_BX);
}

/********************
    getfirstmcb(): ottiene dal DOS l'indirizzo del primo MCB in RAM.
        ATTENZIONE: si basa su un servizio non documentato dell'in 21h
********************/

unsigned getfirstmcb(void)
{
    asm {
        mov ah,52h;
        int 21h;
        mov ax,es:[bx-2];
    }
    return(_AX);
}

/********************
    uninstall(): pilota le oprazioni di disinstallazione. Da restoredata()
        ottiene l'indirizzo di segmento della parte residente; da
        getfirstmcb() ottiene l'indirizzo di segmento del primo MCB nella
        RAM. Questo e' il punto di partenza per un ciclo in cui
        uninstall() libera (azzrandone nel MCB l'indirizzo del PSP
        proprietario) tutti i blocchi appartenenti a PSP residenti ad
        indirizzi successivi a quello del MCB della parte residente.
        Con questa tecnica sfuggono alla disinstallazione i TSR
        eventualmente caricati ad inidirizzi inferiori a quello della
        parte residente, quantunque dopo di essa cronologicamente (evento
        raro, indicatore di problemi di allocazione).
********************/

void uninstall(unsigned cnt)
{
    register resMCB, mcb;

    resMCB = restoredata();
    mcb = getfirstmcb();
    do {
        mcb += ((struct MCB far *)MK_FP(mcb,0))->dim+1;
        if(((struct MCB far *)MK_FP(mcb,0))->psp > resMCB)
            ((struct MCB far *)MK_FP(mcb,0))->psp = 0;
    } while(((struct MCB far *)MK_FP(mcb,0))->pos != LASTBLK);
    printf(UnInstMsg,cnt);
}

/********************
    savedata(): effettua il salvataggio dei dati nella GData(). Sono
        copiati, nell'ordine, il numero della copia di ZAPTSR in fase di
        installazione, l'indirizzo di segmento del MCB del medesimo e la
        tavola dei vettori.
********************/
```

```
void savedata(unsigned cnt)
{
    asm {
        push ds;
        cli;
        cld;
        mov ax,seg GData;
        mov es,ax;
        mov di,offset GData;
        mov ax,cnt;                                        /* salva numero di copia */
        stosw;
        mov ax,_psp;
        dec ax;                                               /* salva segmento MCB */
        stosw;
        xor ax,ax;
        mov ds,ax;
        xor si,si;
        mov cx,512;                                         /* salva tavola vettori */
        rep movsw;
        sti;
        pop ds;
    }
}

/********************
    install(): pilota le operazioni di installazione. Invoca
        _restorezero(), definita nello startup code, per ripristinare i
        vettori eventualmente agganciati dallo startup code medesimo;
        invoca savedata() passandole come parametro il numero della copia
        di ZAPTSR in via di installazione; invoca releaseEnv() per
        disallocare il blocco dell'environment; se e' richiesta l'opzione
        ALLOC_OPT sulla command line install() alloca a ZAPTSR tutti i
        blocchi di RAM liberi aventi indirizzo inferiore a quello di
        ZAPTSR stesso, per evitare che TSRs invocati successivamente
        vi si installino, occultandosi. Infine attiva il gestore
        dell'int 2Fh new2Fh() mediante setvect() e installa ZAPTSR
        chiamando direttamente l'int 21h, servizio 31h (non e' usata
        keep() per evitare una nuova chiamata alla _restorezero()). I
        paragrafi residenti sono calcolati in modo da allocare solo la RAM
        indispensabile a GData() e new2Fh(). Notare che non viene salvato
        il vettore originale dell'int 2Fh con getvect() in quanto esso e'
        comunque presente nella copia della tavola dei vettori generata
        con savedata().
********************/

void install(unsigned cnt,unsigned mem)
{
    register mcb;

    _restorezero();                                  /* ripristina vettori 00h-06h*/
    savedata(cnt);
    releaseEnv();
    if(mem)
        for(mcb = getfirstmcb(); mcb < (_psp-1); mcb +=
                                    ((struct MCB far *)MK_FP(mcb,0))->dim+1)
            if(!((struct MCB far *)MK_FP(mcb,0))->psp)
                ((struct MCB far *)MK_FP(mcb,0))->psp = _psp;
    setvect(0x2F,(void(interrupt *)())new2Fh);
    printf(InstMsg,cnt);
    _DX = FP_SEG(releaseEnv)+FP_OFF(releaseEnv)/16+1-_psp;
    _AX = 0x3100;
    geninterrupt(0x21);
}
```

```
/********************
    AreYouThere(): invoca l'int 2Fh per verificare se e' gia' attiva in
        RAM un'altra copia di ZAPTSR. Solo in questo caso, infatti, in AX
        e' restituito dall'int 2Fh (cioe' dalla new2Fh() residente) il
        valore HERE_I_AM, che funge da parola d'ordine. Allora DX contiene
        il numero della copia di ZAPTSR che ha risposto (l'ultima
        caricata).
********************/

unsigned AreYouThere(void)
{
    _AH = HEY_YOU;
    _AL = HANDSHAKE;
    geninterrupt(0x2F);
    if(_AX == HERE_I_AM)
        return(_DX);
    return(0);
}

/********************
    confirm(): utilizzata per chiedere all'utente la conferma
        dell'intenzione di installare una ulteriore copia di ZAPTSR quando
        ve n'e' gia' una (o piu' di una) attiva in RAM.
********************/

int confirm(unsigned cnt)
{
    int c;

    do {
        printf(ConfirmMsg,cnt);
        printf("%c\n",c = getch());
        switch(c) {
            case 'N':
            case 'n':
                return(FALSE);
            case 'Y':
            case 'y':
                return(TRUE);
        }
    } while(TRUE);
}

/********************
    main(): distingue tra richiesta di installazione o disinstallazione e
        intraprende le azioni opportune. Se il primo (o unico) carattere
        del primo (o unico) parametro della command line e' un asterisco,
        main() procede alla disinstallazione. In qualunque altro caso e'
        effettuata l'installazione. Il registro DOS errorlevel contiene 0
        se ZAPTSR e' stato installato o disinstallato regolarmente; 1 se
        e' stata tentata una disinstallazione senza che ZAPTSR fosse
        attivo in RAM.
********************/

int main(int argc,char **argv)
{
    register cnt = 0, mem = FALSE;

    printf(CopyRight);
    if(argc > 1) {
        if(*argv[1] == UNINST_OPT)
            if(cnt = AreYouThere()) {
                uninstall(cnt);
                return(0);
```

```
            }
            else {
                printf(UnInstErrMsg);
                return(1);
            }
        if(*argv[1] == ALLOC_OPT)
            mem = TRUE;
    }
    if(cnt = AreYouThere())
        if(!confirm(cnt+1))
            return(0);
    install(cnt+1,mem);
    return(0);
}
```

I commenti inseriti nel listato rendono superfluo dilungarsi nella descrizione degli algoritmi; è invece opportuno evidenziare le due limitazioni di cui ZAPTSR soffre. In primo luogo esso controlla solamente i 640 Kb di memoria convenzionale: rimane esclusa la upper memory, resa disponibile tra i 640 Kb e il primo Mb da alcuni gestori di memoria estesa/espansa nonché dal DOS 5.0. Inoltre, la RAM allocata a programmi TSR installati dopo ZAPTSR, ma ad indirizzi di memoria inferiori, non viene liberata: detti programmi sono solamente disattivati mediante il ripristino dei vettori di interrupt originali.

Il programma deve essere compilato con l'opzione -k+, in quanto le parti scritte in inline assembly tengono conto del codice generato dal compilatore per la gestione standard dello stack anche nelle funzioni che non prendono parametri. Ciò vale anche con riferimento alla GData(), che deve riservare ai dati un numero esatto di byte (vedere anche pag.).

Il lettore volonteroso (e temerario) può tentare di realizzare un programma che operi come un vero e proprio controllore del sistema, intercettando i TSR di volta in volta caricati per essere in grado di individuarli e disinstallarli in ogni caso: la tabella che segue contiene alcuni suggerimenti:

INTERRUPT E SERVIZI DI CARICAMENTO E TERMINAZIONE DEI PROGRAMMI

INT	SERV.	STRATEGIA
2Fh		Necessario per consentire la comunicazione tra porzione transiente e porzione residente.
21h	4Bh	Utilizzato dal DOS per caricare da disco a memoria i programmi. Se `AL = 0` il programma viene eseguito. In tal caso `DS:DX` punta al nome; la word ad `ES:BX` è l'indirizzo di segmento dell'environment (se è `0` il programma eseguito condivide l'environment di quello chiamante). Prima di concatenare il gestore originale occorre salvare la tavola dei vettori.
20h 21h 21h	 00h 4Ch	Il programma termina senza rimanere residente: la tabella dei TSR non necessita aggiornamenti.
21h 27h	31h	Il programma termina e rimane residente in memoria. Il PSP del programma che sta per essere installato è quello attuale, a meno che il "controllore" lo abbia sostituito con il proprio attivandosi (vedere pag.). La tabella dei TSR deve essere aggiornata con i dati raccolti intercettando l'int 21h servizio 4Bh.

VETTORI DI INTERRUPT O PUNTATORI?

Degli interrupt e dei loro vettori si parla diffusamente a pag. e seguenti. Qui l'attenzione si sposta sl fatto che normalmente non tutti i 256 vettori della tavola sono utilizzati: molti non vengono inizializzati al bootstrap e, comunque, vi è un certo numero di interrupt riservati alle applicazioni (ad esempio il gruppo F0h-FDh). Da ciò deriva che è perfettamente lecito, per qualsiasi programma, installare proprie routine di interrupt che non siano necessariamente gestori di altre già esistenti ed attive. Ma vi è un'altra implicazione, che rende possibili sviluppi interessanti: la possibilità di utilizzare i vettori di interrupt come puntatori immediatamente conoscibili da tutto il sistema (anche da applicazioni diverse da quella che li inizializza).

Si supponga, ad esempio, che un programma abbia la necessità di condividere con uno o più *child process* (applicazioni da esso stesso lanciate) una quantità di variabili tale da rendere pesante il loro passaggio attraverso la `spawnl()` o `spawnv()` (vedere pag. 129): potrebbe rivelarsi conveniente allocare un'area di memoria di dimensioni sufficienti e scriverne l'indirizzo nella tavola dei vettori perché essa sia accessibile a tutte le applicazioni attive nel sistema.

```
....
void far *common_data;
....
common_data = farmalloc(10000);
setvect(0xF1,(void(interrupt *)())common_data);
....
```

Il frammento di codice riportato alloca 10000 byte al puntatore `common_data` e scrive nella tavola dei vettori l'indirizzo restituito da `farmalloc()`, come vettore F1h: qualunque child process può accedere al buffer `common_data` leggendone l'indirizzo nella tavola dei vettori. Il putatore `common_data` è definito puntatore a dati di tipo `void` per evidenziare che il buffer può contenere qualsivoglia tipo di dati: è sufficiente referenziarlo con i casts di volta in volta opportuni. Inoltre `common_data` è definito `far`, in quanto puntatore a 32 bit (l'ipotesi è di compilare per un modello di memoria "piccolo"; vedere pag. 143): nei modelli di memoria compact, large e huge esso lo è per default.

I successivi esempi di questo paragrafo presumono, per semplicità, l'uso in compilazione di un modello di memoria "grande" (compact, large, huge).

La scelta del vettore da utilizzare è problematica: un programma non ha infatti modo di scoprire con assoluta sicurezza se un vettore sia utilizzato da altre applicazioni oppure sia, al contrario, libero[431]. Per evitare di sottrarre ad un programma uno degli interrupt da esso gestiti si può adottare un accorgimento prudenziale, consistente nell'inserire in testa al buffer un'istruzione di salto all'indirizzo originale dell'interrupt.

```
....
void *common_data;
char *aux_ptr;
....
aux_ptr = (char *)malloc(10000+sizeof(char)+sizeof(void far *));
*aux_ptr = 0xEA;
(void(interrupt *)())*(long *)(aux_ptr+1) = getvect(0xF1);
common_data = (void *)(aux_ptr+sizeof(char)+sizeof(void far *));
setvect(0xF1,(void(interrupt *)())aux_ptr);
....
```

Il puntatore `aux_ptr` è definito per comodità: tutte le operazioni illustrate potrebbero essere effettuate tramite il solo `common_data`, con cast più complessi; inoltre `aux_ptr` è dichiarato `char` per sfruttare con incrementi unitari l'aritmetica dei puntatori. La `malloc()` alloca un buffer la cui ampiezza, rispetto all'esempio precedente, è incrementata di tanti byte quanti sono sufficienti a contenere gli opcodes dell'istruzione di salto[432]. Nel primo byte del buffer è memorizzato il valore `EAh`, opcode dell'istruzione `JMP FAR`; nei successivi quattro il vettore originale dell'int F1h: infatti, dal momento che `aux_ptr` è un puntatore a `char`, l'espressione `aux_ptr+1` punta al secondo byte del buffer, e rappresenta, in particolare, un puntatore a un dato a 32 bit (risultato ottenuto mediante il cast a puntatore a `long`), la cui indirezione (il dato a 32 bit stesso), forzata a puntatore ad `interrupt`, è valorizzata con il valore restituito dalla `getvect()`[433]. Il puntatore `common_data` è poi inizializzato in modo tale da "scavalcare" l'istruzione di salto. Prima di restituire il controllo al sistema, il programma ripristina il vettore originale con l'istruzione:

```
setvect(0xf1,(void(interrupt *)())*(long *)(aux_ptr+1));
```

Operazioni di cast analoghe sono descritte ed utilizzate a pagina ; va ancora sottolineato che il child process che acquisisce, ad esempio mediante `getvect()`, l'indirizzo dell'interrupt prescelto deve

[431] Forse gli unici casi in cui esiste un buon grado di sicurezza sono quelli in cui il vettore punta ad un indirizzo nel quale è davvero improbabile (se non impossibile) che si trovi codice eseguibile (ad esempio un vettore pari a `0000:0000` è sicuramente inutilizzato).

[432] In tutto 5: uno per l'opcode dell'istruzione `JMP FAR` e quattro per il vettore a 32 bit.

[433] Il cast a puntatore a `interrupt` evita che il compilatore, assegnando tale tipo di dato a una locazione che dovrebbe contenere un `long`, segnali un conflitto tra i tipi di dato.

incrementarlo di un numero di byte pari a `sizeof(char)+sizeof(void far *)` per ottenere il reale indirizzo dei dati, corrispondente a `common_data`:

```
....
void *common_data;
....
common_data = (void *)(((char *)getvect(0xf1))+sizeof(char)+sizeof(void *));
....
```

Il cast di `getvect()` a puntatore a character ha lo scopo di forzare incrementi unitari del puntatore sommandovi la dimensione dell'istruzione `JMP FAR` completa di indirizzo.

Ogni buffer allocato da `malloc()` è automaticamente rilasciato quando il programma che ha invocato la `malloc()` termina. Se i dati in esso contenuti devono essere condivisi da applicazioni attive dopo il termine dell'esecuzione del programma, occorre che la memoria necessaria sia loro riservata con altri metodi.

E' valida, allo scopo, la tecnica delle funzioni jolly, utilizzabile dai TSR per lasciare residenti in memoria dati e routine[434], discussa a pag. e alla quale si rimanda, precisando però che allocando nel code segment lo spazio per i dati si determina un incremento delle dimensioni del file eseguibile pari al numero di byte riservati.

In alternativa, è possibile creare il buffer con la `allocmem()`, che utilizza il servizio 48h dell'int 21h (vedere pag.): in questo caso si rendono necessarie due precauzioni.

La prima consiste nel forzare il DOS ad allocare la memoria in modo tale da evitare un'eccessiva frammentazione della RAM libera: allo scopo si può invocare la `allocmem()` dopo avere impostato la strategia di allocazione LastFit (vedere pagina); il buffer occupa la porzione "alta" della memoria convenzionale.

```
....
unsigned blockseg;            /* conterra' l'indirizzo di segmento del buffer */
int strategy;                /* usata per salvare la strategia di allocazione */
....
_AX = 0x5800;
asm int 21h;                    /* individua strategia attuale di allocazione */
strategy = _AX;
_AX = 0x5801;
_BX = 2;
asm int 21h;                                     /* imposta strategia LastFit */
allocmem(1000,&blockseg);           /* alloca 1000 paragr. (circa 16000 bytes) */
_AX = 0x5801;
_BX = strategy;
asm int 21h;                          /* ripristina la strategia di allocazione */
setvect(0xF1,(void(interrupt *)())MK_FP(blockseg,0));
....
```

La macro `MK_FP()` (pag. 24) è utilizzata per costruire l'indirizzo `far` completo del buffer (`blockseg` ne costituisce la parte segmento; l'offset è zero).

La seconda precauzione, ancora più importante[435], sta nell'impedire al DOS di rilasciare, all'uscita dal programma, il buffer allocato con `allocmem()`: infatti tutti i blocchi assegnati ad un programma non TSR vengono liberati (dal DOS) quando esso termina; in altre parole il DOS rilascia tutte

[434] Se la funzione contenente i dati è definita, nel sorgente, prima di ogni altra, il programma può terminare come un TSR e lasciare residente in memoria quella soltanto (oltre, naturalmente, allo startup code). Dal momento che la funzione è in realtà un'area di "parcheggio" per dati, il programma non è un vero TSR, in quanto nessuna sua parte rimane attiva in memoria. Si tratta, ancora una volta, di un trucco...

[435] Direi, anzi, assolutamente fondamentale.

le aree di RAM il cui Memory Control Block reca nel campo PSP (vedere pag.) l'indirizzo di segmento del Program Segment Prefix di quel programma. Per evitare tale spiacevole inconveniente è sufficiente modificare il contenuto del campo PSP del MCB dell'area di RAM allocata al buffer:

```
....
unsigned blockseg;            /* conterra' l'indirizzo di segmento del buffer */
....
allocmem(1000,&blockseg);          /* alloca 1000 paragr. (circa 16000 bytes) */
*(unsigned far *)MK_FP(blockseg-1,1) = 0xFFFF;
....
```

Nell'esempio viene assegnato il valore FFFFh al campo PSP del MCB: si tratta di un valore del tutto arbitrario, che può, tra l'altro, essere utilizzato dalle applicazioni interessate, per individuare l'origine dell'area di RAM.

Le considerazioni sin qui espresse relativamente all'utilizzo dei vettori di interrupt come puntatori a dati mantengono la loro validità anche qualora si intenda servirsi dei medesimi come puntatori a funzioni. In effetti, ogni vettore è, per definizione, un puntatore a funzione, in quanto esprime l'indirizzo di una routine eseguibile, ma, dal momento che gli interrupt non sono vere e proprie funzioni C[436], può essere interessante approfondire appena l'argomento.

Un programma ha la possibilità di mettere a disposizione dell'intero sistema parte del proprio codice. Si tratta, in pratica, di un programma TSR, il quale non installa gestori di interrupt, ma normali funzioni C, che potranno essere eseguite da tutte le applicazioni di volta in volta attive nel sistema, a patto che ne conoscano il prototipo[437]. La chiamata avviene mediante indirezione del puntatore alla funzione, che in questo caso è rappresentato da un vettore di interrupt. Vedere pag. 93.

Per quanto riguarda l'installazione in RAM delle routine si rimanda a quanto discusso circa i programmi TSR (pag.). Si ricordi inoltre che i vettori degli interrupt 00h, 04h, 05h e 06h sono ripristinati dalla keep(), mentre il DOS, da parte sua, ripristina i vettori degli interrupt 23h e 24h prelevandone i valori originali dal PSP del programma.

Circa le funzioni C installabili va invece sottolineato, innanzitutto, che esse devono necessariamente essere dichiarate far, poiché i vettori di interrupt sono indirizzi a 32 bit. Inoltre esse non possono essere invocate con l'istruzione INT[438], ma solo con l'indirezione del vettore (o meglio, del puntatore con esso valorizzato): ciò evita il ricorso a funzioni e strutture di interfaccia o allo inline assembly, e consente di utilizzare le convenzioni di alto livello di chiamata delle funzioni C (passaggio di copie dei parametri attraverso lo stack, restituzione di un valore, etc.). Attenzione, però: il puntatore utilizzato per referenziare la funzione non deve essere un puntatore a funzione interrupt, dal momento che, come si è detto, le rotuine residenti non sono esse stesse funzioni interrupt, bensì normali funzioni far; se si ottiene il vettore (indirizzo della funzione) mediante getvect(), il valore da questa restituito deve subire un cast:

```
....
int (far *resFuncPtr)(char *str);
....
resFuncPtr = (int(far *)(char *str))getvect(0x1F);
....
```

---

[436] Anche i gestori scritti in linguaggio C devono comunque mantenere una rigorosa coerenza con le particolari regole di interfacciamento con il sistema seguite dagli interrupt (vedere, al proposito, pag.).

[437] Se non lo conoscessero, come potrebbero passare i parametri eventualmente richiesti ed interpretare correttamente il valore restituito?

[438] Si può usare l'istruzione INT solo se si tratta di gestori di interrupt.

Per completezza, si deve poi osservare che inserire una `JMP FAR` in testa alla funzione a scopo di sicurezza è più problematico di quanto non lo sia l'inserimento in testa ad un buffer[439]. Infine, l'utilizzo di funzioni di libreria all'interno delle funzioni installabili deve uniformarsi a quanto esposto a pag. con riferimento ai programmi TSR.

IL CMOS

Le macchine dotate di processore Intel 80286 o superiore dispongono di 64 byte (o 128, a seconda dei modelli) di memoria CMOS, permanentemente alimentata da una batteria, nella quale sono memorizzate, oltre alla data e ora, lo *shutdown byte*[440] e le informazioni relative alla configurazione hardware[441].

L'accesso alla memoria CMOS è possibile attraverso operazioni di lettura e scrittura sulle porte hardware 70h e 71h.

```
/********************

    BARNINGA_Z! - 1991

    READCMOS.C - readcmos()

    unsigned char cdecl readcmos(int off);
    int off;     l'offset, nel CMOS, del byte da leggere.
    Restituisce: il byte letto nel CMOS all'offset specificato.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -k- -mx readcmos.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-

unsigned char cdecl readcmos(int off)
```

---

[439] Il compilatore genera in testa alla funzione, in modo automatico e trasparente al programmatore, le istruzioni assembler per la gestione dello stack (le solite `PUSH BP` e `MOV BP,SP` seguite, se nella funzione sono definite variabili locali, dall'istruzione per il decremento di `SP`). Un'istruzione `JMP FAR` seguita dai 4 byte di indirizzo si collocherebbe inevitabilmente dopo dette istruzioni: sarebbe il disastro. Vi è una sola scappatoia semplice, peraltro onerosa e restrittiva dal punto di vista logico: dichiarare tutte le funzioni installabili prive di parametri e di variabili locali, e compilare senza standard stack frame. Chi volesse invece eccedere negli stratagemmi potrebbe riservare i 5 byte in testa alla funzione e poi inizializzare, con una procedura runtime, i primi byte della funzione con gli opcode necessari alla `FAR JMP` e alla gestione dello stack nell'ordine necessario, scrivendoli all'indirizzo puntato dal vettore, come se si avesse a che fare con un buffer piuttosto che con una funzione. Come nel caso del buffer, deve poi incrementare il puntatore alla funzione (non il vettore, ma il puntatore con esso inizializzato) di 5 perché esso punti effettivamente all'inizio del codice eseguibile. Occorre però una buona conoscenza delle modalità di gestione dello stack e può tornare utile qualche occhiatina al sorgente assembler prodotto dal compilatore (opzione `-S`). Forse, dopo tutto, è preferibile correre qualche rischio: se si lavora con attenzione, la probabilità che il vettore prescelto sia gia utilizzato è, in concreto, molto piccola.

[440] Lo shutdown byte è utilizzato per il rientro in *real mode* da *protected mode* mediante reset del processore.

[441] Delle macchine che dispongono di 128 byte di CMOS, molte utilizzano i 64 byte aggiuntivi per la memorizzazione di parametri non standard di configurazione.

```
{
    asm {
        mov al,byte ptr off;
        out 70h,al;                                         /* imposta l'offset */
        jmp $+2;                                        /* genera breve ritardo */
        in al,71h;                                             /* legge il byte */
    }
    return(_AL);
}
```

Il listato della `readcmos()` è estremamente semplice; l'unica particolarità è rappresentata dall'istruzione `JMP $+2`, ininfluente dal punto di vista del flusso di esecuzione[442]: essa ha solamente lo scopo di introdurre un piccolo ritardo tra le due operazioni sulle porte, in modo tale che prima della seconda trascorrano i cicli di clock necessari al completamento della prima.

La scrittura di un byte nel CMOS avviene in maniera analoga:

```
/********************

    BARNINGA_Z! - 1991

    WRITCMOS.C - writcmos()

    void cdecl readcmos(int off, unsigned char val);
    int off;              l'offset, nel CMOS, del byte da scrivere.
    unsigned char val;    il byte da scrivere nel CMOS.
    Restituisce: nulla.

    COMPILABILE CON TURBO C++ 2.0

        tcc -O -d -c -k- -mx writcmos.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/
#pragma  inline
#pragma  option -k-

void cdecl writcmos(int off,unsigned char val)
{
    asm {
        mov al,byte ptr off;
        out 70h,al;
        mov al,val;
        out 71h,al;
    }
}
```

Nella `writcmos()` la presenza dell'istruzione `MOV AL,VAL` tra le due operazioni di scrittura sulle porte elimina la necessità di inserire un'istruzione `JMP $+2`.

Di seguito presentiamo un programma che utilizza le due funzioni testè commentate per effettuare un salvataggio su file del contenuto del CMOS, nonché il suo eventuale ripristino. Chi abbia intenzione di sperimentare l'effetto di modifiche alla configurazione hardware del proprio personal computer pasticciando con le routine di setup del BIOS intuisce al volo quanto possa essere prezioso un backup dei dati originali. Si ricordi inoltre che la batteria di alimentazione del CMOS, come tutte le

---

[442] Il simbolo del dollaro (`$`) rappresenta in assembler l'indirizzo attuale. L'istruzione `JMP $+2` significa quindi "salta in avanti a due byte da qui" ma, poiché essa produce in compilazione due opcodes il salto, in pratica, non ha alcun effetto: viene eseguita l'istruzione immediatamente successiva.

batterie, ha il pessimo vizio di scaricarsi, più o meno lentamente: può accadere a chiunque, un brutto giorno, di ritrovarsi nella necessità di ricostrure a memoria tutta la configurazione dopo avere sostituito la batteria ormai esausta.

```
/******************************************************************************

    Barninga_Z! - 09/08/93

    CMOSBKP.C - Se richiesta opzione -S (SAVE), crea una copia del CMOS nel file
        CMOSBKP.SAV nella stessa directory dalla quale e' stato lanciato (sono
        copiati i byte da offset 0x10 a 0x7F; in pratica NON vengono copiati i
        bytes usati dall'orologio e dalla diagnostica). Se viene richiesta
        opzione -r oppure -R (RESTORE), copia nel CMOS (a partire da offset 0x10)
        il contenuto del file CMOSBKP.SAV precedentemente creato (-r NON
        ripristina i bytes ad offset CMOS da 40h a 4Fh). Se richiesta opzione -o
        oppure -O confronta il contenuto del CMOS con il contenuto del file
        CMOSBKP.SAV (o non confronta i bytes ad offset CMOS da 40h a 4Fh). Se
        richiesta opzione -c effettua un test sul byte del CMOS usato dalla
        diagnostica del bootstrap e visualizza i risultati.
        Il nome di file puo' essere specificato sulla command line se si vuole
        che esso non sia CMOSBKP.SAV.

    Compilato con BORLAND C++ 3.1:

        bcc -O -d -rd -mt -lt cmosbkp.c

******************************************************************************/
#pragma  inline                                       // per readcmos() e writcmos()

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define  CHECK          0x0E                    /* byte diagnostico nel CMOS */
#define  START          0x10               /* primo byte da gestire nel CMOS */
#define  OFF1           0x40    /* inizio intervallo da saltare se richiesto */
#define  OFF2           0x50    /* byte successivo a fine intervallo saltato */
#define  END            0x7F              /* ultimo byte da gestire nel CMOS */
#define  BYTES          (END-START+1) /* numero di bytes da gestire nel CMOS */
#define  NAME           "CMOSBKP"                     /* nome del programma */
#define  EXT            "SAV"                    /* estensione del file dati */
#define  REL            "1.5"                                   /* versione */
#define  OK             "O.K."         /* messaggio per diagnostica tutto OK */
#define  SW             '-'                         /* switch per le opzioni */

int pascal __IOerror(int dosErr);

int main(int argc,char **argv);
void checkcmos(void);
int compcmos(char *datafile,unsigned cod);
int handlecmos(char *opt,char *datafile);
unsigned char cdecl readcmos(int off);
int readfile(char *datafile,unsigned char *cmos);
int restcmos(char *datafile,unsigned cod);
int savecmos(char *datafile);
void writcmos(int off,unsigned char val);

struct CMOSCHK {
    char     *item;                               /* oggetto della diagnosi */
    unsigned  mask;                       /* mask per il bit corrispondente */
    char     *bad;                                /* messaggio per problema */
} chk[] = {
```

```
    {"Date/Time",  4,"Invalid"},
    {"Hard Disk",  8,"Can't boot"},
    {"RAM Size ", 16,"Different from startup check"},
    {"Equipment", 32,"Different from startup check"},
    {"Checksum ", 64,"Different from equipment record"},
    {"Battery  ",128,"Power lost"},
    {NULL,0,NULL}
};

char errmsg[] = "\
%s: syntax: %s option [filename]\n\n\
    option is one of the following:\n\
        -c    check CMOS status\n\
        -O    compare CMOS with file\n\
        -o    compare CMOS with file but offsets 40h to 4Fh\n\
        -R    restore CMOS from file\n\
        -r    restore CMOS from file but offsets 40h to 4Fh\n\
        -S    save CMOS to file but offsets 00h to 0Fh\n\
";

// main() gestisce una prima analisi della command line per decidere quale
// funzione invocare

int main(int argc,char **argv)
{
    int pflag = 2;      /* 0 o 2 - elemento di argv[] contenente il nomefile */

    printf("%s: CMOS save/restore utility -%s- Barninga_Z!\n",NAME,REL);
    switch(argc) {
        case 2:                           /* passata solo l'opzione su cmdlin */
            strcpy(argv[0]+strlen(argv[0])-3,EXT);
            pflag = 0;
        case 1:                  /* nessuna opzione: gestito da handlecmos() */
        case 3:                   /* passati opz e nome di file sulla cmdlin */
            if(!handlecmos(argv[1],argv[pflag]))
                break;
        default:
            perror(NAME);
            printf(errmsg,NAME,NAME);
    }
    return(errno);
}

// checkcmos() legge dal CMOS il byte di checksum, poi effettua la somma dei bytes
// di cui quello e' a sua volta la somma e confronta i valori trovati.

void checkcmos(void)
{
    register i;
    unsigned char cbyte;

    cbyte = readcmos(CHECK);
    printf("%s: CMOS check results as follows:\n\n",NAME);
    for(i = 0; chk[i].item; i++)
        printf("    %s: %s\n",chk[i].item,(cbyte && chk[i].mask) ? chk[i].bad : OK);
}

// compcmos() confronta il contenuto del CMOS con il contenuto di un file per vedere
// se questo e' una copia della configurazione attuale

int compcmos(char *datafile,unsigned cod)
{
    register i, diff;
    unsigned char cmos[BYTES];
```

```
    if(readfile(datafile,cmos))
        return(1);
    for(diff = 0, i = 0; i < OFF1; i++)
        if(cmos[i] != readcmos(i+START))
            ++diff;
    if(!cod)
        for(; i < OFF2; i++)
            if(cmos[i] != readcmos(i+START))
                ++diff;
    for(i = OFF2; i < BYTES; i++)
        if(cmos[i] != readcmos(i+START))
            ++diff;
    printf("%s: %u differences between CMOS and %s data.\n",NAME,diff,datafile);
    return(0);
}

// handlecmos() analizza le opzioni della command line e invoca la funzione
// corrispondente

int handlecmos(char *opt,char *datafile)
{
    register cod = 0;

    if(*opt != SW)
        cod = __IOerror(EINVDAT);
    else
        switch(*(opt+1)) {
            case 'c':
                checkcmos();
                break;
            case 'o':
                cod = 1;
            case 'O':
                cod = compcmos(datafile,cod);
                break;
            case 'r':
                cod = 1;
            case 'R':
                cod = restcmos(datafile,cod);
                break;
            case 's':
                cod = 1;
            case 'S':
                cod = savecmos(datafile);
                break;
            default:
                cod = __IOerror(EINVDAT);                  /* opzione errata */
        }
    return(cod);
}

// readcmos() legge un byte dal CMOS

unsigned char readcmos(int off)
{
    asm {
        mov al,byte ptr off;
        out 70h,al;                                        /* imposta l'offset */
        jmp $+2;                                       /* genera breve ritardo */
        in al,71h;                                             /* legge il byte */
    }
    return(_AL);
}
```

```
// readfile() legge in un buffer il contenuto di un file contenente una
// configurazione di CMOS

int readfile(char *datafile,unsigned char *cmos)
{
    FILE *indata;

    if(!(indata = fopen(datafile,"rb")))
        return(1);
    if(fread(cmos,sizeof(char),BYTES,indata) < BYTES)
        return(1);
    return(0);
}


// restcmos() scrive nel CMOS il contenuto del buffer riempito con i bytes letti
// da file da readfile()

int restcmos(char *datafile,unsigned cod)
{
    register i, cnt;
    unsigned char cmos[BYTES];

    if(readfile(datafile,cmos))
        return(1);
    for(cnt = 0, i = 0; i < OFF1; cnt++, i++)
        writcmos(i+START,cmos[i]);
    if(!cod)
        for(; i < OFF2; cnt++, i++)
            writcmos(i+START,cmos[i]);
    for(i = OFF2; i < BYTES; cnt++, i++)
        writcmos(i+START,cmos[i]);
    printf("%s: %u CMOS bytes restored from %s.\n",NAME,cnt,datafile);
    return(0);
}

// savecmos() scrive il contenuto del CMOS in un buffer e poi scrive questo in un
// file, generando cosi' una copia di backup del CMOS

int savecmos(char *datafile)
{
    FILE *outdata;
    register i;
    unsigned char cmos[BYTES];

    if(!(outdata = fopen(datafile,"wb")))
        return(1);
    for(i = 0; i < BYTES; i++)
        cmos[i] = readcmos(i+START);
    if(fwrite(cmos,sizeof(char),BYTES,outdata) < BYTES)
        return(1);
    printf("%s: %u CMOS bytes saved to %s.\n",NAME,i,datafile);
    return(0);
}

// writcmos() scrive un byte nel CMOS

void writcmos(int off,unsigned char val)
{
    asm {
        mov al,byte ptr off;
        out 70h,al;
        mov al,val;
```

```
        out 71h,al;
    }
}
```

Come si vede, CMOSBKP si compone di poche funzioni, ciascuna dedicata ad una operazione elementare di gestione dei dati del CMOS: non sembra necessario, pertanto, dilungarsi in una loro approfondita descrizione; tuttavia va osservato che il programma potrebbe essere perfezionato eliminando la `handlecmos()` ed utilizzando, in luogo, gli strumenti di gestione delle opzioni della riga di comando descritti a pag. 481 e seguenti. Ciò consentirebbe, inoltre, di razionalizzare la struttura di `main()`.

Concludiamo queste scarne note in tema di CMOS con un... consiglio da amico: è buona norma predisporre un dischetto autopartente[443] e copiare sul medesimo, oltre alle indispensabili componenti del DOS, CMOSBKP.EXE e un file di backup del CMOS da questo generato. Può sempre servire.

C... COME CESARE

Il Cesare in questione è proprio Caio Giulio Cesare, il noto imperatore romano al quale si deve la riforma del calendario, effettuata nell'anno 46 a.C., volta, tra l'altro, al perfezionamento della tecnica di calcolo degli anni bisestili. Dal nome dell'imperatore deriva l'appellativo "numero giuliano", indicante l'integral (vedere pag. 12) ottenibile applicando una formula nota a giorno, mese ed anno di una qualsiasi data: esso esprime il numero di giorni trascorsi da una origine nota (che si colloca più o meno intorno al 5000 a.C.) alla data medesima.

La funzione `date2jul()` consente di calcolare il numero giuliano corrispondente alla data voluta; va osservato che l'anno può essere qualsiasi intero maggiore o uguale a 0: `date2jul()` si occupa di ricondurlo ad un valore "accettabile" sommandovi una costante esprimente il secolo, come descritto nel commento in testa al listato.

```
/******************************************************************

    Barninga_Z! - 1994

    long cdecl date2jul(int day,int month,int year);
    int day;     giorno della data da convertire (1-31).
    int month;   mese della data da convertire (1-12).
    int year;    anno della data da convertire.   Può  essere
                 espresso   con  qualsiasi  numero  di cifre.

    RESTITUISCE  Il numero giuliano corrispondente alla data
                 specificata. Attenzione: se il valore fornito
                 come anno è minore di 100 viene sommato 1900 al
                 valore, perciò 2 = 1902 e 94 = 1994. Per
                 esprimere 2002 bisogna quindi fornire il valore
                 reale. Se il valore fornito è compreso tra 100 e
                 999 la  funzione somma 2000 al valore, perciò
                 995 = 2955. Quindi 1995 deve essere 95 o 1995.

    Compilato con Borland C++ 3.1

    bcc -O -d -c -mx date2jul.c

    dove -mx può essere -mt -ms -mc -mm -ml -mh

******************************************************************/
```

---

[443] Un floppy disk formattato con opzione /s, contenente almeno i file nascosti del DOS e l'interprete dei comandi, in grado di effettuare il bootstrap della macchina senza accedere al disco rigido.

```
long cdecl date2jul(int day,int month,int year)
{
    long ctmp, dtmp, mtmp, ytmp;

    if(year < 100)
        year += 1900;                               // se anno = 82 si assume 1982
    if(year < 1000)
        year += 2000;                              // se anno = 182 si assume 2182
    if(month > 2) {
        mtmp = (long)(month-3);
        ytmp = (long)year;
    }
    else {
        mtmp = (long)(month+9);
        ytmp = (long)(year-1);
    }
    ctmp = (ytmp/100);
    dtmp = ytmp-(100*ctmp);
    return((146097L*ctmp)/4L+(1461L*dtmp)/4L+(153L*mtmp+2)/5L+1721119L+(long)day);
}
```

La formula applicata, di natura strettamente tecnica, non necessita commenti; ne deriva inoltre una funzione C del tutto banale. Il numero giuliano corrispondente alla data è restituito come `long` e può essere validamente utilizzato, ad esempio, nel calcolo dei giorni intercorsi tra due date: essi sono infatti pari alla differenza tra i due numeri giuliani.

La funzione `jul2date()` effettua l'operazione inversa a quella di `date2jul()`: essa infatti calcola la data corrispondente a un numero giuliano.

```
/******************************************************************

    Barninga_Z! - 1994

    int cdecl jul2date(long jul,int *day,int *month,int *year);
    long jul;    il numero giuliano da convertire.
    int *day;    puntatore   al   giorno   della   data   da
                 convertire (conterrà 1-31).
    int *month;  puntatore al mese della data  da  convertire
                 (conterrà 1-12).
    int *year;   puntatore all'anno della data da convertire.
                 Vedere anche il valore restituito.

    RESTITUISCE  Il secolo  corrispondente  alla  data  espressa
                 dal numero giuliano (19 per 1900, e così via).
                 Attenzione:  l'anno  è  sempre  calcolato
                 sottraendovi il secolo, pertanto  occorre
                 controllare il valore restituito dalla funzione
                 per determinare la data  in  modo  completo (se
                 *year vale 82 e la  funzione  restituisce  19,
                 allora l'anno è 1982; se la funzione restituisce
                 20,  allora l'anno è 2082).

    Compilato con Borland C++ 3.1

    bcc -O -d -c -mx jul2date.c

    dove -mx può essere -mt -ms -mc -mm -ml -mh

******************************************************************/

int cdecl jul2date(long jul,int *day,int *month,int *year)
{
    register cent;
```

```
    long dayTmp, monthTmp, yearTmp, tmp;

    tmp = jul-1721119L;
    yearTmp = (4*tmp-1)/146097L;
    tmp = 4*tmp-1-146097L*yearTmp;
    dayTmp = tmp/4;
    tmp = (4*dayTmp+3)/1461;
    dayTmp = 4*dayTmp+3-1461*tmp;
    dayTmp = (dayTmp+4)/4;
    monthTmp = (5*dayTmp-3)/153;
    dayTmp = 5*dayTmp-3-153*monthTmp;
    dayTmp = (dayTmp+5)/5;
    yearTmp = 100*yearTmp+tmp;
    *day = (int)dayTmp;
    *year = (int)yearTmp;
    if((int)monthTmp < 10)
        *month = (int)monthTmp+3;
    else {
        *month = (int)monthTmp-9;
        (*year)++;
    }
    cent = (*year)/100;
    (*year) -= (cent*100);
    return(cent);
}
```

Anche la `jul2date()` utilizza una formula tecnica, che non richiede alcun commento. Più interessante è sottolineare che la funzione richiede quali parametri, oltre al numero giuliano da convertire, anche i puntatori alle variabili in cui memorizzare giorno, mese e anno della data calcolata: ciò si rende necessario in quanto le funzioni C possono restituire un solo valore (vedere pag. 87). Si noti, inoltre, che l'anno memorizzato all'indirizzo `year` è sempre un numero compreso tra `0` e `99` (estremi inclusi): infatti il secolo è restituito dalla funzione (incrementato di uno), cosicché l'anno espresso in 4 cifre può essere calcolato, dopo la chiamata a `jul2date()`, sommando il valore memorizzato all'indirizzo `year` al valore restituito moltiplicato per `1000`. Il secolo corrispondente alla data è ottenibile sottraendo `1` al valore restituito.

Presentiamo ancora una funzione che, pur non operando sui numeri giuliani, risulta utile nell'effettuazione di calcoli sulle date: la `isleapyear()` consente di determinare se un anno è bisestile.

```
/*****************************************************************

    Barninga_Z! - 1994

    int cdecl isleapyear(int year);
    int year;  anno sul quale effettuare il controllo. Può essere
               espresso  con  qualsiasi  numero  di cifre.
    RESTITUISCE   0    l'anno non è bisestile.
                  1    l'anno è bisestile.
                  Attenzione: se il valore fornito come anno è
                  < 100  viene sommato 1900 al valore stesso,
                  perciò 2 = 1902 e 94 = 1994. Per esprimere 2002
                  bisogna quindi fornire il valore reale. Se il
                  valore fornito è compreso tra 100 e 999 la
                  funzione somma 2000 al valore, così 995 = 2955.
                  Quindi il 1995 va indicato come 95 o 1995.

    Compilato con Borland C++ 3.1

    bcc -O -d -c -mx isleapyr.c

    dove -mx può essere -mt -ms -mc -mm -ml -mh
```

```
*****************************************************************/

#include "dates.h"

int cdecl isleapyear(int year)
{
    register lyr;

    lyr = 0;
    if(year < 100)
        year += 1900;                          // se anno = 82 si presume 1982
    if(year < 1000)
        year += 2000;                         // se anno = 182 si presume 2182
    if(year == (4*(year/4)))
        lyr = 1;
    if(year == (100*(year/100)) )
        lyr = 0;
    if(year == (400*(year/400)) )
        lyr = 1;
    return(lyr);
}
```

La `isleapyear()` adotta le medesime convenzioni di `date2jul()` circa il formato del parametro `year`.

Di seguito presentiamo un semplice programma adatto a collaudare le funzioni sin qui descritte (i listati di queste non compaiono nel programma):

```
/******************************************************************

    JULTEST.C - Barninga Z! - 1994

    Programma di prova per funzioni calcolo date.

    Compilato con Borland C++ 3.1

    bcc jultest.c

******************************************************************/
#include <stdio.h>
#include <stdlib.h>

long cdecl date2jul(iny day,int month,int year);
int cdecl isleapyear(int year);
int cdecl jul2date(long jul,int *day,int *month,iny *year);

int main(int argc,char **argv)
{
    int century, year, month, day;
    static char *answers[] = {
        "NO",
        "YES"
    };

    switch(argc) {
        case 2:
            century = jul2date(atol(argv[1]),&day,&month,&year);
            printf("%d/%d/%d%d (leapyear: %s)\n",day,month,century,year,
                                             answers[isleapyear(year)]);
            break;
        case 4:
            printf("julian number: %ld (leapyear: %s)\n",date2jul(atoi(argv[1]),
                             atoi(argv[2]),atoi(argv[3])),isleapyear(atoi(argv[3])));
            break;
```

```
        default:
            printf("Usage: jultest julnum | day month year\n");
            return(1);
    }
    return(0);
}
```

Il programma può essere invocato con uno oppure quattro parametri numerici sulla riga di comando: nel primo caso esso assume che il parametro rappresenti un numero giuliano da convertire in data; nel secondo caso i tre argomenti sono interpretati come giorno, mese e, rispettivamente, anno esprimenti una data da convertire in numero giuliano.

LAVORARE CON I FILE BATCH

L'interprete dei comandi (COMMAND.COM nella configurazione DOS standard) fornisce una interfaccia per l'esecuzione dei programmi (a volte definiti *comandi esterni*[444]) e rende disponibili alcuni *comandi interni*, così detti in quanto implementati mediante routine interne all'interprete stesso (COPY, DEL, DIR, etc.), nonché la capacità, utilizzabile dalle applicazioni opportunamente progettate, di effettuare redirezioni (vedere pag. 116).

L'esecuzione di sequenze di comandi interni ed esterni può essere automatizzata mediante i file batch, che l'interprete è in grado di leggere ed eseguire riga per riga: ognuna contiene, in formato ASCII, un singolo comando; sono inoltre disponibili istruzioni per il controllo del flusso elaborativo, quali FOR, IF (utilizzabili direttamente da prompt) e GOTO. Per i dettagli circa la sintassi dei comandi interni si rimanda alla manualistica DOS; in questa sede si vuole sottolineare che, per mezzo di questi soltanto, è spesso difficile (se non impossibile) implementare algoritmi di una certa complessità. Ad esempio, la IF consente di controllare il contenuto del registro ERRORLEVEL (vedere pag. 108) o di verificare l'esistenza di un file[445]:

```
IF EXIST PIPPO GOTO TROVATO
ECHO PIPPO NON C'E'
GOTO END
:TROVATO
....
:END
```

Non è tuttavia possibile effettuare test su alcuna caratteristica del file stesso, quali data, ora, dimensione, né accertare se si tratti piuttosto di una directory.

L'elenco dei principali limiti all'efficacia dei file batch può continuare a lungo: non vi è modo di inserire in comandi batch stringhe contenenti data e ora di elaborazione, o parti di esse; non è possibile ritardare l'esecuzione di un comando ad un'ora specificata; il comando COPY fallisce se il file origine ha dimensione 0 byte; non vi sono strumenti in grado di estrarre da un flusso ASCII parti di testo in modo "mirato", ad eccezione del programma FIND, che ne visualizza le righe contenenti (o no) una data stringa; l'output di un comando non può essere utilizzato come parte di un successivo comando; una modifica alla lista degli argomenti del comando FOR richiede che sia modificato il file batch contenente il comando stesso[446].

---

[444] La definizione *comandi esterni* è spesso applicata ai soli programmi facenti parte del sistema operativo in senso stretto (DISKCOPY, FORMAT, etc.).

[445] O la non esistenza (IF NOT EXIST...).

[446] FOR consente di ripetere un comando su una lista di argomenti elencati tra parentesi tonde: non vi è modo di utilizzare una lista esterna al batch, quale, ad esempio, un file ASCII gestito automaticamente da altre procedure.

Dette limitazioni sono superate con una minima interattività da parte dell'utilizzatore, ma possono originare problemi quasi insormontabili laddove vi sia la necessità di una completa automazione (ad esempio in elaborazioni notturne): il presente paragrafo presenta alcuni programmi volti a superare le carenze cui si è fatto cenno[447]. Lo scopo è fornire un insieme di spunti e idee perfettibili e, al tempo stesso, adattabili a piacere secondo le specifiche esigenze di ciascuno.

L'idea più semplice: EMPTYLVL

La semplicità del listato che segue è assolutamente disarmante: il programma legge lo standard input ed immediatamente termina, valorizzando il registro `ERRORLEVEL` a `1` se lo stream `stdin` è vuoto o contiene solo spazi, tabulazioni o caratteri di ritorno a capo; in caso contrario `ERRORLEVEL` è azzerato.

```
/******************************************************************************

    EMPTYLVL.C - 20/01/93 - Barninga_Z!

    Legge una stringa da STDIN e setta il registro errorlevel a 1 se la
    stringa contiene solo spazi e/o tabs e/o CR e/o LF. In caso contrario
    il registro e' posto a 0.

    Compilato sotto Borland C++ 3.01:

    bcc -O -rd emptylvl.c

******************************************************************************/

#include <stdio.h>
#include <string.h>

#define  MAXHEAP     4096
#define  MAXBUF      2048
#define  NULLCHARS   " \t\n\r"

extern unsigned _heaplen = MAXHEAP;

int main(void)
{
    char buffer[MAXBUF];

    *buffer = NULL;
    gets(buffer);
    if(!strtok(buffer,NULLCHARS))
        return(1);                          // stringa nulla: errorlevel = 1
    return(0);                              // stringa non nulla: errorlevel = 0
}
```

Come si vede, l'unica particolarità tecnica di qualche rilevanza è costituita dall'assegnazione di un valore predefinito alla variabile `_heaplen`, per fissare la massima dimensione dello heap (vedere pag. 111) e limitare così al minimo indispensabile la quantità di memoria necessaria per il caricamento e l'esecuzione del programma.

---

[447] Si tratta di programmi progettati espressamente per automatizzare elaborazioni eseguite in parallelo da diverse macchine su dati condivisi in rete; tuttavia essi possono, comunque, risultare utili in ambienti meno complessi: si pensi a procedure eseguite su una medesima macchina, ma in multitasking (ad esempio quali task DOS in Microsoft Windows 3.x). Del resto non è affatto esclusa la possibilità di servirsene produttivamente in semplici procedure eseguite in modo standalone e monotasking.

EMPTYLVL si rivela utile in molte situazioni: ad esempio ci consente di scoprire se un file è vuoto, o se si tratta di una directory. Vediamo un esempio:

```
IF EXIST PIPPO GOTO TROVATO
ECHO PIPPO NON C'E'
GOTO END
:TROVATO
DIR | FIND "PIPPO" | FIND "<DIR>" | EMPTYLVL
IF ERRORLEVEL 1 GOTO PIPPOFIL
ECHO PIPPO E' UNA DIRECTORY
GOTO END
:PIPPOFIL
DIR | FIND "PIPPO" | FIND " 0 " | EMPTYLVL
IF ERRORLEVEL 1 GOTO PIPPODAT
ECHO IL FILE PIPPO E' VUOTO
GOTO END
:PIPPODAT
TYPE PIPPO | MORE
:END
```

Le prime righe del batch non rappresentano una novità rispetto quanto sopra accennato. Al contrario, la riga

```
DIR | FIND "PIPPO" | FIND "<DIR>" | EMPTYLVL
```

merita qualche commento. Il comando DIR produce l'elenco di tutti i file e subdirectory presenti nella directory di default: il simbolo '|' ne effettua il *piping*[448] al primo comando FIND (esterno), che lo scandisce e scrive, a sua volta, sullo standard output solo le righe contenenti la stringa "PIPPO". Ma l'output del primo FIND è trasformato (ancora mediante piping) in standard input per il secondo comando FIND, che scrive sul proprio standard output solo le righe contenenti la stringa "<DIR>": ne segue che se il file PIPPO è, in realtà, una directory, lo standard input di EMPTYLVL contiene la riga ad essa relativa dell'output originario di DIR; se invece PIPPO è un file, lo standard input di EMPTYLVL risulta vuoto. Nel primo caso, pertanto, ERRORLEVEL è posto a 0 e la IF non effettua il salto: viene visualizzata la stringa "PIPPO E' UNA DIRECTORY". Nel secondo caso ERRORLEVEL vale 1 e l'esecuzione salta all'etichetta :PIPPOFIL, a partire dalla quale, con un algoritmo del tutto analogo a quello testè commentato, si verifica se la dimensione di PIPPO riportata da DIR è 0. Se il file non è vuoto, il suo contenuto viene visualizzato, una schermata alla volta, grazie all'azione combinata del comandi TYPE (interno) e MORE (esterno).

Data e ora nei comandi: DATECMD

Il programma DATECMD è concepito per consentire l'inserimento automatico di data e ora, o parti di esse, nei comandi DOS. Esso scandisce la propria command line alla ricerca di sequenze note di simboli, che definiamo, per comodità, *macro*, e le sostituisce con la parte di data o ora che rappresentano, valorizzata in base a data e ora di sistema. Ogni macro è costituita dal carattere '@', seguito da una lettera che identifica il valore di sostituzione: così, ad esempio, @M indica il giorno del mese, espresso con due cifre. Le macro ammesse ed il loro significato sono elencati di seguito.

@@ il carattere *at* (@); utile per inserire una '@' nella command line

[448] In poche parole: lo standard output di DIR non viene visualizzato, bensì è trasformato dal DOS in standard input per il comando successivo (in questo caso FIND).

@' le virgolette ("); utile per inserire le virgolette nella command line

@A l'anno; espresso con quattro cifre (es.: 1994)

@a l'anno; espresso con due sole cifre (es.: 94)

@M il mese; espresso con due cifre (es.: 07)

@m il mese; espresso con una sola cifra se la prima è 0 (es.: 7)

@G il giorno del mese; espresso con due cifre (es.: 05)

@g il giorno del mese; espresso con una sola cifra se la prima è 0 (es.: 5)

@R il giorno dell'anno; espresso con tre cifre (es.: 084)

@r il giorno dell'anno; espresso con una o due cifre se le prime sono 0 (es.: 84)

@O l'ora; espressa con due cifre (es.: 02)

@o l'ora; espressa con una sola cifra se la prima è 0 (es.: 2)

@I il minuto; espresso con due cifre (es.: 06)

@i il minuto; espresso con una sola cifra se la prima è 0 (es.: 6)

@S il secondo; espresso con due cifre (es.: 01)

@s il secondo; espresso con una sola cifra se la prima è 0 (es.: 1)

@E il giorno della settimana; è indicato il nome intero (es.: Lunedì)

@e il giorno della settimana; indicato mediante i primi tre caratteri del nome (es.: Lun)

@W il giorno della settimana; espresso con un numero da 0 (Domenica) a 6 (Sabato)

Vediamo un esempio pratico: se il comando

```
datecmd echo Sono le @O:@I:@S di @E @g/@M/@a, @r^ giorno dell'anno: @'Ciao, @@!'@
```

viene eseguito alle 14:52:20 del 2 novembre 1994, DATECMD esegue in realtà il comando

```
echo Sono le 14:52:20 di Mercoledì 2/11/94, 306^ giorno dell'anno: "Ciao, @!"
```

DATECMD accetta inoltre due opzioni, -d e -v, che devono precedere la command line da eseguire. La prima consente di specificare uno "slittamento" di data, positivo o negativo. Se nel comando dell'esempio precedente si antepone -v-1 alla command line di DATECMD (cioè a echo), l'ouput prodotto diventa

```
echo Sono le 14:52:20 di Martedì 1/11/94, 305^ giorno dell'anno: "Ciao, @!"
```

L'opzione -v, invece, richiede a DATECMD di visualizzare soltanto, ma non eseguire, la command line risultante a seguito della risoluzione delle macro.

Segue il listato, ampiamente commentato, del programma (vedere pag. 479 e seguenti circa l'implementazione di PARSEOPT.OBJ; le funzioni di manipolazione delle date sono descritte a pag. 560).

```
/*--------------------------------------------------------------------------------

    DATECMD.C - Barninga_Z! - 27/04/1993

    Esegue command lines DOS con la possibilita' di parametrizzarle rispetto
    alla data e all'ora: nella command line, digitata dopo il nome del
    programma (DATECMD), possono essere inserite delle macro, costituite dal
    carattere macro (@) e da uno dei caratteri elencati nel sorgente (nella
    definizione dell'array delle macro). Dette macro vengono espanse nella
    stringa con il corrispondente significato. In ogni command line possono
    comparire piu' macro, ed ogni macro può comparire piu' volte. La macro
    @' viene espansa nelle virgolette ("); la macro @@ viene espansa nella
    atsign (@). L'utilita' di queste macro e' evidente nel caso in cui si
    debbano inserire le virgolette nella command line o nel caso in cui una
    sequenza di caratteri che non deve essere espansa comprenda in casualmente
    i caratteri costituienti una macro. Ad esempio, @M viene espanso nel mese
    corrente; per non espanderlo bisogna digitare @@M (@@ viene espanso in
    @ e la M non viene modificata). Se la command line e' preceduta dalla
    opzione -v essa viene solo visualizzata e non eseguita. Se è preceduta
    da -d si possono specificare i giorni di differenza rispetto alla data
    attuale (+|- gg).
    _stklen e _heaplen sono settate per ridurre l'ingombro in memoria del
    programma, visto che la command line e' eseguita con una system(). Per
    questo motivo, inoltre, l'interprete dei comandi deve essere accessibile.

    Compilato sotto Borland C++ 3.1:

    bcc -k- -O -d datecmd.c parseopt.obj date2jul.obj jul2date.obj isleapyr.obj

--------------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <process.h>
#include <dir.h>
#include <errno.h>
#include <ctype.h>

#include "parseopt.h"        // per la gestione delle opzioni di command line

#define  _PRG_               "DATECMD"     // nome del programma
#define  _VERSION_           "1.5"         // versione
#define  _YEAR_              "94"          // anno di rilascio
#define  MAXCMD              128           // max lungh. cmd line (ENTER incl.)
#define  MACRO_FLAG          '@'           // carattere che segnala la macro
#define  DQUOTE_CHR          '\x27'        // sostituisce le virgolette
#define  SWITCH              '-'           // switch character per opzioni
#define  YEARLEN             365           // giorni dell'anno non bisestile
#define  WEEKLEN             7             // giorni della settimana
#define  WDAYSHORTLEN        3             // lung. stringa breve giorno sett.

//--------------------------------------------------------------------------------
// prototipi delle funzioni del corpo del programma
//--------------------------------------------------------------------------------

long date2jul(int day,int month,int year);
```

```
int isleapyear(int year);
int jul2date(long jul,int *day,int *month,int *year);

int  main(int argc,char **argv);

void  adjustOrdinal(int baseYear,int shift);
void  adjustWeekDay(int len);
void  fatalError(int errNum);
void  help(void);
char *initProg(int argc,char **argv);
int   isDigitStr(char *string);
char *parseCmd(char *cmdArgs);

//--------------------------------------------------------------------------------
// Ottimizzazione dell'uso della memoria
//--------------------------------------------------------------------------------

extern unsigned _heaplen = 4096;
extern unsigned _stklen  = 2048;

//--------------------------------------------------------------------------------
// Messaggi di errore e di aiuto
//--------------------------------------------------------------------------------

#define  _E_BADARGS          0
#define  _E_CMDLONG          1
#define  _E_BADOPTIONS       2
#define  _E_ALLOCMEM         3

char *weekdays[] = {
    "Domenica",
    "Lunedì",
    "Martedì",
    "Mercoledì",
    "Giovedì",
    "Venerdì",
    "Sabato",
};

char *errors[] = {
    "E00: Numero di argomenti errato",
    "E01: La Command Line risultante è troppo lunga",
    "E02: Opzioni errate",
    "E03: Memoria insufficiente",
};

char *helptext = "\
Uso: DATECMD [-v][-d+|-g] CmdLine\n\
CmdLine è la command line da eseguire (-v visualizza soltanto, -d modifica la\n\
data di +/- g giorni); può contenere una o più delle seguenti macro:\n\
";

//--------------------------------------------------------------------------------
// Variabili globali per inizializzazione e altri scopi
//--------------------------------------------------------------------------------

struct tm *tData;                  // contiene data e ora correnti allo startup

char tStr[20];                     // buffer per sprintf() temporanee

//--------------------------------------------------------------------------------
// gruppo delle funzioni per il macro processing. Ciascuna di esse viene
// invocata se e' incontrata la macro corrispondente (vedere l'array macros)
// e restituisce un puntatore a stringa, che deve essere passato a strcpy()
```

```
// per copiare l'espansione della macro nella stringa di comando che sara'
// passata a spawn()
//-----------------------------------------------------------------------------

char *retMacroFlag(void)              // restituisce il carattere MACRO_FLAG
{
    extern char tStr[];

    sprintf(tStr,"%c",MACRO_FLAG);
    return(tStr);
}

char *retDoubleQuote(void)            // restituisce il carattere '"'
{
    sprintf(tStr,"%c",'\x22');
    return(tStr);
}

char *getYear(void)                   // anno, 4 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d%02d",tData->tm_year < 80 ? 20 : 19,tData->tm_year);
    return(tStr);
}

char *getYear1(void)                  // anno, 2 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d",tData->tm_year);
    return(tStr);
}

char *getMonth(void)                  // mese, 2 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d",tData->tm_mon);
    return(tStr);
}

char *getMonth1(void)                 // mese, 1 o 2 cifre
{
    extern char tStr[];

    getMonth();
    if(*tStr == '0')
        return(tStr+1);
    return(tStr);
}

char *getDay(void)                    // giorno del mese, 2 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d",tData->tm_mday);
    return(tStr);
}
```

```
char *getDay1(void)                    // giorno del mese, 1 o 2 cifre
{
    extern char tStr[];

    getDay();
    if(*tStr == '0')
        return(tStr+1);
    return(tStr);
}

char *getOrdinal(void)                 // giorno dell'anno, 3 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%03d",tData->tm_yday);
    return(tStr);
}

char *getOrdinal1(void)                // giorno dell'anno, 1 o 2 o 3 cifre
{
    register i;
    extern char tStr[];

    getOrdinal();
    for(i = 0; tStr[i] == '0'; )
        i++;
    return(tStr+i);
}

char *getHour(void)                    // ora, 2 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d",tData->tm_hour);
    return(tStr);
}

char *getHour1(void)                   // ora, 1 o 2 cifre
{
    extern char tStr[];

    getHour();
    if(*tStr == '0')
        return(tStr+1);
    return(tStr);
}

char *getMin(void)                     // minuto, 2 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d",tData->tm_min);
    return(tStr);
}

char *getMin1(void)                    // minuto, 1 o 2 cifre
{
    extern char tStr[];

    getMin();
    if(*tStr == '0')
```

```
        return(tStr+1);
    return(tStr);
}

char *getSec(void)                      // secondo, 2 cifre
{
    extern struct tm *tData;
    extern char tStr[];

    sprintf(tStr,"%02d",tData->tm_sec);
    return(tStr);
}

char *getSec1(void)                     // secondo, 1 o 2 cifre
{
    extern char tStr[];

    getSec();
    if(*tStr == '0')
        return(tStr+1);
    return(tStr);
}

char *getWeekDay(void)                  // giorno della settimana
{
    extern char *weekdays[];

    return(weekdays[tData->tm_wday]);
}

char *getWeekDay1(void)                 // giorno della settimana 3 lettere
{
    extern char *weekdays[];

    weekdays[tData->tm_wday][WDAYSHORTLEN] = NULL;
    return(getWeekDay());
}

char *getWeekDayNum(void)               // giorno della settimana numero
{
    extern char tStr[];

    sprintf(tStr,"%1d",tData->tm_wday);
    return(tStr);
}

//--------------------------------------------------------------------------------
// definizione della struttura di gestione delle macro come tipo di dato e
// dichiarazione dell'array di strutture che definisce tutte le macro
//--------------------------------------------------------------------------------

typedef struct {              // struttura per la gestione delle macro
    char symbol;                // simbolo della macro
    char *(*replacer)(void);    // funz. che sostituisce il simbolo col signif.
    char *comment;              // commento; usato nello help
} MACRO;

MACRO macros[] = {
    {MACRO_FLAG,retMacroFlag,"il carattere macro"},
    {DQUOTE_CHR,retDoubleQuote,"le virgolette"},
    {'A',getYear,"l'anno; quattro cifre"},
    {'a',getYear1,"l'anno; due sole cifre"},
    {'M',getMonth,"il mese; sempre due cifre"},
    {'m',getMonth1,"il mese; una sola cifra se la prima è 0"},
```

```
    {'G',getDay,"il giorno del mese; sempre due cifre"},
    {'g',getDay1,"il giorno del mese; una sola cifra se la prima è 0"},
    {'R',getOrdinal,"il giorno dell'anno; sempre tre cifre"},
    {'r',getOrdinal1,"il giorno dell'anno; una o due cifre se le prime sono 0"},
    {'O',getHour,"l'ora; sempre due cifre"},
    {'o',getHour1,"l'ora; una sola cifra se la prima è 0"},
    {'I',getMin,"il minuto; sempre due cifre"},
    {'i',getMin1,"il minuto; una sola cifra se la prima è 0"},
    {'S',getSec,"il secondo; sempre due cifre"},
    {'s',getSec1,"il secondo; una sola cifra se la prima è 0"},
    {'E',getWeekDay,"il giorno della settimana; nome intero"},
    {'e',getWeekDay1,"il giorno della settimana; primi tre caratteri del nome"},
    {'W',getWeekDayNum,"il giorno della settimana; numero (0 = dom; 6 = sab)"},
    {NULL,NULL,NULL}
};

//-----------------------------------------------------------------------------
// organizzazione delle opzioni
//-----------------------------------------------------------------------------

#define  DISPLAY_ONLY        1         // opzione solo visualizza cmd line
#define  DAY_SHIFT           2         // opzione shift data in giorni

unsigned options;               // variabile per contenere i bits delle opzioni

char *optionS = "d:v";                 // stringa definizione opzioni

#pragma  warn -par
#pragma  warn -rvl

// l'opzione -d consente di specificare uno scostamento in giorni dalla data
// corrente. Lo scostamento puo' essere positivo (es: 1 = doamni) o negativo
// (es: -1 = ieri). La funzione esegue tutti i controlli formali e modifica
// di conseguenza i dati di lavoro.

int valid_d(struct OPT *vld,int cnt)                     // convalida opzione d
{
    register len, baseYear;
    long jul;
    extern struct tm *tData;
    extern unsigned options;

    len = strlen(vld->arg);
    if((len < 2) || (len > 4) || ((*vld->arg != '-') && (*vld->arg != '+')) ||
                                                        (!isDigitStr(vld->arg+1)))
        fatalError(_E_BADOPTIONS);
    jul = date2jul(tData->tm_mday,tData->tm_mon,baseYear = tData->tm_year);
    jul += (len = atoi(vld->arg));
    jul2date(jul,&tData->tm_mday,&tData->tm_mon,&tData->tm_year);
    adjustOrdinal(baseYear,len);
    adjustWeekDay(len);
    return(options |= DAY_SHIFT);
}

// l'opzione -v forza DATECMD a non eseguire la command line costruita con
// l'espansione delle macro, bensi' a visualizzarla solamente. Qui viene
// settato il flag che indica che l'opzione e' stata richiesta

int valid_v(struct OPT *vld,int cnt)                     // convalida opzione v
{
    extern unsigned options;

    return(options |= DISPLAY_ONLY);
}
```

```
// gestione delle opzioni specificate in modo errato

int err_handle(struct OPT *vld,int cnt)                  // opzioni errate
{
    help();
    fatalError(_E_BADOPTIONS);
}

#pragma  warn .par
#pragma  warn .rvl

// array che associa ogni ozione alla corrispondente funzione di validazione

static struct VOPT vfuncs[] = {
    {'d',valid_d},
    {'v',valid_v},
    {ERRCHAR,err_handle},
    {NULL,NULL}
};


//-----------------------------------------------------------------------------
// corpo del programma (dopo main() le funzioni sono in ordine alfabetico)
//-----------------------------------------------------------------------------

int main(int argc,char **argv)                     // pilota tutte le operazioni
{
    char *initArgs;
    char *cmdLine;
    extern unsigned options;
    extern char *sys_errlist[];

    printf("%s %s - Esegue command lines con macro data/ora - Barninga_Z!
'%s\n",_PRG_,_VERSION_,_YEAR_);
    initArgs = initProg(argc,argv);
    cmdLine = parseCmd(initArgs);
    printf("\n%s\n\n",cmdLine);
    if(!(options & DISPLAY_ONLY))
        if(system(cmdLine))
            return(printf("%s: %s\n",_PRG_,sys_errlist[errno]));
    return(0);
}

// adjustOrdinal() modifica il numero di giorno nell'anno (tm_yday) in base
// al numero di giorni di shift della data (opzione -d), tenendo presente che
// lo shift potrebbe generare un cambiamento di anno in piu' o in meno.

void adjustOrdinal(int baseYear,int shift)
{
    register diff, year;
    extern struct tm *tData;

    year = tData->tm_year;
    tData->tm_yday += shift;
    if(tData->tm_yday <= 0) {
        for(diff = 0; year < baseYear; year++)
            diff += YEARLEN+isleapyear(year);
        tData->tm_yday += diff;
    }
    else {
        for(diff = 0; baseYear < year; baseYear++)
            diff += YEARLEN+isleapyear(baseYear);
        tData->tm_yday -= diff;
```

```
    }
}

// adjustWeekDay() modifica il numero di giorno nella settimana (tm_wday) in
// base al numero di giorni di shift della data (opzione -d), tenendo presente
// che lo shift potrebbe generare un cambiamento di settimama.

void adjustWeekDay(int shift)
{
    register temp = 0;
    extern struct tm *tData;

    if((tData->tm_wday += shift) < 0) {
        tData->tm_wday = -tData->tm_wday;
        temp = 1;
    }
    tData->tm_wday %= WEEKLEN;
    if(temp && tData->tm_wday)
        tData->tm_wday = WEEKLEN-tData->tm_wday;
}

// fatalError() esce a DOS quando si verifica un errore, visualizzando un
// messaggio dall'array errors.

void fatalError(int errNum)
{
    printf("%s: %s",_PRG_,errors[errNum]);
    exit(1);
}

// help() stampa una videata di aiuto usando il campo comment di dell'array
// di strutture MACRO macros per visualizzare la descrizioni delle macro

void help(void)
{
    register i;
    extern MACRO macros[];
    extern char *helptext;

    printf(helptext);
    for(i = 0; macros[i].symbol; i++)
        printf("@%c => %s\n",macros[i].symbol,macros[i].comment);
}

// initProg() controlla i parametri e concatena gli elementi di argv per
// ricostruire la command line di DATECMD in un'unica stringa.

char *initProg(int argc,char **argv)
{
    register i;
    long timebits;
    struct OPT *optn;
    static char initArgs[MAXCMD];
    extern struct tm *tData;

    switch(argc) {
        case 1:
            help();
            fatalError(_E_BADARGS);
        default:
            time(&timebits);
            tData = localtime(&timebits);
            ++tData->tm_yday;                                       // 1 - 366
            ++tData->tm_mon;                                        // 1 - 12
```

```
            if(!(optn = parseopt(argc,argv,optionS,SWITCH,NULL,NULL,vfuncs)))
                fatalError(_E_ALLOCMEM);
    }
    *initArgs = NULL;
    for(i = argc-optn[0].val; ; ) {
        strcat(initArgs,argv[i]);
        if(++i < argc)
            strcat(initArgs," ");
        else
            break;
    }
    return(initArgs);
}

// isDigitStr() controlla se una stringa contiene solo 0-9

int isDigitStr(char *string)
{
    for(; *string; string++)
        if(!isdigit(*string))
            return(NULL);
    return(1);
}

// parseCmd() ricerca ed espande le macro presenti nella command line di
// DATECMD, costruendo il comando da eseguire, nel quale compaiono, in luogo
// delle macro, gli elementi di data e ora desiderati.

char *parseCmd(char *cmdArgs)
{
    register mIndex, i, j;
    char *ptr;
    static char cmdLine[MAXCMD];
    extern MACRO macros[];

    *cmdLine = NULL;
    for(i = 0, j = 0; cmdArgs[i]; i++) {
        switch(cmdArgs[i]) {
            case MACRO_FLAG:
                for(mIndex = 0; macros[mIndex].symbol; mIndex++) {
                    if(macros[mIndex].symbol == cmdArgs[i+1])
                        break;
                }
                if(macros[mIndex].symbol) {
                    ptr = (macros[mIndex].replacer)();
                    if((j += strlen(ptr)) < MAXCMD) {
                        strcat(cmdLine,ptr);
                        ++i;
                    }
                    else
                        fatalError(_E_CMDLONG);
                    break;
                }
            default:
                cmdLine[j++] = cmdArgs[i];
        }
    }
    return(cmdLine);
}
```

DATECMD può rivelarsi particolarmente utile quando vi sia la necessità di dare ad un file, mediante i comandi DOS COPY o REN, un nome dipendente dalla data e ora in cui l'operazione stessa è effettuata. Si pensi, ad esempio, ad una procedura che, quotidianamente, scrive il risultato delle proprie

elaborazioni nel file OUTPUT.DAT: qualora sia necessario conservare a lungo i file generati, può risultare comodo riservare loro una directory e copiarveli di giorno in giorno, rinominandoli in modo appropriato. Il solo modo di inserire il comando in un file batch, senza necessità alcuna di intervento interattivo da parte dell'utente, consiste nel servirsi di DATECMD:

```
datecmd copy d:\proc\output.dat c:\proc\out\@a@M@G.dat
```

Se ogni file deve essere conservato per una settimana soltanto, il comando presentato necessita appena un piccolo aggiustamento:

```
datecmd copy d:\proc\output.dat c:\proc\out\output.@e
```

Infatti, dal momento che la macro @e viene sostituita dai tre caratteri iniziali del nome del giorno della settimana, è evidente che ogni giorno il nuovo file sovrascrive quello copiato sette giorni prima.

Va ancora sottolineato che DATECMD consente una gestione avanzata di piping e redirezione: ad esempio, il comando

```
datecmd copy d:\proc\output.dat c:\proc\out\@a@M@G.dat >> batch.ctl
```

redirige lo standard output di DATECMD in coda al file BATCH.CTL, mentre il comando

```
datecmd "copy d:\proc\output.dat c:\proc\out\@a@M@G.dat >> batch.ctl" > nul
```

aggiunge a BATCH.CTL lo standard output del comando COPY e sopprime quello di DATECMD; si noti l'uso delle virgolette, indispensabili per evitare che il DOS interpreti la redirezione facente parte della command line che deve essere eseguita da DATECMD[449].

File piccoli a piacere: FCREATE

Gli esempi presentati poco sopra nascondono una insidia: il comando DOS COPY fallisce se il file origine ha dimensione pari a 0 byte; la conseguenza è che nella directory dedicata alla memorizzazione dei file di output potrebbero mancarne alcuni[450].

FCREATE aggira il problema, consentendo la creazione di un file della dimensione voluta. Il file generato, qualora abbia dimensione maggiore di 0, contiene esclusivamente byte nulli (zero binario).

```
/******************************************************************************

    FCREATE.C - Barninga Z! - 27/10/1994

    Crea un file della dimensione indicata sulla riga di comando.

    Compilato sotto Borland C++ 3.1
```

[449] DATECMD esegue la propria command line effettuando una *DOS shell*, cioè invocando una seconda istanza (transiente) dell'interprete dei comandi. Ne segue che COMMAND.COM (o, comunque, l'interprete utilizzato) deve essere disponibile (nella directory corrente o in una di quelle elencate nella variabile d'ambiente PATH) e deve esserci memoria libera in quantità sufficiente per il caricamento dell'interprete stesso e per l'esecuzione del comando da parte di questo. Vedere pag. 129.

[450] Può trattarsi di un inconveniente grave qualora, ad esempio, altre procedure abbiano necessità di accedere comunque ai file (o, quanto meno, di verificarne l'esistenza) per operare correttamente.

```
    bcc fcreate.c

******************************************************************************/
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys\stat.h>

#define  PRG       "FCREATE"
#define  VER       "1.0"
#define  YEAR      "94"

#define  OPENMODE    O_WRONLY+O_CREAT+O_TRUNC+O_BINARY
#define  OPENATTR    S_IREAD+S_IWRITE

#define  E_SYNTAX    1
#define  E_OPEN      2
#define  E_CHSIZE    3
#define  E_CLOSE     4
#define  E_PARM      5

// tutte le operazioni sono effettuate in main()

int main(int argc,char **argv)
{
    register handle, retcode;
    long size;

    fprintf(stderr,"%s %s - Creates file(s) of given size - Barninga Z! '%s\n",PRG,
                                                                          VER,YEAR);

// controllo dei parametri della command line

    if(argc != 3) {
        fprintf(stderr,"%s: Syntax is: %s size filename\n",PRG,PRG);
        return(E_SYNTAX);
    }

// controllo del valore di size

    if((size = atol(argv[1])) < 0L) {
        fprintf(stderr,"%s: requested size must be 0 or greater.\n",PRG);
        return(E_PARM);
    }
    retcode = 0;

// apertura del file: se non esiste viene creato; se esiste e' troncato a 0

    if((handle = open(argv[2],OPENMODE,OPENATTR)) == -1) {
        retcode = E_OPEN;
        printf("%s: error opening %s\n",PRG,argv[2]);
    }
    else {

// se il file e' stato aperto regolarmente se ne imposta la dimensione voluta
// con chsize()

        if(chsize(handle,size)) {
            retcode = E_CHSIZE;
            printf("%s: error sizing %s\n",PRG,argv[2]);
        }
        else
```

```
// se l'impostazione della nuova dimensione e' riuscita si calcola la nuova
// dimensione del file per verifica

            size = filelength(handle);

// chiusura del file

        if(close(handle)) {
            retcode = E_CLOSE;
            printf("%s: error closing %s\n",PRG,argv[2]);
        }
    }

// test e azione conseguente in caso di errore o meno

    if(!retcode)
        printf("%s: created %s (%ld bytes)\n",PRG,argv[2],size);
    return(retcode);
}
```

FCREATE richiede due parametri: il primo esprime la dimensione desiderata per il file; il secondo è il nome di questo. Insieme con EMPTYLVL (pag. 565) e DATECMD (pag. 567) è possibile implementare un algoritmo privo di criticità:

```
DIR "D:\PROC\OUTPUT.DAT" | FIND " 0 " | EMPTYLVL
IF ERRORLEVEL 1 GOTO NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE 0 BYTES: UTILIZZO FCREATE...
DATECMD FCREATE 0 C:\PROC\OUT\@a@M@G.OUT
GOTO END
:NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE MAGGIORE DI 0 BYTES: UTILIZZO COPY...
DATECMD COPY D:\PROC\OUTPUT.DAT C:\PROC\OUT\@a@M@G.OUT
:END
```

FCREATE può validamente essere utilizzato per generare file aventi la funzione di *placeholder*, cioè per riservare spazio ad utilizzi futuri.

Attendere il momento buono: TIMEGONE

Per rendere le cose più complicate, assumiamo che il drive D:, sul quale viene prodotto OUTPUT.DAT dalla ormai nota procedura, non sia un disco fisicamente presente nel personal computer su cui lavoriamo, ma si tratti, al contrario, di un drive remoto[451], reso disponibile da una seconda macchina: il server di rete. La procedura, a sua volta, è eseguita su un terzo personal computer, anch'esso collegato al network ed in grado di accedere al medesimo disco condiviso. E' evidente, a questo punto, che non ha alcun senso lanciare il nostro file batch prima che la procedura in questione abbia terminato le proprie elaborazioni. Nell'ipotesi che ciò avvenga poco prima delle 23:00, chi non desideri trascorrere le proprie serate alla tastiera, in trepidante attesa, deve necessariamente adottare qualche provvedimento.

[451] Si dice *remoto* (in contrapposizione a *locale*) un disco appartenente ad una macchina diversa da quella sulla quale si svolge la sessione di lavoro. Se più personal computer sono collegati in rete, ciascuno di essi può utilizzare, in aggiunta alle proprie risorse locali, quelle remote rese disponibili dalle macchine configurate come *server*. L'utilizzo alle risorse remote avviene in modo condiviso, in quanto più personal computer possono accedervi contemporaneamente.

TIMEGONE è... il provvedimento necessario[452]: esso controlla se è trascorsa l'ora (ed eventualmente la data) indicata e termina, valorizzando di conseguenza il registro ERRORLEVEL. In particolare, ERRORLEVEL vale 1 se l'ora è trascorsa; 0 se non lo è.

L'ora da tenere sotto controllo deve essere specificata sulla command line di TIMEGONE nel formato *hhmmss* (due cifre per l'ora, due per i minuti, due per i secondi): ad esempio, il comando

```
timegone 060400
```

indica le ore 6:04. I minuti e i secondi devono essere sempre specificati.

TIMEGONE consente di indicare, in aggiunta all'ora, anche una data: allo scopo sono riconosciute tre opzioni di command line: -d consente di specificare il giorno, sempre espresso con due cifre. Ad esempio, il 12 del mese si indica con -d12. L'opzione -m specifica il mese: febbraio, per esempio, si indica con -m02. Infine, l'opzione -y permette di indicare l'anno, in 4 cifre. Il 1994, pertanto, si specifica con -y1994. Il comando

```
timegone -d03 -m11 170000
```

richiede a TIMEGONE di verificare se siano trascorse le ore 17 del 3 novembre: dal momento che l'anno non è specificato, esso non viene controllato (il risultato del test è indipendente dall'anno di elaborazione).

Mentre l'indicazione di giorno, mese e anno è facoltativa, l'ora deve essere sempre specificata. Tuttavia, in luogo dell'espressione in formato *hhmmss* può essere digitato un asterisco ('*') per forzare il programma a ricavare date e ora dalla stringa assegnata alla variabile d'ambiente TIMEGONE: questa ha formato YYYYMMGGhhmmss (l'indicazione dell'ora è preceduta da quattro cifre per l'anno, due per il mese, due per il giorno); è lecito specificare zeri per anno, mese e giorno (sempre in numero di quattro, due e, rispettivamente, ancora due) se si desidera che siano ignorati. La condizione dell'esempio precedente può essere espressa valorizzando la variabile TIMEGONE con il comando

```
set timegone=00001103170000
```

ed eseguendo successivamente

```
timegone *
```

Le opzioni eventualmente specificate sulla riga di comando hanno precedenza rispetto alla variabile d'ambiente; questa, infine, è ignorata se a TIMEGONE è passata, quale parametro, l'ora. Così

---

[452] Qualcosa si può fare anche senza TIMEGONE, con l'aiuto di EMPTYLVL (pag. 565). Innanzitutto si deve creare un file ASCII contenente un CR (ASCII 13), che, per comodità, indichiamo col nome CR.TXT. Occorre poi inserire nella procedura batch una sequenza di istruzioni analoga alla seguente:

```
:ASPETTA
TIME < CR.TXT | FIND "15:09" | EMPTYLVL
IF ERRORLEVEL 1 GOTO ASPETTA
```

Lo standard input del comando TIME è rediretto da CR.TXT, con l'effetto di terminare il comando senza modificare l'ora di sistema; lo standard output è invece rediretto, mediante piping, sullo standard input di FIND, che cerca la stringa contenente l'ora desiderata (nell'esempio le 15:09); infine, EMPTYLVL memorizza 1 in ERRORLEVEL se la stringa non è trovata (l'ora non è quella voluta). I limiti della soluzione descritta sono evidenti: in primo luogo il batch deve essere lanciato necessariamente prima dell'ora specificata, in quanto la stringa non viene trovata anche se quella è già trascorsa; inoltre si tratta di un algoritmo applicabile con difficoltà al comando DATE, in particolare quando si desideri effettuare il test su parte soltanto della data. La utility CUT (presentata a pag. 593) può venire in soccorso, ma si introducono, comunque, complicazioni notevoli al listato qui riprodotto.

```
timegone -d04 *
```

verifica se sono trascorse le 17 del 4 novembre (fermo restando il valore della variabile TIMEGONE dell'esempio precedente).

TIMEGONE riconosce una quarta opzione: -r richiede che il risultato del test sia invertito (ERRORLEVEL vale 1 se data e ora non sono trascorse, 0 altrimenti).

Segue il listato, ampiamente commentato, del programma (vedere pag. 479 e seguenti circa l'implementazione di PARSEOPT.OBJ; le funzioni di manipolazione delle date sono descritte a pag. 560).

```
/******************************************************************************

    TIMEGONE.C - Barninga Z! - 29/09/94

    Setta errorlevel a seconda che la data e l'ora correnti superino o meno
    quelle richieste. Vedere helpStr per i dettagli della sintassi e delle
    opzioni.

    Compilato sotto Borland C++ 3.1

    bcc timegone.c isleapyear.obj parseopt.obj

******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>

#include "parseopt.h"

#define  YEAR       1
#define  MONTH      2
#define  DAY        4
#define  REVERSE    8
#define  SWCHAR     '-'
#define  ILLMSG     "Invalid option"
#define  YEARLEN    4
#define  MONTHLEN   2
#define  DAYLEN     2
#define  DATELEN    (YEARLEN+MONTHLEN+DAYLEN)
#define  TIMELEN    6
#define  DTLEN      (DATELEN+TIMELEN)
#define  TVARSYM    "*"
#define  TVARNAME   "TIMEGONE"

#define  PRG        "TIMEGONE"
#define  REL        "1.0"
#define  YR         "94"
#define  RETERR     255
#define  FALSE      0
#define  TRUE       1
#define  ASCIIBASE  0x30
#define  MIN_YEAR   1980
#define  MAX_YEAR   2099

char *helpStr = "\
\n\
options: one or more of the following:\n\
  -dDD    day DD of the month (always 2 digits).\n\
  -mMM    month MM of the year (always 2 digits).\n\
  -yYYYY  year YYYY (always 4 digits).\n\
  -r      reverse the result.\n\
```

```
\n\
time      time, in hhmmss format (always 6 digits). If time is the string \"*\",\n\
          the environment variable TIMEGONE is used. It must contain a\n\
          datetime string in YYYYMMGGhhmmss format; YYYY=0000, MM=00, GG=00\n\
          cause year, month and day respectively to be ignored.\n\
\n\
Command line parameters, if given, always override the environment variable.\n\
\n\
Errorlevel is set to 1 if the (date and) time specified has gone; it is set to\n\
0 if not yet. The -r option causes the result to be reversed (0 if gone, 1 if\n\
not yet). If an error occurs, Errorlevel is set to 255.\n\n\
";

int isleapyear(int year);

int main(int argc,char **argv);
int hasTimeGone(void);
int isDateValid(void);
int isNumeric(char *string);

int valid_d(struct OPT *vld,int cnt);
int valid_m(struct OPT *vld,int cnt);
int valid_r(struct OPT *vld,int cnt);
int valid_y(struct OPT *vld,int cnt);
int err_handle(struct OPT *vld,int cnt);

#pragma warn -rvl
#pragma warn -par
#pragma warn -aus

// l'opzione -d consente di specificare il giorno per comporre una data da
// controllare insieme all'ora. Qui sono effettuati tutti i controlli formali;
// 00 e' valido e indica che qualsiasi giorno va bene

int valid_d(struct OPT *vld,int cnt)
{
    extern unsigned options;
    extern struct date da;

    if(options & DAY)
        err_handle(NULL,NULL);
    if((strlen(vld->arg) != DAYLEN) || !isNumeric(vld->arg))
        err_handle(NULL,NULL);
    da.da_day = atoi(vld->arg);
    options |= DAY;
}

// l'opzione -m consente di specificare il mese per comporre una data da
// controllare insieme all'ora. Qui sono effettuati tutti i controlli formali;
// 00 e' valido e indica che qualsiasi mese va bene

int valid_m(struct OPT *vld,int cnt)
{
    extern unsigned options;
    extern struct date da;

    if(options & MONTH)
        err_handle(NULL,NULL);
    if((strlen(vld->arg) != MONTHLEN) || !isNumeric(vld->arg))
        err_handle(NULL,NULL);
    da.da_mon = atoi(vld->arg);
    options |= MONTH;
}
```

```
// l'opzione -r rovescia il risultato del test: se il momento e' trascorso
// ERRORLEVEL e' settato a 0 invece che a 1, e viceversa

int valid_r(struct OPT *vld,int cnt)
{
    extern unsigned options;

    options |= REVERSE;
}

// l'opzione -y consente di specificare il anno per comporre una data da
// controllare insieme all'ora. Qui sono effettuati tutti i controlli formali;
// 0000 e' valido e indica che qualsiasi anno va bene

int valid_y(struct OPT *vld,int cnt)
{
    extern unsigned options;
    extern struct date da;

    if(options & YEAR)
        err_handle(NULL,NULL);
    if((strlen(vld->arg) != YEARLEN) || !isNumeric(vld->arg))
        err_handle(NULL,NULL);
    da.da_year = atoi(vld->arg);
    options |= YEAR;
}

// controlla la validita' formale dell'ora indicata sulla command line. Il
// formato deve essere hhmmss. In luogo dell'ora sulla cmdline si puo'
// specificare un asterisco: in tal caso il programma verifica se esiste la
// variabile d'ambiente TIMEGONE e ricava data e ora dalla stringa as essa
// assegnata.

int valid_time(struct OPT *vld,int cnt)
{
    int dummy;
    static int instance;
    extern unsigned options;
    extern struct dostime_t ti;
    extern struct date da;

    if(instance++)
        err_handle(NULL,NULL);              // solo un'ora puo' essere indicata

// se e' specificato l'asterisco...

    if(!strcmp(vld->arg,TVARSYM)) {

// ...si cerca la variabile d'ambiente TIMEGONE: se e' presente se ne assegna
// l'indirizzo a vld->arg, in modo da poter simulare, dopo l'elaborazione
// della data, la digitazione di un'ora sulla command line

        if(!(vld->arg = getenv(TVARNAME))) {
            vld->arg = TVARNAME" environment variable missing";
            err_handle(vld,NULL);
        }

// si effettua il controllo di numericita' sulla stringa YYYYMMDD e si
// valorizzano i campi della struttura date

        if((strlen(vld->arg) != DTLEN) || !isNumeric(vld->arg)) {
            vld->arg = "illegal "TVARNAME" environment variable value";
            err_handle(vld,NULL);
        }
```

```
        if(sscanf(vld->arg,"%4d%02d%02d",
                             (options & YEAR) ? &dummy : (int *)&da.da_year,
                             (options & MONTH) ? &dummy : (int *)&da.da_mon,
                             (options & DAY) ? &dummy : (int *)&da.da_day) < 3)
            err_handle(NULL,NULL);

// vld->arg e' forzato a puntare alla seconda parte della stringa, avente
// formato hhmmss

        vld->arg += DATELEN;
    }
    else

// da qui in poi l'elaborazione e' identica sia nel caso di ora passata sulla
// command line che di utilizzo della variabile TIMEGONE, in quanto, in
// entrambi i casi, vld->arg punta a una stringa in formato hhmmss

        if((strlen(vld->arg) != TIMELEN) || !isNumeric(vld->arg))
            err_handle(NULL,NULL);

// valorizzazione dei campi della struttura dostime_t

    if(sscanf(vld->arg,"%02d%02d%02d",(int *)&ti.hour,
                                      (int *)&ti.minute,
                                      (int *)&ti.second) < 3)
        err_handle(NULL,NULL);

// controllo di validita' formale della data

    if(!isDateValid())
        err_handle(NULL,NULL);

// controllo di validita' formale dell'ora

    if((ti.hour > 23) || (ti.minute > 59) || (ti.second > 59))
        err_handle(NULL,NULL);
}

// gestione degli errori: visualizzazione della stringa di help e del valore
// di ERRORLEVEL

int err_handle(struct OPT *vld,int cnt)
{
    extern char *helpStr;

    if(vld)
        fprintf(stderr,"%s: %s.\n",PRG,vld->arg);
    fprintf(stderr,"%s: Usage: %s [option(s)] time | *\n%s",PRG,PRG,helpStr);
    printf("%s: errorlevel set to %d.\n",PRG,RETERR);
    exit(RETERR);                                            // non togliere!!
}

#pragma warn .par
#pragma warn .rvl
#pragma warn .aus

unsigned options;

// ogni opzione e' associata alla corrispondente funzione di validazione

struct VOPT valfuncs[] = {
    {'d',valid_d},
    {'m',valid_m},
    {'r',valid_r},
```

```
    {'y',valid_y},
    {ERRCHAR,err_handle},
    {NULL,valid_time},
};

// elenco delle opzioni: quelle seguite da ':' vogliono un argomento

char *optionS = "d:m:ry:";

struct date da;
struct dostime_t ti;

// main() analizza le opzioni via parseopt() ed effettua il test su data/ora
// via hasTimeGone()

int main(int argc,char **argv)
{
    register retval;

    printf("%s %s - Test if (date/)time has gone - Barninga Z! '%s.\n",PRG,REL,YR);
    if(argc < 2)
        err_handle(NULL,NULL);
    if(!parseopt(argc,argv,optionS,SWCHAR,ILLMSG,ILLMSG,valfuncs)) {
        perror(PRG);
        retval = RETERR;
    }
    else {
        retval = hasTimeGone();
        if(options & REVERSE)
            if(retval)
                retval = 0;
            else
                retval = 1;
    }
    printf("%s: errorlevel set to %d.\n",PRG,retval);
    return(retval);
}

// controlla se la data/ora attuale ha superato la data/ora specificate via
// opzioni e/o variabile d'ambiente

int hasTimeGone(void)
{
    char datetimeNow[DTLEN+1], datetime[DTLEN+1];
    struct date daNow;
    struct dostime_t tiNow;
    extern struct date da;
    extern struct dostime_t ti;

// richiesta a sistema di data e ora attuali

    getdate(&daNow);
    _dos_gettime(&tiNow);

// valorizzazione della stringa in formato YYYYMMDDhhmmss con data/ora attuali

    sprintf(datetimeNow,"%4d%02d%02d%02d%02d%02d",daNow.da_year,
                                                  daNow.da_mon,
                                                  daNow.da_day,
                                                  tiNow.hour,
                                                  tiNow.minute,
                                                  tiNow.second);

// valorizzazione stringa YYYYMMDDhhmmss con data/ora richieste, tenendo conto
```

```
// che un valore di 0 per anno, mese o giorno deve essere sostituito con oggi

    sprintf(datetime,"%4d%02d%02d%02d%02d%02d",da.da_year?da.da_year:daNow.da_year,
                                           da.da_mon ?da.da_mon :daNow.da_mon,
                                           da.da_day ?da.da_day :daNow.da_day,
                                           ti.hour,
                                           ti.minute,
                                           ti.second);

// confronto tra le due stringhe per determinare se il momento e' trascorso.
// Il confronto tra stringhe evita di confrontare uno a uno i campi numerici
// incrociando i risultati

    if(strcmp(datetimeNow,datetime) < 0)
        return(0);
    return(1);
}

// isDateValid() controlla la correttezza formale della data, verificando che
// per anno, mese e giorno siano verificati valori plusibili

int isDateValid(void)
{
    static int monLenTbl[] = {31,28,31,30,31,30,31,31,30,31,30,31};
    extern struct date da;

// se il gmese e' < 0 o > 12 errore

    if((da.da_mon < 0 ) || (da.da_mon > 12))
        return(0);

// se l'anno non e' 0 deve essere compreso tra gli estremi di validita'; se lo
// e' si determina se e' bisestile

    if(da.da_year) {
        if((da.da_year < MIN_YEAR) || (da.da_year > MAX_YEAR))
            return(0);
        else
            monLenTbl[1] += isleapyear(da.da_year);
    }
    else

// se l'anno e' 0 occorre ammettere che potrebbe essere bisestile, chissa'...

        monLenTbl[1] += 1;

// se il giorno non e' 0 esso deve essere positivo e minore o uguale al numero
// di giorni del mese; se il mese e' 0 bisogna ammettere che il 31 e' valido

    if(da.da_day) {
        if(da.da_day < 0)
            return(0);
        if(da.da_mon) {
            if(da.da_day > monLenTbl[da.da_mon-1])
                return(0);
        }
        else
            if(da.da_day > 31)                   // sempre e comunque errore!!
                return(0);
    }
    return(1);                                   // 1 = data OK
}

// controlla la numericita' della stringa ricevuta come parametro
```

```
int isNumeric(char *string)
{
    register i;

    for(; *string; string++) {
        for(i = (ASCIIBASE+0); i < (ASCIIBASE+10); i++)
            if(*string == i)
                break;
        if(i == (ASCIIBASE+10))
            return(FALSE);
    }
    return(TRUE);
}
```

Vediamo, finalmente, `TIMEGONE` al lavoro nel nostro esempio pratico:

```
:ATTESA
TIMEGONE 230000
IF ERRORLEVEL 1 GOTO ADESSO
GOTO ATTESA
:ADESSO
ECHO LE ORE 23:00 SONO APPENA TRASCORSE
DIR "D:\PROC\OUTPUT.DAT" | FIND " 0 " | EMPTYLVL
IF ERRORLEVEL 1 GOTO NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE 0 BYTES: UTILIZZO FCREATE...
DATECMD FCREATE 0 C:\PROC\OUT\@a@M@G.OUT
GOTO END
:NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE MAGGIORE DI 0 BYTES: UTILIZZO COPY...
DATECMD COPY D:\PROC\OUTPUT.DAT C:\PROC\OUT\@a@M@G.OUT
:END
```

Il batch può essere eseguito a qualsiasi ora: `TIMEGONE` forza un loop sulle prime quattro righe del file sino alle 23:00 (se il batch è lanciato dopo le 23 il flusso elaborativo salta immediatamente all'etichetta `:ADESSO`).

Estrarre una sezione da un file: SELSTR

Sempre più difficile: il famigerato `OUTPUT.DAT` contiene una sezione di testo che costituisce la base per alcune postelaborazioni. Ipotizziamo che essa abbia il seguente layout:

```
* RIEPILOGO *
....
* FINE *
```

Il contenuto della sezione, per il momento, non è rilevante: l'obiettivo è estrarla in modo automatico da `OUTPUT.DAT` e scriverla in un secondo file, da processare in seguito. La utility `SELSTR` rappresenta una soluzione: il comando

```
selstr "* RIEPILOGO *" "* FINE *" < output.dat > riepilog.dat
```

scrive sullo standard output le righe di testo, lette dallo standard input, a partire da quella contenente la prima stringa specificata e conclude l'estrazione con la riga contenente la seconda. Ancora una volta, perciò, è possibile sfruttare le capacità di redirezione offerte dal sistema operativo.

Tuttavia, `SELSTR` è in grado di fare di più. Esso, infatti, accetta sulla riga di comando alcune opzioni tramite le quali è possibile modificarne il comportamento, indicando se l'estrazione debba iniziare

o terminare ad una riga specificata: per la descrizione dettagliata delle opzioni e dei loro parametri si veda il testo assegnato alla variabile `helpStr`, nel listato del programma. Qui vale la pena di soffermarsi sull'opzione `-c`, che consente di determinare la modalità di *caching*[453] degli stream. In particolare, `-ci` richiede a `SELSTR` di attivare il caching dello standard input; `-co` richiede l'attivazione del caching per lo standard output; infine, `-cb` attiva il caching per entrambi gli stream. L'uso attento dell'opzione `-c` permette di ottenere un significativo incremento della efficienza di elaborazione.

Segue il listato, ampiamente commentato, del programma (vedere pag. 479 e seguenti circa l'implementazione di `PARSEOPT.OBJ`).

```
/******************************************************************************

    SELSTR.C - Barninga Z! - 20/09/94

    Seleziona una porzione di file in base a stringhe specificate come
    delimitatori di inizio o fine. Vedere helpStr per i dettagli.

    Compilato sotto Borland C++ 3.1

    bcc selstr.c parseopt.obj

******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

#include "parseopt.h"

#define  PRG        "SELSTR"
#define  VER        "1.0"
#define  YEAR       "94"
#define  MAXLIN     1024
#define  RET_ERR    255
#define  BUFSIZE    16348

#define  FROMNUM    1
#define  FROMBEG    2
#define  FROMLIN    4
#define  TONUM      8
#define  CACHE_I    16
#define  CACHE_O    32
#define  FROMBEG_C  'b'
#define  FROMLIN_C  'l'
#define  CACHE_I_C  'i'
#define  CACHE_O_C  'o'
#define  CACHE_B_C  'b'
#define  SWCHAR     '-'
#define  ILLMSG     "Invalid option"

int main(int argc,char **argv);
int search(char *string1,char *string2);

int valid_c(struct OPT *vld,int cnt);
int valid_f(struct OPT *vld,int cnt);
int valid_t(struct OPT *vld,int cnt);
int err_handle(struct OPT *vld,int cnt);
```

[453] Si tratta di una tecnica di ottimizzazione delle operazioni di I/O, implementata mediante algoritmi, più o meno sofisticati, di gestione di buffer, che consentono di limitare il numero di accessi alle periferiche hardware (con particolare riferimento ai dischi). Sull'argomento vedere pag. 124.

```
int ctl_strings(struct OPT *vld,int cnt);

unsigned long startNum = 1L;
unsigned long stopNum = 0xFFFFFFFFL;

char *helpStr = "\
options:\n\
-c: cache data streams:\n\
     -ci: cache standard input\n\
     -co: cache standard output\n\
     -cb: cache both stdin and stdout\n\
-f: line selection origin mode:\n\
     -fn: from linenumber: lines are selected from line n (n > 0) to the\n\
          line containing string1 or to the line defined by -t, whichever\n\
          comes first.\n\
     -fb: from beginning: lines are selected from BOF to the line\n\
          containing string1 or to the line defined by -t, whichever comes\n\
          first. Same as -f1.\n\
     -fl: from line (default mode): lines are selected from the line\n\
          containing string1 to EOF. Selection stops at the line\n\
          containing string2, if given and found, or at the line defined by -t.\n\
-t: line selection end mode:\n\
     -tn: to linenumber: lines are selected from the origin defined by -f\n\
          to line n (n > 0). Anyway, selection stops at the line\n\
          containing string1 (or string2 for -fl), if found.\n\
ErrorLevel: 255: error;\n\
              1: ok, one or more lines selected;\n\
              0: ok, no lines selected.\
";

unsigned options = FROMLIN;

#pragma warn -rvl
#pragma warn -par
#pragma warn -aus

// validazione dell'opzione -c per la richiesta di caching degli streams.
// Sono attivati i buffers di cache richiesti

int valid_c(struct OPT *vld,int cnt)
{
    static int instance;
    extern unsigned options;

    if(instance++ || (strlen(vld->arg) != 1))
        err_handle(NULL,NULL);
    switch(*vld->arg) {
        case CACHE_B_C:
        case CACHE_I_C:
            if(setvbuf(stdin,NULL,_IOFBF,BUFSIZE)) {
                perror(PRG);
                exit(RET_ERR);
            }
            options |= CACHE_I;
            if(*vld->arg == CACHE_I_C)
                break;
        case CACHE_O_C:
            if(setvbuf(stdout,NULL,_IOFBF,BUFSIZE)) {
                perror(PRG);
                exit(RET_ERR);
            }
            options |= CACHE_O;
            break;
        default:
```

```
            err_handle(NULL,NULL);
    }
}

// l'opzione -f consente di specificare quale deve essere la prima linea
// copiata da stdin a stdout: quella contenente il primo nonoption item
// (default), quella il cui numero e' specificato come argomento, o la prima
// riga del file

int valid_f(struct OPT *vld,int cnt)
{
    char *ptr;
    extern unsigned options;
    static int instance;

    if(instance++)
        err_handle(NULL,NULL);
    for(ptr = vld->arg; *ptr; ptr++)
        if(!isdigit(*ptr))
            if(strlen(vld->arg) > 1)
                err_handle(NULL,NULL);
            else {
                switch(*vld->arg) {
                    case FROMLIN_C:
                        break;                  // FROMLIN settato per default
                    case FROMBEG_C:
                        options &= ~FROMLIN;
                        options |= FROMBEG;
                        break;                  // startNum = 1 per default
                    default:
                        err_handle(NULL,NULL);
                }
                return;
            }
    startNum = atol(vld->arg);
    options &= ~FROMLIN;
    options |= FROMNUM;
}

// l'opzione -t consente di specificare quale deve essere l'ultima riga del
// file copiata da stdin a stdout. -t consente di indicare un numero di riga;
// se -t non e' specificata la selezione termina alla riga contenente il
// secondo nonoption item sulla command line; in assenza di questo la selezione
// termina a fine file

int valid_t(struct OPT *vld,int cnt)
{
    char *ptr;
    static int instance;
    extern unsigned options;
    extern unsigned long stopNum;

    if(instance++)
        err_handle(NULL,NULL);
    for(ptr = vld->arg; *ptr; ptr++)
        if(!isdigit(*ptr))
            err_handle(NULL,NULL);
    stopNum = atol(vld->arg);
    options |= TONUM;
}

// gsetione errore di sintassi: visualizzazione del messaggio di help e
// uscita con errore
```

```
int err_handle(struct OPT *vld,int cnt)
{
    extern char *helpStr;

    fprintf(stderr,"%s: Usage: %s [option(s)] string1 [string2]\n%s",PRG,PRG,
                                                                    helpStr);
    exit(RET_ERR);                                          // non togliere!!
}

// controllo dei parametri non-opzione presenti sulla command line: possono
// essere nessuno, uno o due a seconda delle opzioni richieste

int ctl_strings(struct OPT *vld,int cnt)
{
    static int instance;

    switch(instance) {
        case 0:
            break;                                  // string1 sempre ammessa
        case 1:
            if(!(options & FROMLIN))
                err_handle(NULL,NULL);      // string2 ammessa solo se c'e' -fl
            break;
        default:
            err_handle(NULL,NULL);                          // troppi parametri
    }
    return(++instance);
}

#pragma warn .par
#pragma warn .rvl
#pragma warn .aus

// array di strutture che associano ad ogni opzione la corrispondente
// funzione di validazione

struct VOPT valfuncs[] = {
    {'c',valid_c},
    {'f',valid_f},
    {'t',valid_t},
    {ERRCHAR,err_handle},
    {NULL,ctl_strings},
};

// stringa elencante le opzioni ammesse. Quelle seguite da ':' richiedono un
// argomento

char *optionS = "c:f:t:";

// main() scandisce la command line via parseopt() e, se le opzioni sono
// regolari, procede nell'elaborazione via search()

int main(int argc,char **argv)
{
    struct OPT *opt;

    fprintf(stderr,"%s %s - line selection from stdin to stdout - Barninga Z! '%s\n",
                                                                PRG,VER,YEAR);
    if(argc == 1)
        err_handle(NULL,NULL);                              // non ritorna!
    if(!(opt = parseopt(argc,argv,optionS,SWCHAR,ILLMSG,ILLMSG,valfuncs))) {
        perror(PRG);
        return(RET_ERR);
    }
```

```
    if(!opt[0].val)
        err_handle(NULL,NULL);                              // non c'e' string1
    return(search(argv[argc-opt[0].val],argv[argc-opt[0].val+1]));
}

// search() legge stdin riga per riga e, secondo le opzioni e le stringhe
// specificate sulla riga di comando seleziona le righe da visualizzare su
// stdout

int search(char *string1,char *string2)
{
    register lFlag = 0;
    char line[MAXLIN];
    unsigned long i;
    extern unsigned options;
    extern unsigned long startNum, stopNum;

// dapprima sono scartate tutte le righe che precedono la prima da selezionare
// (se non e' stato specificata -fnum allora startNum e' 0 e non e' scartata
// alcuna riga)

    for(i = 1L; i < startNum; i++)
        fgets(line,MAXLIN,stdin);

// se la selezione deve iniziare alla riga contenente string1, questa e'
// cercata nel file e si comincia a copiare righe su stdout non appena essa
// e' trovata

    if(options & FROMLIN) {
        for(; fgets(line,MAXLIN,stdin); i++)
            if(strstr(line,string1)) {
                lFlag = printf(line);
                break;
            }

// la selezione di righe prosegue fino a che non e' incontrata la riga finale
// definita da stopNum (-t) o segnalata dalla presenza di string2

        for(; fgets(line,MAXLIN,stdin) && (i < stopNum); i++) {
            printf(line);
            if(string2)
                if(strstr(line,string2))
                    break;
        }
    }

// se la selezione deve partire dalla prima riga allora si copiano su stdout
// tutte le righe sino a quella contenente string1, che in questo caso
// permette di selezionare la riga di stop. Se era stato specificata una riga
// di stop con -t, allora e' controllata stopNum (che altrimenti contiene il
// massimo valore esprimible con un long, rendendo ininfluente il test)

    else {
        for(; fgets(line,MAXLIN,stdin) && (i < stopNum); i++) {
            lFlag = printf(line);
            if(strstr(line,string1))
                break;
        }
    }
    return(lFlag ? 1 : 0);       // 1 = copiate righe; 0 = nessuna riga copiata
}
```

Ecco la nuova versione del nostro file batch:

```
:ATTESA
TIMEGONE 230000
IF ERRORLEVEL 1 GOTO ADESSO
GOTO ATTESA
:ADESSO
ECHO LE ORE 23:00 SONO APPENA TRASCORSE
DIR "D:\PROC\OUTPUT.DAT" | FIND " 0 " | EMPTYLVL
IF ERRORLEVEL 1 GOTO NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE 0 BYTES: UTILIZZO FCREATE...
DATECMD FCREATE 0 C:\PROC\OUT\@a@M@G.OUT
GOTO END
:NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE MAGGIORE DI 0 BYTES: UTILIZZO COPY...
DATECMD COPY D:\PROC\OUTPUT.DAT C:\PROC\OUT\@a@M@G.OUT
ECHO GENERAZIONE DEL FILE DI RIEPILOGO...
SELSTR -cb "* RIEPILOGO *" "* FINE *" < D:\PROC\OUTPUT.DAT > C:\PROC\RIEPILOG.DAT
IF ERRORLEVEL 255 GOTO SELERROR
IF ERRORLEVEL 1 GOTO END
ECHO IL FILE OUTPUT.DAT NON CONTIENE LA SEZIONE DI RIEPILOGO
GOTO END
:SELERROR
ECHO ERRORE NELLA GENERAZIONE DEL RIEPILOGO
:END
```

La sezione estratta da `OUTPUT.DAT`, presente sul disco remoto, è scritta nel file `RIEPILOG.DAT`, sul disco locale. Dal momento che `SELSTR` restituisce in `ERRORLEVEL` un codice indicativo dello stato di uscita, è possibile effettuare in modo semplice la gestione degli errori: come si vede, in caso di errore di elaborazione viene restituito `255`: il flusso del file batch salta all'etichetta `SELERROR` ed è visualizzato un opportuno messaggio. Se `ERRORLEVEL` vale `1`, significa che l'elaborazione è terminata regolarmente ed è stata estratta almeno una riga di testo; altrimenti `ERRORLEVEL` contiene necessariamente `0`, che indica terminazione regolare, ma senza estrazione di testo: non rimane che segnalare che il contenuto di `OUTPUT.DAT` non è quello atteso.

Estrarre colonne di testo da un file: CUT

A questo punto vi è la necessità (ebbene sì) di postelaborare `RIEPILOG.DAT`: a scopo di esempio, ipotizziamo che tra riga iniziale e finale ne sia presente un numero variabile, tutte aventi il formato descritto di seguito:

```
+-CODICE-_+-DATA-_+-FILE-_+-IMPORTO-_
```

La sezione estratta, priva di riga inizale e finale, ha pertanto un aspetto analogo a quello delle righe seguenti:

```
099355476719940722OPER000100120344720
098446273319940722OPER003400009873325
088836288219940831OPER102800014436255
....
094553728219940912OPER001700225467520
062253725519941004OPER013100067855490
084463282919941103OPER000700127377385
```

E' necessario generare una tabella, da scrivere in un nuovo file, in cui solamente i campi `DATA`, `IMPORTO` e `CODICE` siano riportati in modo leggibile e nell'ordine indicato. E' inoltre necessario generare un secondo file, ogni riga del quale contenga esclusivamente il campo `FILE`, a scopo di elaborazione successiva.

Strumento atto allo scopo è la utility `CUT`, ispirata all'omonimo comando Unix, del quale implementa le caratteristiche principali (in sostanza, la capacità di estrarre colonne di testo da un file), affiancandovi alcune novità.

`CUT` legge lo standard input, da ogni riga del quale estrae i caratteri occupanti le posizioni specificate mediante l'opzione della command line `-c` e li scrive sullo standard output. Tutti i messaggi sono scritti sullo standard error, al fine di non influenzare l'eventuale redirezione dell'output prodotto.

E' possibile specificare più istanze dell'opzione `-c` per richiedere l'estrazione di più colonne di testo: queste possono sovrapporsi parzialmente o totalmente; inoltre la posizione di partenza può essere superiore a quella di arrivo (in questo caso `CUT` estrae i caratteri da destra a sinistra). Vediamo un esempio: nell'ipotesi che lo standard input di `CUT` sia costituito dalla riga

```
ABCDE12345FGHIJ67890
```

il comando

```
cut -c1-3 -c2-5 -c8-5 -c15-25
```

scrive sullo standard output la riga

```
ABCBCDE321EJ67890
```

Si noti che le posizioni dei caratteri sono numerate a partire da 1. Inoltre, per default, laddove la lunghezza della riga di standard input non sia sufficiente a soddisfare interamente uno degli intervalli specificati (come nel caso `-c15-25`), vengono comunque estratti almeno i caratteri presenti: sono disponibili, però, tre opzioni di command line con le quali è possibile gestire tali situazioni in modo differente. In particolare, `-d` forza `CUT` a scartare la riga di input, mentre `-x` determina l'interruzione dell'elaborazione con `ERRORLEVEL` pari a `255` (il valore di default per una terminazione regolare è `0`). Ancora, l'opzione `-p` consente di specificare una stringa dalla quale estrarre i caratteri necessari a compensare quelli mancanti nello standard input: il comando

```
cut -p#=@ -c15-25
```

estrae dallo standard input dell'esempio precedente la riga

```
J67890#=@#=
```

Alcuni caratteri della stringa "`#=@`" sono ripetuti, in quanto la sua lunghezza non è sufficiente; al contrario, i caratteri eventualmente eccedenti la lunghezza necessaria sono ignorati.

`CUT` riconosce inoltre alcune opzioni che consentono di inserire stringhe in testa (`-b`) e in coda (`-e`) alle righe di standard output, nonché tra gli intervalli di caratteri (`-m`). Vediamone un esempio di utilizzo nel solito file batch:

```
:ATTESA
TIMEGONE 230000
IF ERRORLEVEL 1 GOTO ADESSO
GOTO ATTESA
:ADESSO
ECHO LE ORE 23:00 SONO APPENA TRASCORSE
DIR "D:\PROC\OUTPUT.DAT" | FIND " 0 " | EMPTYLVL
IF ERRORLEVEL 1 GOTO NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE 0 BYTES: UTILIZZO FCREATE...
DATECMD FCREATE 0 C:\PROC\OUT\@a@M@G.OUT
GOTO END
:NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE MAGGIORE DI 0 BYTES: UTILIZZO COPY...
```

```
DATECMD COPY D:\PROC\OUTPUT.DAT C:\PROC\OUT\@a@M@G.OUT
ECHO GENERAZIONE DEL FILE DI RIEPILOGO...
SELSTR -cb "* RIEPILOGO *" "* FINE *" < D:\PROC\OUTPUT.DAT > C:\PROC\RIEPILOG.DAT
IF ERRORLEVEL 255 GOTO SELERROR
IF ERRORLEVEL 1 GOTO POSTELAB
ECHO IL FILE OUTPUT.DAT NON CONTIENE LA SEZIONE DI RIEPILOGO
GOTO END
:SELERROR
ECHO ERRORE NELLA GENERAZIONE DEL RIEPILOGO
:POSTELAB
ECHO GENERAZIONE DELLA TABELLA DI RIEPILOGO...
TYPE C:\PROC\RIEPILOG.DAT FIND /V "* RIEPIL" | FIND /V "* FINE *" > C:\PROC\TR.TMP
ECHO TABELLA DI RIEPILOGO > C:\PROC\TR.TXT
ECHO. >> C:\PROC\TR.TXT
ECHO    DATA       IMPORTO       CODICE >> C:\PROC\TR.TXT
ECHO +-------------------------------------+ >> C:\PROC\TR.TXT
CUT -b"_ " -m" _ " -e" _" -c11-18 -c27-37 -c1-10 < C:\PROC\TR.TMP >> C:\PROC\TR.TXT
ECHO +-------------------------------------+ >> C:\PROC\TR.TMP
ECHO GENERAZIONE DELLA LISTA DI FILES...
CUT -c19-26 < C:\PROC\TR.TMP > C:\PROC\FILES.DAT
DEL C:\PROC\TR.TMP
:END
```

Il comando FIND è utilizzato con l'opzione /V per generare un file temporaneo (TR.TMP) contenente tutte le righe di RIEPILOG.DAT, eccetto la prima e l'ultima (intestazione e chiusura); TR.TMP è poi elaborato con CUT per produrre la tabella desiderata (TR.TXT) e il file contenente le sole occorenze del campo FILES.

Il contenuto del file TR.TXT è il seguente:

TABELLA DI RIEPILOGO

DATA	IMPORTO	CODICE
19940722	00120344720	0993554767
19940722	00009873325	0984462733
19940831	00014436255	0888362882
....		
19940912	00225467520	0945537282
19941004	00067855490	0622537255
19941103	00127377385	0844632829

Il contenuto di FILES.DAT è invece:

```
OPER0001
OPER0034
OPER1028
....
OPER0017
OPER0131
OPER0007
```

Segue il listato commentato di CUT (vedere pag. 479 e seguenti circa l'implementazione di PARSEOPT.OBJ).

```
/****************************************************************************

    CUT.C - Barninga Z! - 1994

    Utility che riprende ed amplia le funzionalita' del comando CUT di Unix.
    Circa la sintassi vedere la variabile helpStr.
```

```
    Compilato sotto Borland C++ 3.1

    bcc -O -d -rd -k- cut.c parseopt.obj

******************************************************************************/
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <stdlib.h>

#include "parseopt.h"

#define  PRG         "CUT"         // nome del programma
#define  REL         "1.0"         // versione
#define  YEAR        "94"          // anno di rilascio
#define  ERRCOD      255           // codice errore uscita programma
#define  MAXLIN      2048          // massima lunghezza di una riga
#define  MAXPAD      128           // max lungh. stringa di padding
#define  CUTLEN      4             // max 4 cifre per opz. -c
#define  CUTOPT      1             // -c
#define  EXITOPT     2             // -x
#define  DISCRDOPT   4             // -d
#define  PADOPT      8             // -p
#define  MIDOPT      16            // -s
#define  BEGOPT      32            // -b
#define  ENDOPT      64            // -e
#define  SWITCH      '-'           // switch character per opzioni

// CUTTER e' il template di struttura che controlla la formazione delle
// colonne di testo: start contiene l'offset del primo byte; stop quello
// dell'ultimo. In pratica sono gli estremi del range da copiare. start
// puo' essere maggiore di stop, nel qual caso i bytes sono copiati in
// ordine inverso

typedef struct {
    unsigned start;
    unsigned stop;
} CUTTER;

char *helpStr = "\
Syntax is: cut option(s)\n\
Default exit code: 0. Input: stdin; Output: stdout; Msgs: stderr. Options are:\n\
-bBEGSTR      BEGSTR is the string to be inserted at the beginning of each\n\
              output line.\n\
-cSTART-STOP  START-STOP specify a range of characters to be copied from each\n\
              input line to output. If START is greater than STOP, characters\n\
              in the range are reversed. More than one range can be given;\n\
              anyway, at least one range is required. See also -d -p -x.\n\
-d            If one or more -c ranges fall outside an input line, the line\n\
              itself will be discarded. By default, columns are output even\n\
              if one or more ranges are not full. -d -p -x are mutually\n\
              exclusive.\n\
-eENDSTR      ENDSTR is the string to be appended at the end of each output\n\
              line.\n\
-mMIDSTR      MIDSTR is the string to be inserted between two -c ranges; useful\n\
              only if more than one range is given.\n\
-pPADSTR      PADSTR is the string used to pad output columns of -c ranges\n\
              falling outside a stdin line. When -p is not requested, output\n\
              lines can have different length. -p -d -x are mutually exclusive.\n\
-x            If one or more -c ranges fall outside an input line, the program\n\
              will immediately return 255. -x -d -p are mutually exclusive.\n\
-?            Displays this help screen and exits with a return code of 255.\
```

```
";

char *crgtMsg = "%s %s: cuts stdin into columns. Barninga Z! '%s. Help: -?\n";
char *eLinEnd = "Cut pointers past end of line";
char *eOptCut = "No cut option specified";
char *eOptFmt = "Invalid option format";
char *eOptGen = "Invalid option or option format";
char *eOptSeq = "Invalid option sequence";

CUTTER *cutArr;               // struttura di controllo
char padStr[MAXPAD];          // stringa per il padding
char begStr[MAXPAD];          // stringa di apertura
char midStr[MAXPAD];          // stringa di fincatura
char endStr[MAXPAD];          // stringa di chiusura

int valid_b(struct OPT *vld,int cnt);
int valid_c(struct OPT *vld,int cnt);
int valid_d(struct OPT *vld,int cnt);
int valid_e(struct OPT *vld,int cnt);
int valid_m(struct OPT *vld,int cnt);
int valid_p(struct OPT *vld,int cnt);
int valid_x(struct OPT *vld,int cnt);
int hlp_handle(struct OPT *vld,int cnt);
int err_handle(struct OPT *vld,int cnt);

int main(int argc,char **argv);
void cutstream(char *line,char *buffer,char *bufprint,int maxlen);

#pragma  warn -par
#pragma  warn -rvl

// se specificata opzione -b la stringa argomento deve essere copiata nello
// spazio apposito (begStr). Questa stringa e' scritta in testa ad ogni riga
// di standard output prodotta dal programma.

int valid_b(struct OPT *vld,int cnt)
{
    static int instance;
    extern char *eOptSeq;
    extern unsigned options;
    extern char begStr[];

    if(instance++)
        err_handle(NULL,(int)eOptSeq);
    strcpy(begStr,vld->arg);
    return(options |= BEGOPT);
}

// almeno una opzione -c deve sempre essere specificata. Qui vengono effettuati
// tutti i controlli di liceita' sull'argomento (estremi del range di colonne
// da estrarre da stdin) e per ogni range e' allocata una struttura CUTTER,
// formando cosi' un array di strutture che descrivono come deve essere
// "ritagliato" stdin.

int valid_c(struct OPT *vld,int cnt)
{
    register i, flag;
    static int instance;
    extern char *eOptFmt;
    extern unsigned options;
    extern CUTTER *cutArr;

    if(!isdigit(*(vld->arg)))
        err_handle(NULL,(int)eOptFmt);    // errore: primo car. non e' numerico
```

```
    for(i = 1, flag = 0; vld->arg[i]; i++)
        if(vld->arg[i] == '-') {             // un - puo' (deve) esserci
            if(flag)
                err_handle(NULL,(int)eOptFmt);  // errore: gia' trovato un -
            if(i > CUTLEN)
                err_handle(NULL,(int)eOptFmt);  // errore: numero troppo alto
            flag = i+1;                         // indice inizio secondo numero
        }
        else
            if(!isdigit(vld->arg[i]))
                err_handle(NULL,(int)eOptFmt); // errore: non e' - o una cifra
    if(!flag)
        err_handle(NULL,(int)eOptFmt);          // errore: non c'e' il -
    if((i-flag) > CUTLEN)
        err_handle(NULL,(int)eOptFmt);          // errore: numero troppo alto
    if(!cutArr)                                           // solo prima volta!!
        if(!(cutArr = (CUTTER *)malloc(sizeof(CUTTER))))
            err_handle(NULL,(int)sys_errlist[errno]);
    sscanf(vld->arg,"%d-%d",&(cutArr[instance].start),&(cutArr[instance].stop));
    if(!cutArr[instance].start || !cutArr[instance].stop)
        err_handle(NULL,(int)eOptFmt);
    ++instance;
    if(!(cutArr = (CUTTER *)realloc(cutArr,(instance+1)*sizeof(CUTTER))))
        err_handle(NULL,(int)sys_errlist[errno]);
    cutArr[instance].start = cutArr[instance].stop = NULL;
    return(options |= 1);
}

// l'opzione -d richiede che se uno o entrambi gli estermi di un range (-c)
// "cadono" al di fuori della riga attualmente processata la riga stessa
// deve essere scartata. L'opzione -d e' alternativa a -p e -x.

int valid_d(struct OPT *vld,int cnt)
{
    static int instance;
    extern char *eOptSeq;
    extern unsigned options;

    if(instance++ || (options & PADOPT+EXITOPT))
        err_handle(NULL,(int)eOptSeq);
    return(options |= DISCRDOPT);
}

// se specificata opzione -e la stringa argomento deve essere copiata nello
// spazio apposito (endStr). Questa stringa e' scritta in coda ad ogni riga
// di standard output prodotta dal programma.

int valid_e(struct OPT *vld,int cnt)
{
    static int instance;
    extern char *eOptSeq;
    extern unsigned options;
    extern char endStr[];

    if(instance++)
        err_handle(NULL,(int)eOptSeq);
    strcpy(endStr,vld->arg);
    return(options |= ENDOPT);
}

// se specificata opzione -m la stringa argomento deve essere copiata nello
// spazio apposito (midStr). Questa stringa e' scritta, in ogni riga di
// standard output prodotta dal programma, tra due range di caratteri estratti
// (-c) da stdin, se almeno 2 istanze di -c sono state specificate. Se ne e'
```

```
// stata richiesta una sola, l'opzione -m e' ignorata.

int valid_m(struct OPT *vld,int cnt)
{
    static int instance;
    extern char *eOptSeq;
    extern unsigned options;
    extern char midStr[];

    if(instance++)
        err_handle(NULL,(int)eOptSeq);
    strcpy(midStr,vld->arg);
    return(options |= MIDOPT);
}

// se specificata opzione -p la stringa argomento deve essere copiata nello
// spazio apposito (padStr). Questa stringa e' utilizzata, in ogni riga di
// standard output prodotta dal programma, per riempire il range richiesto
// (-c), se il numero di caratteri estratti da stdin e' minore dell'ampiezza
// specificata per il range stesso. E' significativa solo se uno o entrambi
// gli estremi di un range "cadono" al di fuori della riga di stdin attualmente
// processata. E' alternativa a -d e -x.

int valid_p(struct OPT *vld,int cnt)
{
    static int instance;
    extern char *eOptSeq;
    extern unsigned options;
    extern char padStr[];

    if(instance++ || (options & DISCRDOPT+EXITOPT))
        err_handle(NULL,(int)eOptSeq);
    strcpy(padStr,vld->arg);
    return(options |= PADOPT);
}

// l'opzione -x richiede che se uno o entrambi gli estermi di un range (-c)
// "cadono" al di fuori della riga attualmente processata l'elaborazione deve
// essere interrotta e restituito un codice di errore. Alternativa a -p e -x.

int valid_x(struct OPT *vld,int cnt)
{
    static int instance;
    extern char *eOptSeq;
    extern unsigned options;

    if(instance++ || (options & DISCRDOPT+PADOPT))
        err_handle(NULL,(int)eOptSeq);
    return(options |= EXITOPT);
}

// visualizza la videata di help se e' specificata opzione -?

int hlp_handle(struct OPT *vld,int cnt)
{
    extern char *helpStr;

    err_handle(NULL,(int)helpStr);
}

// gestisce tutte le situazioni in cui e' necessario visualizzare un messaggio
// ed uscire con un errore. Se il parametro vld e' NULL significa che la
// chiamata e' fatta esplicitamente dal programmatore e quindi e' visualizzato
// il messaggio il cui indirizzo e' passato come secondo parametro (con
```

```
// opportuno cast!). Se vld non e' NULL, allora err_handle() e' chiamata da
// parseopt() e quindi e' visualizzato un messaggio generico.

int err_handle(struct OPT *vld,int cnt)
{
    extern char *eOptGen;

    fprintf(stderr,"%s: %s\n",PRG,vld ? eOptGen : (char *)cnt);
    exit(ERRCOD);
}

#pragma  warn .par
#pragma  warn .rvl

// array che associa le opzioni alla funzione di validazione corrispondente

struct VOPT vfuncs[] = {
    {'b',valid_b},
    {'c',valid_c},
    {'d',valid_d},
    {'e',valid_e},
    {'m',valid_m},
    {'p',valid_p},
    {'x',valid_x},
    {'?',hlp_handle},
    {ERRCHAR,err_handle},
    {NULL,NULL}
};

// stringa di elenco delle opzioni; se una lettera e' seguita da ':' significa
// che l'opzione richiede un argomento

char *optionStr = "b:c:de:m:p:x?";

unsigned options;                                         // flags delle opzioni

// main() valida le opzioni via parseopt() e gestisce il ciclo di estrazione
// dei range per ogni riga di input. Tutti i messaggi del programma, incluso
// il copyright, sono scritti su stderr; in tal modo la redirezione
// dell'output prodotto non li include.

int main(int argc,char **argv)
{
    register int maxlen;
    register char *inibuf;
    char line[MAXLIN], buffer[MAXLIN];
    extern char *crgtMsg;
    extern char *eOptCut;
    extern unsigned options;
    extern char *optionStr;

    fprintf(stderr,crgtMsg,PRG,REL,YEAR);
    if(!parseopt(argc,argv,optionStr,SWITCH,NULL,NULL,vfuncs))
        err_handle(NULL,(int)sys_errlist[errno]);   // sys_errlist: errori std
    if(!(options & CUTOPT))
        err_handle(NULL,(int)eOptCut);
    inibuf = buffer;
    maxlen = MAXLIN;

// se specificata -b, begStr puo' essere copiata nel buffer fuori dal ciclo
// perche' comunque e' sempre la stessa per ogni riga

    if(options & BEGOPT) {
        strcpy(buffer,begStr);
```

```
        inibuf += strlen(begStr);    // nuovo indirizzo a cui copiare i range
        maxlen -= strlen(begStr);    // spazio residuo nel buffer
    }
    while(gets(line))
        cutstream(line,inibuf,buffer,maxlen);
    return(0);
}

// cutstream() estrae i range dalla riga di testo ricevuta come parametro e
// scrive la riga output su stdout. Si noti che il paramtero bufprint e'
// l'indirizzo originale del buffer: questo infatti deve essere stampato
// tutto anche se cutstream() ne lavora solo una parte (se c'e' begStr). E'
// evidente che in assenza di opzione -b, buffer e bufprint contengono lo
// stesso indirizzo.

void cutstream(char *line,char *buffer,char *bufprint,int maxlen)
{
    register i, j, k;
    int len, inc, start, stop, lim;
    extern char *eLinEnd;
    extern unsigned options;
    extern CUTTER *cutArr;
    extern char padStr[], midStr[], endStr[];

    j = 0;
    len = strlen(line);

// ciclo di estrazione di range, iterato su ogni riga per tutti gli elementi
// dell'array di strutture CUTTER: ciascuna, infatti, descrive un range.

    for(i = 0; cutArr[i].start; ) {

// calcolo dell'incremento di scansione della riga: se start e' < di stop si
// procede da sx a dx e percio' l'incremento e' +1; se start > stop allora si
// procede da dx a sx (a ritroso): l'incremento e' -1.

        inc = (start = cutArr[i].start) > (stop = cutArr[i].stop) ? -1 : 1;
        lim = 0;                                       // usato per il padding

// se uno degli estremi del range cade fuori dalla riga allora bisogna valutare
// che fare a seconda delle opzioni richieste

        if(start > len || stop > len)
            if(options & DISCRDOPT)
                return;                                // -d: scartare la riga
            else
                if(options & EXITOPT)
                    err_handle(NULL,(int)eLinEnd);     // -x: fine programma
                else {

// se la riga deve essere comunque processata allora si controlla se entrambi
// gli estremi sono fuori dal range: in questo caso i caratteri di padding
// sono pari alla differenza tra gli estremi piu' uno.

                    if(start > len && stop > len) {
                        len = 0;
                        if(options & PADOPT)
                            lim = 1+(stop-start)*inc;
                    }

// se uno solo degli estremi e' fuori dalla riga, allora una parte del range
// deriva da caratteri della riga, la cui fine diviene uno dei due estremi del
// range stesso (start o stop, a seconda della direzione). Il numero di
// caratteri di padding e' pari al numero di caratteri non estraibili dalla
```

```
// riga, cioe' alla parte "scoperta" del range.

                    else {
                        if(options & PADOPT)
                            lim = max(start,stop)-len;
                        start = min(start,len);
                        stop = min(stop,len);
                    }
                }
        if(len)                       // si copia solo se la riga non e' vuota!

// il ciclo gestisce l'operazione di copia; l'applicazione a start di un
// incremento positivo o negativo a seconda dei casi permette di gestire
// entrambe le situazioni (da sx a dx o viceversa)

            for(--start, --stop; j < maxlen-1; start += inc) {
                buffer[j++] = line[start];
                if(start == stop)
                    break;                                    // fine del range
            }

// se e' richiesta -p si copia la stringa di padding nel buffer per il numero
// di caratteri ancora mancanti nel range. Se la stringa e' piu' lunga del
// necessario viene troncata; se e' piu' corta viene ripetuta sino a riempire
// tutto lo spazio.

        if(options & PADOPT)
            for(k = 0; j < maxlen-1 && lim; lim--) {
                buffer[j++] = padStr[k++];
                if(!padStr[k])
                    k = 0;
            }
        ++i;

// se e' specificata -m e ci sono ancora range da estrarre (l'ultimo elemento
// dell'array di struct CUTTER contiene NULL in entrambe i campi per segnalare
// la fine) allora si concatena al buffer la stringa midStr. Non si usa
// strcat() per essere sicuri di non andare oltre lo spazio rimanente nel
// buffer.

        if((options & MIDOPT) && cutArr[i].start)
            for(k = 0; j < maxlen-1 && midStr[k]; )
                buffer[j++] = midStr[k++];
        else

// se invece quello appena estratto e' l'ultimo range ed e' specificata -e
// si concatena al buffer la endStr.

            if((options & ENDOPT) && !cutArr[i].start)
                for(k = 0; j < maxlen-1 && endStr[k]; )
                    buffer[j++] = endStr[k++];
    }
    buffer[j] = NULL;                                                // tappo!
    puts(bufprint);
}
```

Lanciando `CUT` con l'opzione `-?` sulla riga di comando viene visualizzato un testo di aiuto, che ne riassume le principali caratteristiche ed opzioni.

Il comando FOR rivisitato: DOLIST

Veniamo a `FILES.DAT` (generato da `CUT`), ogni riga del quale, ai fini del nostro esempio, rappresenta un nome di file: ipotizziamo che su ciascuno di essi sia necessario ripetere la medesima elaborazione. L'interprete DOS implementa il comando `FOR`, che non è in grado di accettare una lista variabile di parametri (vedere pag. 565): risulta quindi impossibile utilizzare `FILES.DAT` allo scopo. La utility `DOLIST` offre un rimedio: essa esegue una riga di comando, che viene considerata suddivisa in due parti, di cui la seconda opzionale, inserendo tra di esse la riga letta dallo standard input. Se da questo possono essere lette più righe, il comando viene iterato per ciascuna di esse.

La complessità del meccanismo è solo apparente: nell'ipotesi che l'elaborazione del nostro esempio consista nell'eseguire un secondo file batch il cui parametro sia ogni volta un diverso file, la procedura diviene la seguente:

```
:ATTESA
TIMEGONE 230000
IF ERRORLEVEL 1 GOTO ADESSO
GOTO ATTESA
:ADESSO
ECHO LE ORE 23:00 SONO APPENA TRASCORSE
DIR "D:\PROC\OUTPUT.DAT" | FIND " 0 " | EMPTYLVL
IF ERRORLEVEL 1 GOTO NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE 0 BYTES: UTILIZZO FCREATE...
DATECMD FCREATE 0 C:\PROC\OUT\@a@M@G.OUT
GOTO END
:NONZERO
ECHO IL FILE OUTPUT.DAT HA DIMENSIONE MAGGIORE DI 0 BYTES: UTILIZZO COPY...
DATECMD COPY D:\PROC\OUTPUT.DAT C:\PROC\OUT\@a@M@G.OUT
ECHO GENERAZIONE DEL FILE DI RIEPILOGO...
SELSTR -cb "* RIEPILOGO *" "* FINE *" < D:\PROC\OUTPUT.DAT > C:\PROC\RIEPILOG.DAT
IF ERRORLEVEL 255 GOTO SELERROR
IF ERRORLEVEL 1 GOTO POSTELAB
ECHO IL FILE OUTPUT.DAT NON CONTIENE LA SEZIONE DI RIEPILOGO
GOTO END
:SELERROR
ECHO ERRORE NELLA GENERAZIONE DEL RIEPILOGO
:POSTELAB
ECHO GENERAZIONE DELLA TABELLA DI RIEPILOGO...
TYPE C:\PROC\RIEPILOG.DAT FIND /V "* RIEPIL" | FIND /V "* FINE *" > C:\PROC\TR.TMP
ECHO TABELLA DI RIEPILOGO > C:\PROC\TR.TXT
ECHO. >> C:\PROC\TR.TXT
ECHO    DATA       IMPORTO       CODICE >> C:\PROC\TR.TXT
ECHO +------------------------------------+ >> C:\PROC\TR.TXT
CUT -b"_ " -m" _ " -e" _" -c11-18 -c27-37 -c1-10 < C:\PROC\TR.TMP >> C:\PROC\TR.TXT
ECHO +------------------------------------+ >> C:\PROC\TR.TMP
ECHO GENERAZIONE DELLA LISTA DI FILES...
CUT -c19-26 < C:\PROC\TR.TMP > C:\PROC\FILES.DAT
DEL C:\PROC\TR.TMP
ECHO ELABORAZIONE DELLA LISTA DI FILES IN FILES.DAT...
DOLIST "CALL ELABFILE.BAT" < FILES.LST
IF ERRORLEVEL GOTO FILERROR
GOTO END
:FILERROR
ECHO ERRORE NELLA ELABORAZIONE DELLA LISTA DI FILES
:END
```

`DOLIST` esegue la riga di comando una volta per ogni riga presente nel file `FILES.DAT`: con i dati dell'esempio, sono generati e lanciati i seguenti comandi:

```
CALL ELABFILE.BAT OPER0001
CALL ELABFILE.BAT OPER0034
```

```
CALL ELABFILE.BAT OPER1028
....
CALL ELABFILE.BAT OPER0017
CALL ELABFILE.BAT OPER0131
CALL ELABFILE.BAT OPER0007
```

Si noti l'utilizzo delle virgolette: esse sono necessarie in quanto CALL ed ELABFILE.BAT devono costituire insieme la prima parte del comando. Senza le virgolette DOLIST interpreterebbe CALL come prima porzione della command line e ELABFILE.BAT come seconda, inserendo tra esse la riga letta da FILES.DAT. I comandi eseguiti sarebbero perciò

```
CALL OPER0001 ELABFILE.BAT
CALL OPER0034 ELABFILE.BAT
CALL OPER1028 ELABFILE.BAT
....
CALL OPER0017 ELABFILE.BAT
CALL OPER0131 ELABFILE.BAT
CALL OPER0007 ELABFILE.BAT
```

con scarse possibilità di successo.

Segue il listato di DOLIST, ampiamente commentato.

```
/******************************************************************************

    DOLIST.C - Barninga_Z! - 17/10/93

    Esegue un comando dos su una lista di argomenti. Esempio

    dolist copy c:\backups\*.bak < files.lst

    copia tutti i files elencati nel file files.lst nella dir c:\backups
    attribuendo a ciascuno estensione .bak. Il files contenente gli
    argomenti deve essere in formato ascii, una riga per ogni comando.
    In pratica dolist compone ed esegue il comando:

    comando riga_della_lista comando_2a_parte

    In uscita ERRORLEVEL e' settato come segue:

    0 = comando eseguito (indipenentemente dal suo risultato)
    1 = elaborazione interrotta (errore di sintassi della cmdline di DOLIST)
    2 = comando non eseguito (system fallita)
    3 = comando non eseguito (command line da eseguire > 127 caratteri)

    Compilato sotto Borland C++ 3.1:

    bcc -d -rd -k- dolist.c

******************************************************************************/
#pragma  warn -pia

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define  PRG         "DOLIST"
#define  VER         "1.0"
#define  YEAR        "93"
#define  SEP_STR     " "
#define  NUL_CHRS    " \t\n\r"
#define  MAXCMD      128
```

```
#define  MAXLIN      256

// main() gestisce tutte le operazioni necessarie

int main(int argc,char **argv)
{
    register len, len2 = 0, retCode = 0;
    char *ptr;
    char cmd[MAXCMD], line[MAXLIN];

    printf("%s %s: esegue comando su lista argomenti - Barninga Z! '%s\n",PRG,
                                                                   VER,YEAR);
    if((argc < 2) || (argc > 3)) {
        printf("%s: sintassi: comando [comando_2a_parte] < lista_args\n",PRG);
        retCode = 1;
    }
    else {

// comincia la costruzione della command line copiando la prima parte del
// comando nel buffer appositamente predisposto

        strcpy(cmd,argv[1]);
        strcat(cmd,SEP_STR);
        len = strlen(cmd);
        if(argc == 3)
            len2 = strlen(argv[2])+strlen(SEP_STR);

// ciclo di elaborazione: per ogni riga del file specificato come stdin viene
// costruita una command line concatenando alla prima parte del comando
// (argv[1]) la riga letta dal file e la seconda parte del comando (argv[2])
// e viene eseguito il comando risultante via system()

        while(gets(line)) {
            if(ptr = strtok(line,NUL_CHRS)) {
                cmd[len] = NULL;
                if((len+len2+strlen(ptr)) < MAXCMD) {
                    strcat(cmd,ptr);
                    if(len2) {
                        strcat(cmd,SEP_STR);
                        strcat(cmd,argv[2]);
                    }
                    if(!system(cmd))                              // esecuzione!
                        printf("%s: eseguito: %s\n",PRG,cmd);
                    else {
                        printf("%s: fallito:  %s\n",PRG,cmd);
                        retCode = 2;
                    }
                }
                else {
                    printf("%s: ignorato: %s%s %s\n",PRG,cmd,ptr,argv[2]);
                    retCode = 3;
                }
            }
        }
    }
    return(retCode);
}
```

DOLIST restituisce in ERRORLEVEL un valore determinato dallo stato dell'elaborazione: in particolare, 0 indica che tutti i comandi sono stati eseguiti; 1 evidenza un errore di sintassi nella riga di comando; 2 indica il fallimento della chiamata alla funzione di libreria system(), che invoca

l'interprete dei comandi DOS[454]; 3, infine, segnala che la lunghezza del comando composto mediante l'inserimento della riga di standard input tra prima e seconda parte della riga di comando risulta eccessiva[455].

I comandi nidificati: CMDSUBST

Che cosa accade in ELABFILE.BAT? Le ipotesi del nostro esempio si complicano sempre più (ma è l'ultimo sforzo, promesso!): la prima riga di ciascuno dei file elencati in FILES.DAT contiene un codice numerico, espresso in formato ASCII, che esprime uno stato relativo alle elaborazioni del riepilogo. La stringa 000 indica che quella particolare operazione è stata conclusa senza errori (e pertanto non vi sono altre righe nel file); tutti gli altri codici segnalano errori (il file contiene altre righe, descrittive dell'errore stesso). Scopo di POSTELAB.BAT è generare un file contenente le sole segnalazioni di errore.

Anche in questo caso il sistema Unix è fonte di ispirazione: la utility CMDSUBST consente di eseguire comandi DOS formati in tutto o in parte dall'output di altri comandi[456]. Vediamo subito un esempio: il comando

```
cmdsubst echo $type c:\autoexec.bat$
```

produce un output costituito dalla prima riga del file AUTOEXEC.BAT. Infatti, CMDSUBST scandisce la command line alla ricerca dei caratteri '$'[457] ed esegue, come un normale comando DOS, quanto tra essi compreso. La prima riga dello standard output prodotto dal comando è inserita nella command line originale, in luogo della stringa eseguita. Infine, è eseguita la command line risultante. Nell'esempio testè presentato, pertanto, viene dapprima lanciato il comando

```
type c:\autoexec.bat
```

Nell'ipotesi che la prima riga del file sia

```
@ECHO OFF
```

la command line risultante, eseguita da CMDSUBST, è

```
echo @ECHO OFF
```

la quale, a sua volta, produce lo standard output

---

[454] DOLIST esegue la command line effettuando una *DOS shell*, cioè invocando una seconda istanza (transiente) dell'interprete dei comandi. Ne segue che COMMAND.COM (o, comunque, l'interprete utilizzato) deve essere disponibile (nella directory corrente o in una di quelle elencate nella variabile d'ambiente PATH) e deve esserci memoria libera in quantità sufficiente per il caricamento dell'interprete stesso e per l'esecuzione del comando da parte di questo. Vedere pag. 129.

[455] Il DOS accetta comandi di lunghezza inferiore o pari a 128 caratteri, compreso il CR terminale.

[456] In realtà non esiste, in Unix, un comando analogo a CMDSUBST. Tuttavia la *shell* standard (l'equivalente dell'interprete dei comandi in ambiente DOS) scandisce la command line alla ricerca di stringhe racchiuse tra apici inversi (il carattere '`'), le esegue come veri e propri comandi e le sostutuisce, nella command line stessa, con lo standard output prodotto.

[457] CMDSUBST utilizza il carattere '$' in luogo dell'apice inverso per comodità: quest'ultimo non è presente sulla tastiera italiana.

```
@ECHO OFF
```

E' possibile specificare più subcomandi, ma non è possibile nidificarli: in altre parole, è valida una command line come:

```
cmdsubst echo $echo pippo$ $echo pluto$ $echo paperino$
```

che produce l'output

```
pippoplutopaperino
```

ma non lo è, o meglio, non è interpretata come forse ci si aspetterebbe, la seguente:

```
cmdsubst echo $echo $echo pippo$$
```

L'output prodotto è

```
echo is ON echo pippo$
```

Infatti, CMDSUBST individua il primo subcomando nella sequenza "$echo $", mentre la coppia "$$" è sostituita con un singolo carattere '$' (è questa, del resto, la sintassi che consente di specificare una command line contenente il '$' medesimo).

Torniamo al nostro caso pratico: il file ELABFILE.BAT può dunque essere strutturato come segue[458]:

```
@ECHO OFF
CMDSUBST IF @$TYPE D:\PROC\%1$@==@000@ GOTO END
IF ERRORLEVEL 1 GOTO ERRORE
SELSTR -f2 STRINGA_INESISTENTE < D:\PROC\%1 >> C:\PROC\ERRORI.DAT
GOTO END
:ERRORE
ECHO ERRORE NELL'ESECUZIONE DELL'ELABORAZIONE DI %1
:END
```

La sostituzione effettuata da CMDSUBST genera un test IF dalla sintassi lecita; SELTSR estrae dal file tutto il testo a partire dalla seconda riga (opzione -f), nell'ipotesi che la stringa "STRINGA_INESISTENTE" non compaia mai nei file elaborati. La variabile %1 è sostituita dal DOS con il parametro passato a ELABFILE.BAT dal batch chiamante (nell'esempio si tratta del nome del file da elaborare). Si noti, infine, che nel batch chiamante è opportuno inserire, prima della riga che invoca DOLIST, un comando di cancellazione del file C:\PROC\ERRORI.DAT, dal momento che SELSTR vi scrive lo standard output in modalità di *append* (il testo è aggiunto in coda al file, se esistente; in caso contrario questo è creato e "allungato" di volta in volta).

Si ha perciò:

```
...
ECHO ELABORAZIONE DELLA LISTA DI FILES IN FILES.DAT...
IF EXIST C:\PROC\ERRORI.DAT DEL C:\PROC\ERRORI.DAT
DOLIST "CALL ELABFILE.BAT" < FILES.LST
IF ERRORLEVEL 1 GOTO FILERROR
...
```

Il listato di CMDSUBST, ampiamente commentato, è riportato di seguito.

[458] Ancora una volta, CUT e EMPTYLVL potrebbero ottenere, da soli, un risultato analogo, ma con un algoritmo più complesso e meno flessibile.

```
/******************************************************************************

    CMDSUBST.C - Barninga Z! - 1994

    Esegue command line sostituendo la prima riga dello stdout di un altro
    comando compreso tra i caratteri $. Vedere stringa di help per i
    particolari.

    Compilato sotto Borland C++ 3.1

    bcc cmdsubst.c

******************************************************************************/
#pragma  warn -pia

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <alloc.h>
#include <dir.h>
#include <process.h>

#define  PRG            "CMDSUBST"
#define  VER            "1.0"
#define  YEAR           "'93"
#define  SUBST_FLAG     '$'
#define  MAXCMD         128
#define  TEMPLATE       "CSXXXXXX"
#define  ROOTDIR        "?:\\"
#define  OUTREDIR       ">"
#define  SEPSTR         " "
#define  EOL            '\n'

char *helpStr = "\
The command line is searched for '$' characters. If found, a string between two\n\
'$' is executed as a command line and the first line of its standard output\n\
replaces it in the CMDSUBST command line itself. Example: if the first line of\n\
the file c:\\autoexec.bat is\n\n\
@ECHO OFF\n\n\
the command line\n\n\
CMDSUBST echo $type c:\\autoexec.bat$\n\n\
causes the command\n\n\
echo @ECHO OFF\n\n\
to be actually executed. If a CMDSUBST command line contains a '$', it must be\n\
typed twice.\n\
";

// il template SUBCTL costituisce lo strumento per il controllo della
// sostituzione dei comandi nella command line. Il campo init referenzia
// il primo byte del comando, il campo len ne contiene la lunghezza. Viene
// allocato un array di strutture, ogni elemento del quale controlla la
// sostituzione di un comando.

typedef struct {
    char *init;
    int   len;
} SUBCTL;

int main(int argc,char **argv);
char *execSubCmd(char *subCmd,char *tempPath);
char *getCmdLine(int argc,char **argv);
char *getSubCmdOutput(char *outString,char *tempPath);
SUBCTL *initSubCtl(SUBCTL *ctl,int items,char *cmdBase);
char *makeCmdLine(SUBCTL *ctl,char *cmdLine,char *tempPath);
```

```
char *makeTempPath(char *tempPath);
SUBCTL *parseCmdLine(char *cmdLine);

// main() controlla l'esecuzione del programma in 4 fasi: ricostruzione
// della command line; creazione di un file temporaneo per la gestione
// dello stdout dei programmi; scansione della command line ed esecuzione
// dei subcomandi in essa contenuti; costruzione della nuova command line
// e sua esecuzione.

int main(int argc,char **argv)
{
    char *cmdLine;
    char tempPath[MAXPATH];
    SUBCTL *ctl;

    fprintf(stderr,"%s %s - Command line parms substitution - Barninga Z! %s\n",
                                                                  PRG,VER,YEAR);
    if(!(cmdLine = getCmdLine(argc,argv))) {
        fprintf(stderr,"%s: error: a command line must be supplied.\n%s",PRG,
                                                                      helpStr);
        return(1);
    }
    if(!makeTempPath(tempPath)) {
        fprintf(stderr,"%s: error: no more unique file names.\n",PRG);
        return(1);
    }
    if(!(ctl = parseCmdLine(cmdLine))) {
        fprintf(stderr,"%s: error: memory fault or malformed command.\n",PRG);
        return(1);
    }
    if(cmdLine = makeCmdLine(ctl,cmdLine,tempPath))
        system(cmdLine);
    return(0);
}

// execSubCmd() esegue un subcomando compreso nella command line di CMDSUBST.
// Viene costruita una command line contenente il subcomando e l'istruzione
// di redirezione del suo standard output nel file temporaneo il cui nome
// e' stato costruito da makeTempPath(). L'output e' poi analizzato dalla
// getSubCmdOutput() e il file temporaneo e' cancellato. Il sottocomando e'
// eseguito via system().

char *execSubCmd(char *subCmd,char *tempPath)
{
    char *ptr = NULL;
    static char outString[MAXCMD];

    if((strlen(subCmd)+strlen(tempPath)+strlen(OUTREDIR)) < MAXCMD) {
        strcat(subCmd,OUTREDIR);
        strcat(subCmd,tempPath);
        if(!system(subCmd))
            if(!getSubCmdOutput(outString,tempPath))
                fprintf(stderr,"%s: error: subcmd output not available.\n",PRG);
            else
                ptr = outString;
        else
            fprintf(stderr,"%s: error: subcmd exec failure.\n",PRG);
        unlink(tempPath);
    }
    else
        fprintf(stderr,"%s: error: subcmd string too long.\n",PRG);
    return(ptr);
}
```

```
// getCmdLine() ricostruisce la command line passata a CMDSUBST concatenando
// in un buffer static tutti gli elementi di argv. Il buffer deve essere
// static perche' l'indirizzo e' restituito alla funzione chiamante. In
// alternativa lo si potrebbe allocare con malloc()

char *getCmdLine(int argc,char **argv)
{
    register i;
    static char cmdLine[MAXCMD];

    if(argc == 1)
        return(NULL);
    cmdLine[0] = NULL;
    for(i = 1; argv[i]; i++) {
        strcat(cmdLine,argv[i]);
        strcat(cmdLine,SEPSTR);
    }
    cmdLine[strlen(cmdLine)-1] = NULL;
    return(cmdLine);
}

// getSubCmdOutput() apre il file temporaneo contenente lo standard output
// del subcomando eseguito e ne legge la prima riga, che deve essere
// sostituita al subcomando stesso nella command line di CMDSUBST.

char *getSubCmdOutput(char *outString,char *tempPath)
{
    char *ptr;
    FILE *subCmdOut;

    *outString = NULL;
    if(subCmdOut = fopen(tempPath,"r")) {
        if(!fgets(outString,MAXCMD,subCmdOut))
            *outString = NULL;
        else
            if(ptr = strrchr(outString,EOL))
                *ptr = NULL;
        fclose(subCmdOut);
    }
    return(outString);
}

// initSubCtl() inizializza una struttura di controllo del subcomando,
// riallocando l'array. La prima chiamata a initSubCtl() le passa un NULL
// come puntatore all'array, cosi' la funzione puo' usare realloc() anche
// per allocare la prima struttura.

SUBCTL *initSubCtl(SUBCTL *ctl,int items,char *cmdBase)
{
    if(!(ctl = (SUBCTL *)realloc(ctl,items*sizeof(SUBCTL)))) {
        fprintf(stderr,"%s: error: not enough memory.\n",PRG);
        return(NULL);
    }
    ctl[items-1].init = cmdBase+strlen(cmdBase);
    ctl[items-1].len = 0;
    return(ctl);
}

// makeCmdLine() costruisce la nuova command line che CMDSUBST deve eseguire
// sostituendo ai sottocomandi la prima riga del loro standard output. La
// command line cosi' costruita e' restituita a main(), che la esegue via
// system().

char *makeCmdLine(SUBCTL *ctl,char *cmdLine,char *tempPath)
```

```
{
    register i = 0, j;
    char *newPtr, *outString;
    char subCmd[MAXCMD];
    static char newCmdLine[MAXCMD];

    newPtr = newCmdLine;
    do {
        while(cmdLine < ctl[i].init-1)
            *newPtr++ = *cmdLine++;
        if(!(*ctl[i].init))
            *newPtr++ = *cmdLine;
        *newPtr = NULL;
        cmdLine += ctl[i].len+2;
        for(j = 0; j < ctl[i].len; j++)
            subCmd[j] = ctl[i].init[j];
        if(ctl[i].len) {
            subCmd[j] = NULL;
            if(!(outString = execSubCmd(subCmd,tempPath)))
                return(NULL);
            if(!(strlen(newCmdLine)+strlen(outString) < MAXCMD))
                return(NULL);
            strcat(newCmdLine,outString);
            newPtr = strchr(newCmdLine,NULL);
        }
    } while(ctl[i++].len);
    return(newCmdLine);
}

// makeTempPath() prepara un nome di file temporaneo completo di path formato
// dal drive di default e dalla root. Il file temporaneo e' percio' scritto
// in root ed e' destinato a contenere lo stdout del comando eseguito

char *makeTempPath(char *tempPath)
{
    strcpy(tempPath,ROOTDIR);
    strcat(tempPath,TEMPLATE);
    *tempPath = (char)getdisk()+'A';
    return(mktemp(tempPath+strlen(ROOTDIR)));
}

// parseCmdLine() scandisce la command line passata a CMDSUBST per individuare
// eventuali subcomandi racchiusi tra i '$'. Per ogni subcomando e' allocata
// una nuova struttura SUBCTL nell'array. Se sono incontrati due '$'
// consecutivi, si presume che essi rappresentino un solo '$' effettivo nella
// command line: uno e' scartato e non ha luogo alcuna sostituzione. La
// funzione non e' in grado di gestire sostituzioni nidificate.

SUBCTL *parseCmdLine(char *cmdLine)
{
    register i = 0, flag = 0;
    char *ptr, *cmdBase;
    SUBCTL *ctl;

    if(!(ctl = initSubCtl(NULL,1,cmdBase = cmdLine)))
        return(NULL);
    while(ptr = strchr(cmdLine,SUBST_FLAG)) {
        cmdLine = ptr+1;
        if(*cmdLine == SUBST_FLAG)
            strcpy(cmdLine,cmdLine+1);
        else {
            flag = flag ? 0 : 1;
            if(!(*ctl[i].init))
                ctl[i].init = cmdLine;
```

```
            else {
                ctl[i].len = ptr-ctl[i].init;
                ++i;
                if(!(ctl = initSubCtl(ctl,i+1,cmdBase)))
                    return(NULL);
            }
        }
    }
    return(flag ? NULL : ctl);
}
```

Qualora non sia possibile effettuare la sostituzione nella command line (riga di comando risultante eccessivamente lunga o caratteri '`$`' scorrettamente appaiati), o la funzione `system()` restituisca un codice di errore[459], `CMDSUBST` termina con un valore di `ERRORLEVEL` diverso da `0`.

NUMERI A CASO

La libreria C include alcune funzioni per la generazione di numeri casuali. Va subito precisato che si tratta di una casualità piuttosto… fittizia, in quanto essi sono generati applicando un algoritmo di una certa complessità ad un numero base, detto *seed* (seme). Dato che l'algoritmo è sempre lo stesso, ad uguale seed corrisponde uguale numero casuale che, quindi, tanto casuale poi non è (infatti si parla, per la precisione, di numeri *pseudocasuali*). Per ovviare almeno parzialmente al problema, l'algoritmo di generazione del numero casuale modifica il seed, con l'effetto di abbattere la probabilità che venga generato sempre il medesimo numero casuale.

Ciò premesso, come fare ad ottenere un numero "casuale"? Semplice: basta invocare la funzione di libreria

```
int rand(void);
```

(prototipo in `STDLIB.H`), che restituisce un numero pseudocasuale compreso tra `0` e il valore della costante manifesta `RAND_MAX` (definita in `LIMITS.H`: in molte implementazioni è pari a `32767`), anch'essa definita in `STDLIB.H`. La funzione `rand()` applica l'algoritmo sul seed, che viene modificato solo durante il calcolo. Poiché il seed è una variabile statica inzializzata a `1`, una serie di chiamate a `rand()` produce sempre la stessa sequenza di numeri casuali, come il seguente banalissimo programma può facilmente evidenziare se lanciato più volte:

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

int main(void)
{
    register int i;

    for(i = 0; i < 10; i++)
        printf("%d\n",rand());
    return(0);
}
```

Un rimedio può essere rappresentato dalla funzione

```
void srand(int newseed);
```

[459] Anche `CMDSUBST` esegue la command line effettuando una *DOS shell*, cioè invocando una seconda istanza (transiente) dell'interprete dei comandi. Vedere pag. 129.

(prototipo in `STDLIB.H`), che assegna al seed il valore passatole come parametro (trattandosi di un intero, il valore massimo è `32767`). E' evidente che successive chiamate a `rand()` producono identiche sequenze di numeri pseudocasuali, se il seed viene reinizializzato ad uno stesso valore:

```
    ....
    srand(15);
    for(i = 0; i < 10; i++)
        printf("%d\n",rand());
    srand(15);
    for(i = 0; i < 10; i++)
        printf("%d\n",rand());                    // sequenza identica alla precedente
    srand(16);
    for(i = 0; i < 10; i++)
        printf("%d\n",rand());                    // sequenza diversa dalle precedenti
    ....
```

Più interessante appare la macro

```
void randomize(void);
```

definita (ancora una volta) in `STDLIB.H`. La `randomize()` assegna al seed il valore dei 16 bit meno significativi del numero di secondi trascorsi dal 1 gennaio 1970, calcolato mediante una chiamata alla funzione `time()`[460]:

```
#include <stdlib.h>
#include <time.h>                              // per randomize(), oltre a <stdlib.h>
....
    randomize();
    for(i = 0; i < 10; i++)
        printf("%d\n",rand());
    randomize();
    for(i = 0; i < 10; i++)
        printf("%d\n",rand());                    // sequenza diversa dalla precedente
    ....
```

Come accennato poco sopra, `randomize()` è una macro e chiama la funzione `time()`, il cui prototipo si trova in `TIME.H`: di qui la necessità della direttiva

```
#include <time.h>
```

inserita nel precedente frammento di codice. Si noti che solo `rand()` e `srand()` sono normalmente disponibili sui sistemi Unix (vedere, circa la portabilità, pag. 461 e seguenti).

Generare numeri pseudocasuali compresi tra `0` e `RAND_MAX` è certamente interessante, ma spesso lo è ancora di più controllare l'ampiezza del range: in altre parole, ottenere numeri compresi tra un valore minimo ed uno massimo scelti a piacere. A tale scopo in `STDLIB.H` è definita la macro

```
int random(int maxnum);
```

che genera un numero pseudocasuale compreso tra `0` e `maxnum-1`. Ad esempio

```
    casuale = random(51);
```

---

[460] Può essere interessante sbirciare in `STDLIB.H` le definizioni di `randomize()` e `random()`, descritta oltre.

restituisce un numero compreso tra `0` e `50`, estremi inclusi, e lo assegna alla variabile (`int`) `casuale`. Se il limite inferiore del range deve essere diverso da `0`, è sufficiente sottrarre la differenza (rispetto a `0`) a `maxnum` e addizionarla al risultato:

```
    casuale = random(41)+10;
```

assegna a `casuale` un valore compreso tra `10` e `50`, estremi inclusi. La sottrazione e l'addizione descritte sono da intendersi in senso algebrico: se il limite inferiore è minore di `0`, la differenza tra questo e `0` va sommata al limite superiore e sottratta al valore restituito da `random()`:

```
    casuale = random(61)-10;
```

assegna a `casuale` un valore compreso tra `-10` e `50`, estremi inclusi. Pertanto, una semplice funzione in grado di accettare anche il limite inferiore potrebbe somigliare alla seguente:

```
#include <stdlib.h>

int random2(int limite_inf, int limite_sup)
{
    return(random(limite_sup-limite_inf+1)+limite_inf);
}
```

La `random2()` restituisce un numero pseudocasuale compreso tra `limite_inf` e `limite_sup` inclusi (si noti la differenza con `random()`, che esclude `maxnum` dal range). Va sottolineato che se l'espressione

```
limite_sup-limite_inf+1
```

restituisce un valore superiore al massimo ntero (con segno), `random()` (e, di conseguenza, anche `random2()`) restituisce sempre detto valore: nelle implementazioni in cui gli interi sono gestiti con 16 bit, pertanto, l'ampiezza massima del range è `32767`: dal momento che `random()` consente di specificare il limite superiore, questo può essere `32767` se il limite inferiore è `0`; se quest'ultimo è minore di `0`, allora quello superiore, per la formula esaminata poco sopra, è pari, al massimo, a `32767-limite_inf`.

Tale limitazione può essere superata con uno stratagemma, consistente nel considerare un qualsiasi numero come una semplice stringa di bit e quindi comporre il numero casuale "concatenando" tante stringhe quante sono necessarie:

```
/********************

    BARNINGA_Z! - 1998

    RANDOML.C - genera un numero casuale tra 0 e LONG_MAX-1

    long randoml(long upperlim);

    COMPILABILE CON TURBO C++ 2.0

        bcc -O -d -c -mx randoml.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

#include <stdlib.h>
#include <limits.h>
```

```
#define ITERATIONS (                                              \
                    ((sizeof(long) - sizeof(RAND_MAX)) * CHAR_BIT) /   \
                    (sizeof(RAND_MAX) * CHAR_BIT - 1)                  \
                    +                                                  \
                    (                                                  \
                     (                                                 \
                      ((sizeof(long) - sizeof(RAND_MAX)) * CHAR_BIT) % \
                      (sizeof(RAND_MAX) * CHAR_BIT - 1)                \
                     ) ? 1 : 0                                         \
                    )                                                  \
                   )

long cdecl randoml(long upperlim)
{
    register i;
    long randnum;

    randnum = (long)rand();
    for(i = 1; i <= ITERATIONS; i++)
        randnum += (long)rand() << (i * (sizeof(RAND_MAX) * CHAR_BIT - 1));
    return((randnum & LONG_MAX) % upperlim);
}
```

La funzione ha utilizzo analogo a quello di `random()`, ma accetta come parametro un numero compreso tra `0` e `LONG_MAX` (costante manifesta definita in `LIMITS.H`); tuttavia l'algoritmo utilizzato è sensibilmente diverso.

Infatti viene generato un primo numero casuale (mediante la funzione `rand()`), che occupa parte dei bit (quelli meno significativi) della variabile `randnum`. Per utilizzare tutti i bit disponibili[461] è necessario generare altri numeri casuali e "affiancarli", mediante operazioni di shift (vedere pag. 69), a quelli già generati: detta operazione è gestita con un ciclo `for`, che ha lo scopo di rendere la funzione portabile ai sistemi nei quali il tipo `long` è gestito con un numero di bit diverso dai 32 solitamente utilizzati con i processori Intel per personal computer.

La costante manifesta `ITERATIONS` esprime il numero di cicli necessari per utilizzare tutti i bit restanti (esclusi, cioè, quelli già impegnati dal primo numero casuale generato). La sua definizione, solo apparentemente complessa[462], esprime il rapporto tra il numero di bit ancora liberi nella variabile di tipo `long` e il numero di bit occorrenti per memorizzare un numero casuale generato da `rand()`[463], nell'ipotesi che, in entrambe le grandezze, i *sign bit* non siano utilizzati. Detto rapporto è incrementato di `1` se la divisione tra le due grandezze fornisce un resto (in pratica il quoziente è arrotondato per eccesso).

Il numero casuale generato ad ogni iterazione è memorizzato nella variabile dopo uno shift a sinistra di un numero di bit pari a quanti ne sono già stati "occupati" dai numeri casuali generati in precedenza: ciò ha l'effetto di concatenare il nuovo numero (cioè la stringa di bit che lo rappresenta) alla sinistra della sequenza di bit già valorizzati nella variabile. L'arrotondamento per eccesso (sopra descritto) del numero di iterazioni può portare la conseguenza che parte dei bit del numero casuale generato nell'ultima iterazione sia perso, in quanto "spinto" dallo shift fuori dallo spazio disponibile nella variabile, senza che ciò abbia comunque conseguenze negative.

---

[461] In una variabile di tipo `long` vi sono `sizeof(long)*CHAR_BIT` bit, dai quali va escluso quello utilizzato per la gestione del segno, mentre la dimensione (in bit) di `RAND_MAX` è pari a `sizeof(RAND_MAX)*CHAR_BIT`, meno il bit più significativo (al solito, il *sign bit*).

[462] Si osservi che, nei dividendi, `CHAR_BIT` è raccolto a fattore comune e i numeri `1` rappresentanti i *sign bit* si elidono: ciò contribuisce a rendere meno esplicito il significato della formula.

[463] Si sottrae `1` a `sizeof(RAND_MAX)*CHAR_BIT` nell'assunzione che il valore espresso da `RAND_MAX` sia da intendersi "segnato" e perciò non utilizzi il bit più significativo.

I 32 bit così valorizzati sono poi posti in AND (vedere pag. 71) con la costante manifesta LONG_MAX: in tal modo il *sign bit* è sempre uguale a 0. Infine, l'operazione modulo (pag. 68) con il parametro passato alla funzione garantisce che randoml() restituisca un valore compreso tra 0 e il parametro stesso, diminuito di 1 (in altre parole, il resto del rapporto tra il numero casuale generato e il parametro), analogamente alla random(). Ne segue che il massimo valore ammissibile in input alla funzione è LONG_MAX.

Si noti che ITERATIONS è una costante, e come tale viene calcolata dal compilatore al momento della compilazione del sorgente: nonostante l'elevato numero di calcoli necessari per ottenerla, essa non può influenzare negativamente le prestazioni del programma che utilizzi randoml(). Vale piuttosto la pena di osservare che, in tal senso, può essere rilevante il numero di iterazioni necessarie per generare un numero casuale: posto RAND_MAX pari a 32767, occorrono 2 cicli se i dati di tipo long sono gestiti con 32 bit; per dati a 64 bit le iterazioni diventano 4.

Vediamo ora un approccio alternativo, consistente nel generare un numero casuale mediante la libreria C e riproporzionarlo successivamente al limite superiore richiesto.

```
/********************

    BARNINGA_Z! - 1998

    RANDOMX.C - genera un numero casuale tra 0 e LONG_MAX-1

    long randomx(long upperlim);

    COMPILABILE CON TURBO C++ 2.0

        bcc -O -d -c -mx randomx.c

    dove -mx puo' essere -mt -ms -mc -mm -ml -mh

********************/

#include <stdlib.h>

long randomx(long upperlim)
{
    long factor;

    if(upperlim < RAND_MAX)
        return(rand() % upperlim);
    factor = upperlim / RAND_MAX + ((upperlim % RAND_MAX) ? 1 : 0);
    return((rand() * factor + randomx(factor)) % upperlim);
}
```

Se il limite superiore specificato è minore di RAND_MAX, la random() restituisce immediatamente il numero pseudocasuale generato dalla funzione di libreria rand() (l'operazione modulo con upperlim garantisce che il valore restituito sia compreso tra 0 e RAND_MAX-1). In caso contrario viene calcolato un fattore di proporzionalità come rapporto tra il limite superiore richiesto e RAND_MAX, arrotondato per eccesso. A questo punto, generando un numero casuale compreso tra 0 e RAND_MAX-1 (mediante rand()) e moltplicandolo per il fattore sopra calcolato, si ottiene un risultato grossolanamente[464] compreso nel range desiderato: si osservi però che detto risultato è sempre, per definizione, un multiplo di factor; per "tappare i buchi" tra un multiplo ed il successivo si deve sommare al numero generato un secondo numero pseudocasuale compreso tra 0 e factor-1. Ma

[464] "Grossolanamente" significa che, a causa dell'arrotondamento per eccesso applicato al calcolo del fattore di proporzionalità, il numero casuale ottenuto potrebbe essere superiore al limite indicato.

generare numeri casuali compresi tra 0 e un limite dato è esattamente il compito di randomx(), che quindi può farlo ricorsivamente, cioè chiamando se stessa (vedere pag. 100): ecco perché al prodotto rand()*factor viene sommato il valore restituito da randomx(factor). L'approccio ricorsivo consente inoltre di gestire correttamente, in modo del tutto trasparente, il caso in cui factor risulti maggiore di RAND_MAX-1 (il che si verifica quando upperlim è maggiore o uguale al quadrato di RAND_MAX).

Il massimo valore di upperlim è pari a LONG_MAX-((2*factor)-2)[465]: infatti, essendo factor arrotondato per eccesso, il prodotto rand()*factor può risultare superiore a upperlim di factor-1; inoltre si deve considerare che a detto prodotto è sommato il secondo numero casuale, anch'esso pari, al massimo, a factor-1.

Vale la pena di confrontare rapidamente randoml() e randomx().

Sotto l'aspetto della praticità di utilizzo, risulta senz'altro preferibile la randoml(), in quanto il massimo valore di upperlim è noto e costante, ed è pari a LONG_MAX-1 (il range disponibile è dunque più esteso).

D'altra parte, randomx() è portabile, a differenza di randoml(), la quale, effettuando operazioni di shift a sinistra, assume implicitamente che l'organizzazione interna delle variabili (cioè il significato dei bit che le compongono) sia di tipo *backwords* e che il *sign bit* sia sempre quello più significativo, il che non è vero per tutti i processori[466].

Infine, qualche considerazione interessante nasce circa la velocità di calcolo (assumendo, per semplicità, che i dati di tipo long siano gestiti in 32 bit). Il tempo di elaborazione richiesto da randoml() è indipendente dal valore di upperlim: esso è, anzi, costante, in quanto per la generazione di un numero casuale la funzione effettua sempre tre chiamate a rand(). Al contrario, il tempo richiesto da randomx() aumenta al crescere di upperlim; tuttavia non vi è un rapporto di proporzionalità diretta. Infatti, per valori di upperlim inferiori a RAND_MAX viene effettuata una sola chiamata a rand(); per valori di upperlim compresi tra RAND_MAX e il suo quadrato ne vengono effettuate due; infine, rand() è chiamata tre volte per valori uguali o superiori al quadrato di RAND_MAX. Sulla scorta di quanto esposto[467], ci si può aspettare che il tempo di elaborazione di randomx() sia pari a circa un terzo di quello della randoml() per valori di upperlim minori di RAND_MAX, per poi passare a circa due terzi fino al quadrato di questo; i tempi di elaborazione dovrebbero essere assai vicini per valori di upperlim uguali o superiori al quadrato di RAND_MAX.

Una verifica empirica (effettuata cronometrando entrambe le funzioni, eseguite centomila volte con diversi valori di upperlim) conferma quanto ipotizzato per valori di upperlim inferiori a

---

[465] Ne segue che, da un punto di vista formale, il metodo per conoscerlo consiste nel risolvere il seguente sistema di due equazioni a due incognite:

```
upperlim = LONG_MAX - 2 * factor - 2
factor = upperlim / RAND_MAX
```

che, risolto, fornisce per upperlim un valore (arrotondato per difetto) di 2147352578 (assumendo che i dati di tipo long siano gestiti con 32 bit).

[466] Una versione portabile di randoml() dovrebbe utilizzare moltiplicazioni (per 2 o per potenze di 2) in luogo degli shift. Il carico elaborativo (tanto delle moltiplicazioni quanto del calcolo delle potenze, da effettuare dinamicamente) risulterebbe estremamente penalizzante.

[467] Quindi senza considerare l'*overhead* introdotto dagli shift in randoml() e dal calcolo di factor in randomx().

RAND_MAX (randomx() impiega meno della metà di randoml())[468] e compresi tra RAND_MAX e UINT_MAX (randomx() impiega circa due terzi del tempo richiesto da randoml()). Per valori compresi tra UINT_MAX e il quadrato di RAND_MAX si verifica invece un inatteso e considerevole aumento del tempo di elaborazione di randomx(), che risulta significativamente superiore a quello di randoml(), ma costante in tutto l'intervallo. Per valori superiori al quadrato di RAND_MAX il tempo di elaborazione di randomx() aumenta (come atteso) di una quantità circa pari all'incremento verificatosi tra il primo e il secondo dei range controllati. L'analisi dei sorgenti assembler di randomx(), generati mediante l'opzione -S del compilatore, evidenzia la causa del rallentamento inaspettato: le operazioni aritmetiche coinvolte nell'algoritmo di calcolo (in particolare moltiplicazione, divisione e modulo) vengono effettuate mediante funzioni di libreria non documentate, che ottimizzano il calcolo utilizzando algoritmi semplificati se le word più significative dei dati a 32 bit sono pari a 0: è proprio il caso dei valori compresi tra 0 e UINT_MAX. Attribuendo alla maggiore complessità del calcolo il rallentamento osservato per upperlim compreso tra UINT_MAX e il quadrato di RAND_MAX, il comportamento di randomx() si riconduce a quanto teoricamente previsto. Compilando randomx() con l'opzione -3 del compilatore, che genera codice specifico per processori a 32 bit, una parte dei calcoli sui dati long viene risolta utilizzando direttamente le istruzioni e i registri del processore, senza l'ausilio delle funzioni di libreria, con un considerevole miglioramento della performance. Di conseguenza, il tempo di elaborazione di randomx() diviene circa pari a quello di randoml() per valori di upperlim compresi tra UINT_MAX e il quadrato di RAND_MAX, e risulta superiore solo per valori maggiori di quest'ultimo.

[468] La stessa verifica ha inoltre chiaramente evidenziato che il tempo di elaborazione di randoml() è, effettivamente, costante.

CONTENUTO DEL FLOPPY DISK

Il floppy disk allegato costituisce una raccolta di esempi (programmi e funzioni) estratti dal testo. Essi sono presenti in forma sorgente e, ove possibile, compilata (object, libreria, eseguibile, file binario).

ATTENZIONE: a causa del poco tempo a disposizione non è stato possibile verificare a fondo il funzionamento di tutto il materiale raccolto nel disco (e, in generale, nel testo); si declina ogni responsabilità per qualsivoglia conseguenza derivante, direttamente o indirettamente, dall'utilizzo del medesimo.

Il contenuto del disco è suddiviso come segue:

```
\
+---SOURCES
_   +---CLIPPER
_   +---DEVDRV
_   _   +---TOOLKIT
_   _       +---DRVSET
_   _       +---LIBMOD
_   _       \---TEST
_   +---FUNC
_   _   \---TEST
_   +---PROGS
_   +---SCHED411
_   +---SYNTAX
_   \---UTIL
\---COMPILED
    +---CLIPPER
    +---DEVDRV
    _   \---TOOLKIT
    _       \---TEST
    +---FUNC
    _   \---TEST
    +---PROGS
    +---SCHED411
    \---UTIL
```

E' immediato notare una corrispondenza quasi buinivoca tra l'albero sottostante alla directory SOURCES e quello sottostante alla COMPILED: directory sottostanti la COMPILED, omologhe ad altre sottostanti la SOURCES, contengono il risultato della compilazione del contenuto di queste ultime.

\

Il file FLOPPY.TXT e le directory SOURCES e COMPILED.

\SOURCES

Raccoglie tutte le subdirectory contenenti sorgenti.

\SOURCES\CLIPPER

Contiene i sorgenti di alcune funzioni C richiamabili in applicazioni Clipper. Sono presenti sul disco i corrispondenti moduli oggetto.

`\SOURCES\DEVDRV`

Contiene il sorgente di un device driver realizzato in C e inline assembly. Sul disco è presente il corrispondente file binario.

`\SOURCES\DEVDRV\TOOLKIT`

Raccoglie le directory relative al progetto di toolkit per lo sviluppo di device driver descritto nel testo.

`\SOURCES\DEVDRV\TOOLKIT\DRVSET`

Contiene il sorgente della utility DRVSET descritta nel testo. Sul disco è presente l'eseguibile.

`\SOURCES\DEVDRV\TOOLKIT\LIBMOD`

Contiene i sorgenti C e assembler dello startup module e della libreria, compreso l'include file. Sul disco è presente il toolkit completo sotto forma di include file, modulo oggetto e libreria.

`\SOURCES\DEVDRV\TOOLKIT\TEST`

Contiene i sorgenti C di due device drivers e una applicazione, atti a testare le capacità del toolkit. Sul disco sono presenti i due files binari e l'eseguibile.

`\SOURCES\FUNC`

Raccoglie i sorgenti di funzioni aventi varia finalità (gestione della memoria, dei file, del CMOS, dei pathnames, e altro). Le funzioni, a differenza di quanto avviene nel testo, sono dotate dei necessari include file: è così possibile la creazione di una libreria. Nella directory è presente anche il file di comandi per TLIB (`BZC.LST`) e un file batch (`MAKELIB.BAT`) per la costruzione della libreria. La libreria stessa (per tutti i modelli di memoria) e gli include file sono presenti sul disco tra i file compilati.

`\SOURCES\FUNC\TEST`

Raccoglie i sorgenti di programmi che consentono di testare alcune delle funzioni sopra citate. Sono presenti sul disco anche i corrispondenti eseguibili.

`\SOURCES\PROGS`

Contiene i sorgenti di alcuni semplici programmi dimostrativi. Sono presenti sul disco i corrispondenti eseguibili.

`\SOURCES\SCHED411`

Contiene i sorgenti e i make file della utility SCHED (versione 4.11). Sono presenti sul disco i corrispondenti eseguibili. Si tenga presente che per compilare SCHED è necessario utilizzare la libreria delle funzioni descritte nel testo (vedere `\SOURCES\FUNC` e `\COMPILED\FUNC`) per il modello di memoria COMPACT.

`\SOURCES\SYNTAX`

Raccoglie esempi di funzioni e programmi strettamente attinenti particolarità sintattiche e simili. Detti sorgenti non sono stati compilati, dal momento che alcuni includono volutamente errori aventi finalità esplicative.

`\SOURCES\UTIL`

Contiene i sorgenti di alcuni programmi di utilità presentati nel testo. Sul disco sono presenti i corrispondenti eseguibili.

\COMPILED

Raccoglie le directory contenenti i file compilati (eseguibili, librerie e corrispondenti include file, moduli oggetto, file binari).

\COMPILED\CLIPPER

Contiene i moduli oggetto (direttamente consolidabili a moduli oggetto Clipper) di funzioni richiamabili in sorgenti Clipper.

\COMPILED\DEVDRV

Contiene un device driver in grado di installare un nuovo buffer di tastiera.

\COMPILED\DEVDRV\TOOLKIT

Contiene il toolkit di sviluppo per device driver, comprendente il modulo oggetto di startup, la libreria e il necessario include file.

\COMPILED\DEVDRV\TOOLKIT\TEST

Contiene due device drivers atti a testare le funzionalità offerte dal toolkit ed un eseguibile necessario al pilotaggio IOCTL di uno di essi.

\COMPILED\FUNC

Contiene una libreria (per tutti i modelli di memoria) che raccoglie le funzioni presentate nel testo a titolo di esempio ed i corrispondenti include file, che le raggruppano per "argomento". Vedere gli include file per l'elenco completo delle funzioni disponibili.

\COMPILED\FUNC\TEST

Contiene alcuni programmi eseguibili compilati facendo uso della libreria sopra descritta. I sorgenti RANDOML.C e RANDOMX.C (in \SOURCES\FUNC) sono stati compilati sia a 16 bit (RANDOML.EXE e RANDOMX.EXE), sia a 32 bit (RANDOML3.EXE e RANDOMX3.EXE).

\COMPILED\PROGS

Contiene alcuni eseguibili dimostrativi.

\COMPILED\SCHED411

Contiene la utility SCHED versione 4.11. Si tratta di un programma TSR in grado di pilotare il personal computer in modo del tutto automatico, eseguendo comandi o macro di tastiera al momento voluto, in base ad una tabella definibile dall'utente. Sono presenti nella directory, oltre al programma SCHED.EXE, le utility di conversione della tabella eventi da versione 3.5 a versione 4.x e viceversa, due file di ridefinizione degli hotkey di controllo di SCHED, i file di help e messaggi in versione italiana e inglese, il manuale (SCHED.TXT) in formato ASCII.

\COMPILED\UTIL

Contiene alcuni programmi eseguibili di utilità.

Per completezza sono elencati i files contenuti in ogni directory, in ordine alfabetico crescente, per estensione e per nome.

\
_ FLOPPY.TXT

```
_
+---SOURCES
_   +---CLIPPER
_   _       CL_BDOS.C
_   _       CL_EXENM.C
_   _       CL_MODF.C
_   _
_   +---DEVDRV
_   _   _   KBDBUF.C
_   _   _
_   _   +---TOOLKIT
_   _       +---DRVSET
_   _       _       DRVSET.C
_   _       _
_   _       +---TEST
_   _       _       TESTINIT.BAT
_   _       _       TESTDRV.BAT
_   _       _       DEVIOCTL.C
_   _       _       TESTINIT.C
_   _       _       TESTDRV.C
_   _       _       YES.TXT
_   _       _
_   _       +---LIBMOD
_   _               DDSEGCOS.ASI
_   _               DDBUIBPB.ASM
_   _               DDDEVCLO.ASM
_   _               DDDEVOPE.ASM
_   _               DDDUMMY.ASM
_   _               DDENDOFS.ASM
_   _               DDGENIOC.ASM
_   _               DDGETLOG.ASM
_   _               DDHEADER.ASM
_   _               DDINIT.ASM
_   _               DDINPFLU.ASM
_   _               DDINPIOC.ASM
_   _               DDINPND.ASM
_   _               DDINPSTA.ASM
_   _               DDINPUT.ASM
_   _               DDMEDCHE.ASM
_   _               DDMEDREM.ASM
_   _               DDOUTBUS.ASM
_   _               DDOUTFLU.ASM
_   _               DDOUTIOC.ASM
_   _               DDOUTPUT.ASM
_   _               DDOUTVER.ASM
_   _               DDOUTSTA.ASM
_   _               DDRESVEC.ASM
_   _               DDSAVVEC.ASM
_   _               DDSETCMD.ASM
_   _               DDSETLOG.ASM
_   _               DDSETSTK.ASM
_   _               DD_EXPTR.ASM
_   _               DD_VECT.ASM
_   _               DDDISCRD.C
_   _               BZDD.H
_   _               BZDD.LST
_   _
_   +---FUNC
_   _   _   MAKELIB.BAT
_   _   _   A20DISAB.C
_   _   _   A20ENABL.C
_   _   _   ALCSTRAT.C
_   _   _   BOOT.C
_   _   _   CHAINVEC.C
```

```
_    _    _    CLEARKBD.C
_    _    _    CTLALDEL.C
_    _    _    CTLBREAK.C
_    _    _    DATE2JUL.C
_    _    _    DOSPTRS.C
_    _    _    DOSSEG.C
_    _    _    EMBALLOC.C
_    _    _    EMBFREE.C
_    _    _    EMBRESIZ.C
_    _    _    EMMALLOC.C
_    _    _    EMMFRAME.C
_    _    _    EMMFREEP.C
_    _    _    EMMFREEH.C
_    _    _    EMMGHNAM.C
_    _    _    EMMGPMD.C
_    _    _    EMMMOVM.C
_    _    _    EMMOHNDL.C
_    _    _    EMMPGMAP.C
_    _    _    EMMPPH.C
_    _    _    EMMRPM.C
_    _    _    EMMSHNAM.C
_    _    _    EMMSPM.C
_    _    _    EMMTEST.C
_    _    _    EMMTEST2.C
_    _    _    EMMTOTP.C
_    _    _    EMMVER.C
_    _    _    EXTFREE.C
_    _    _    EXTINST.C
_    _    _    FATTOR.C
_    _    _    FIRSTMCB.C
_    _    _    GETRDIR.C
_    _    _    HMAALLOC.C
_    _    _    HMADEALL.C
_    _    _    INDOSADR.C
_    _    _    ISA20ON.C
_    _    _    ISFSAME.C
_    _    _    ISHMAFRE.C
_    _    _    ISLEAPYR.C
_    _    _    ISREMOTE.C
_    _    _    JUL2DATE.C
_    _    _    LASTTSR.C
_    _    _    MCBCHAIN.C
_    _    _    PARSEMCB.C
_    _    _    PARSEOPT.C
_    _    _    PATHNAME.C
_    _    _    PRNTOSCR.C
_    _    _    RANDOML.C
_    _    _    RANDOMX.C
_    _    _    READCMOS.C
_    _    _    RELENV.C
_    _    _    RELENV2.C
_    _    _    RELENVC.C
_    _    _    RESMEM.C
_    _    _    RSLVPATH.C
_    _    _    SCANDIR.C
_    _    _    UMBALLOC.C
_    _    _    UMBDOS.C
_    _    _    UMBFREE.C
_    _    _    UMBQEMM.C
_    _    _    WRITCMOS.C
_    _    _    XMSFREEB.C
_    _    _    XMMDVERS.C
_    _    _    XMMADDR.C
_    _    _    XMMISHMA.C
```

```
_   _   _   XMMVERS.C
_   _   _   XMSFREEM.C
_   _   _   XMSMOVM.C
_   _   _   DATES.H
_   _   _   INT.H
_   _   _   MEM.H
_   _   _   PARSEOPT.H
_   _   _   TSR.H
_   _   _   VAR.H
_   _   _   BZC.LST
_   _   _
_   _   +---TEST
_   _           BOOT.C
_   _           CTLBREAK.C
_   _           CTLALDEL.C
_   _           EMS.C
_   _           JULTEST.C
_   _           RANDOML.C
_   _           RANDOMX.C
_   _           SCANDIR.C
_   _
_   +---PROGS
_   _       GETCMD.C
_   _       PROV2TSR.C
_   _       PROVATSR.C
_   _       TURBOC.C
_   _
_   +---SCHED411
_   _       MAKE.BAT
_   _       MAKEV.BAT
_   _       SCHED.C
_   _       SCHED3B4.C
_   _       SCHED4B3.C
_   _       SCHED.H
_   _
_   +---SYNTAX
_   _       ALLOC.C
_   _       ALLOC2.C
_   _       ARGCARGV.C
_   _       ARGCARG2.C
_   _       ARRAY.C
_   _       AUTOVAR.C
_   _       AUTOVAR2.C
_   _       CIAO.C
_   _       CIAO2.C
_   _       EXTERN.C
_   _       EXTERN2.C
_   _       EXTERN3.C
_   _       EXTERN4.C
_   _       EXTERN5.C
_   _       EXTERN6.C
_   _       EXTERN7.C
_   _       FUNC.C
_   _       FUNC2.C
_   _       FUNC3.C
_   _       FUNC4.C
_   _       FUNC5.C
_   _       FUNCPTR.C
_   _       OPEINT16.C
_   _       OPEINT32.C
_   _       ROSSO.C
_   _       STATIC.C
_   _       STATIC2.C
_   _       STRUCT.C
```

```
_   _       STRUCT2.C
_   _       STRUCT3.C
_   _
_   +---UTIL
_           CMDSUBST.C
_           CMOSBKP.C
_           CUT.C
_           DATECMD.C
_           DISINFES.C
_           DOLIST.C
_           EMPTYLVL.C
_           FCREATE.C
_           KBDCODES.C
_           KBDPLUS.C
_           KBDPLUS2.C
_           SELSTR.C
_           SHFVWRIT.C
_           SSS.C
_           TIMEGONE.C
_           VIDEOCAP.C
_           ZAPTSR.C
_
+---COMPILED
    +---CLIPPER
    _       CL_BDOS.OBJ
    _       CL_EXENM.OBJ
    _       CL_MODF.OBJ
    _
    +---DEVDRV
    _   _   KBDBUF.SYS
    _   _
    _   +---TOOLKIT
    _       _   DRVSET.EXE
    _       _   BZDD.H
    _       _   BZDD.LIB
    _       _   DDHEADER.OBJ
    _       _
    _       +---TEST
    _               DEVIOCTL.EXE
    _               TESTDRV.SYS
    _               TESTINIT.SYS
    _
    +---FUNC
    _   _   DATES.H
    _   _   INT.H
    _   _   MEM.H
    _   _   PARSEOPT.H
    _   _   TSR.H
    _   _   VAR.H
    _   _   BZCC.LIB
    _   _   BZCH.LIB
    _   _   BZCL.LIB
    _   _   BZCM.LIB
    _   _   BZCS.LIB
    _   _
    _   +---TEST
    _           BOOT.EXE
    _           CTLALDEL.EXE
    _           CTLBREAK.EXE
    _           EMS.EXE
    _           RANDOML.EXE
    _           RANDOML3.EXE
    _           RANDOMX.EXE
    _           RANDOMX3.EXE
```

```
_           SCANDIR.EXE
_
+---PROGS
_       GETCMD.EXE
_       PROV2TSR.EXE
_       PROVATSR.EXE
_       TURBOC.EXE
_
+---SCHED411
_       SCHED.EXE
_       SCHED3B4.EXE
_       SCHED4B3.EXE
_       SCHED.HLP
_       SCHED.ITH
_       SCHED.ITM
_       SCHED.KEY
_       SCHED.KKK
_       SCHED.MSG
_       SCHED.TXT
_       SCHED.UKH
_       SCHED.UKM
_
+---UTIL
        CMDSUBST.EXE
        CMOSBKP.EXE
        CUT.EXE
        DATECMD.EXE
        DISINFES.EXE
        DOLIST.EXE
        EMPTYLVL.EXE
        FCREATE.EXE
        KBDCODES.EXE
        KBDPLUS.EXE
        KBDPLUS2.EXE
        SELSTR.EXE
        SHFVWRIT.EXE
        SSS.EXE
        TIMEGONE.EXE
        VIDEOCAP.EXE
        ZAPTSR.EXE
```

INDICE DELLE FIGURE

SEGMENTAZIONE DELLA MEMORIA NEL MODELLO TINY. 144

SEGMENTAZIONE DELLA MEMORIA NEL MODELLO SMALL. 144

SEGMENTAZIONE DELLA MEMORIA NEL MODELLO MEDIUM. 145

SEGMENTAZIONE DELLA MEMORIA NEL MODELLO COMPACT. 145

SEGMENTAZIONE DELLA MEMORIA NEL MODELLO LARGE...146

SEGMENTAZIONE DELLA MEMORIA NEL MODELLO HUGE..147

UTILIZZO, DA PARTE DEL DOS, DELLA MEMORIA CONVENZIONALE. 191

LA STRUTTURA DEI MEMORY CONTROL BLOCK..191

UTILIZZO DEGLI INDIRIZZI DI MEMORIA TRA I 640 KB E IL MEGABYTE. 198

SCHEMA DI MAPPING EMM TRA PAGINE FISICHE (EMM PAGE FRAME) E PAGINE LOGICHE. 202

LO STACK DOPO L'INGRESSO NEL GESTORE DI INTERRUPT DICHIARATO CON PARAMETRI FORMALI. 257

LO STACK DOPO L'INGRESSO NEL GESTORE far DI INTERRUPT DICHIARATO CON PARAMETRI FORMALI. 261

LO STACK IN chainvector() PRIMA DELL'ESECUZIONE DELLA RET. 264

LA STRUTTURA DI UN TSR. 276

LA STRUTTURA DI TSR GENERATA DAL COMPILATORE C...277

LA STRUTTURA DI TSR GENERATA DAL LINKER...289

STRUTTURA DI UN DEVICE DRIVER...356

COMUNICAZIONE TRA APPLICAZIONE E PERIFERICA VIA DEVICE DRIVER. 357

INDICE DELLE TABELLE

TIPI DI DATO IN C 12
TIPI DI DATO E DICHIARATORI 15
OPERATORI C 62
PARAMETRI DI main() 106
STREAM STANDARD 116
MODALITÀ DI APERTURA DEL FILE CON fopen() 119
MODALITÀ OPERATIVE DI fseek() 121
MODALITÀ DI CACHING CON setvbuf() 125
MODALITÀ DI APERTURA DEL FILE CON open(): PARTE 1 126
MODALITÀ DI APERTURA DEL FILE CON open(): PARTE 2 127
PERMESSI DI ACCESSO AL FILE CON open() 127
MODALITÀ DI CONDIVISIONE DEL FILE CON _open() 128
INT 13H, SERV. 02H: LEGGE SETTORI IN UN BUFFER 140
VALORI DELLA MACRO __TURBOC__ 174
INT 21H, SERV. 48H: ALLOCA UN BLOCCO MEMORIA 189
INT 21H, SERV. 49H: DEALLOCA UN BLOCCO DI MEMORIA 190
INT 21H, SERV. 4AH: MODIFICA L'AMPIEZZA DEL BLOCCO DI MEMORIA ALLOCATO 190
INT 21H, SERV. 34H: INDIRIZZO DELL'INDOS FLAG 192
INT 21H, SERV. 52H: INDIRIZZO DELLA LISTA DELLE LISTE 194
INT 21H, SERV. 58H: GESTIONE DELLA STRATEGIA DI ALLOCAZIONE 197
INT 67H, SERV. 46H: VERSIONE EMM 204
INT 67H, SERV. 41H: INDIRIZZO DELLA PAGE FRAME 205
INT 67H, SERV. 42H: NUMERO DI PAGINE 206
INT 67H, SERV. 4BH: NUMERO DI HANDLE EMM APERTI 208
INT 67H, SERV. 4DH: PAGINE ALLOCATE AGLI HANDLE 208
INT 67H, SERV. 53H: NOME DELLO HANDLE EMS 210
INT 67H, SERV. 43H: ALLOCA PAGINE LOGICHE NELLA MEMORIA ESPANSA 213
INT 67H, SERV. 44H: EFFETTUA IL MAPPING DI PAGINE LOGICHE A PAGINE FISICHE 214
INT 67H, SERV. 45H: DEALLOCA LE PAGINE ASSOCIATE AD UNO HANDLE 215
INT 67H, SERV. 4EH: SALVA E RIPRISTINA LA PAGE MAP 219
INT 67H, SERV. 57H: TRASFERISCE DA MEM. CONVENZ. A MEM. EMS E VICEVERSA 221
FORMATO DEL BUFFER UTILIZZATO DALL'INT 67H, SERV. 57H 222

INT 67H, SERV. 50H: MAPPING MULTIPLO E SU MEMORIA CONVENZIONALE....................224
CODICI DI ERRORE EMS 225
INT 2FH: PRESENZA DEL DRIVER XMM...227
INT 2FH: ENTRY POINT DEL DRIVER XMM...227
SERVIZIO XMS 00H: VERSIONE DEL DRIVER XMM E ESISTENZA DELLA HMA..................228
SERVIZIO XMS 08H: QUANTITÀ DI MEMORIA XMS DISPONIBILE.......................................230
INT 15H, SERV. 88H: MEMORIA ESTESA (NON XMS) DISPONIBILE 232
SERVIZIO XMS 09H: ALLOCA UN BLOCCO DI MEMORIA XMS..233
SERVIZIO XMS 0BH: COPIA TRA AREE DI MEMORIA CONVENZIONALE O XMS................234
FORMATO DEL BUFFER UTILIZZATO DAL SERVIZIO XMS 0BH...235
SERVIZIO XMS 0AH: DEALLOCA UNO HANDLE XMS..237
SERVIZIO XMS 0FH: ALLOCA UN UMB 238
SERVIZIO XMS 01H: ALLOCA LA HMA (SE DISPONIBILE)..240
INT 2FH, SERV. 4AH: GESTISCE LA PORZIONE LIBERA DI UNA HMA GIÀ ALLOCATA.......240
SERVIZIO XMS 02H: DEALLOCA LA HMA..241
SERVIZIO XMS 07H: STATO DELLA A20 LINE...243
SERVIZIO XMS 03H: ABILITA LA A20 LINE...244
SERVIZIO XMS 04H: DISABILITA LA A20 LINE..244
SERVIZIO XMS 0FH: ALLOCA UN UMB 246
SERVIZIO XMS 11H: DEALLOCA UN UN UMB 247
CODICI DI ERRORE XMS...248
INT 21H, SERV. 25H: SCRIVE UN INDIRIZZO NELLA TAVOLA DEI VETTORI.......................252
INT 21H, SERV. 35H: LEGGE UN INDIRIZZO NELLA TAVOLA DEI VETTORI........................252
MODALITÀ DI UTILIZZO DEI GESTORI ORIGINALI DI INTERRUPT......................................262
INT 21H, SERV. 31H: TERMINA MA RESTA RESIDENTE IN RAM...276
INT 21H, SERV. 4DH: CODICE DI USCITA DELL'APPLICAZIONE TERMINATA......................277
VALORI DEL FLAG DI STATO DELL'INT 05H 299
SHIFT STATUS BYTE ED EXTENDED SHIFT STATUS BYTE..302
KEYBOARD STATUS BYTE..303
INT 16H, SERV. 00H: LEGGE UN TASTO DAL BUFFER DI TASTIERA 305
INT 16H, SERV. 01H: CONTROLLA SE È PRESENTE UN TASTO NEL BUFFER DI TASTIERA...306
INT 16H, SERV. 02H: STATO DEGLI SHIFT 306
INT 16H, SERV. 05H: INSERISCE UN TASTO NEL BUFFER DELLA TASTIERA.......................306
PUNTATORI AL BUFFER DI TASTIERA...307
INT 2FH: MULTIPLEX INTERRUPT 312

INT 21H, SERV. 0AH: INPUT DI UNA STRINGA MEDIANTE BUFFER 313
INT 10H, SERV. 00H: STABILISCE NUOVI MODO E PAGINA VIDEO 314
INT 10H, SERV. 02H: SPOSTA IL CURSORE ALLE COORDINATE SPECIFICATE....................315
INT 10H, SERV. 03H: LEGGE LA POSIZIONE DEL CURSORE...315
INT 10H, SERV. 05H: STABILISCE LA NUOVA PAGINA VIDEO...315
INT 10H, SERV. 08H: LEGGE CARATTERE E ATTRIBUTO ALLA POSIZIONE DEL CURSORE...315
INT 10H, SERV. 09H: SCRIVE CARATTERE E ATTRIBUTO ALLA POSIZIONE DEL CURSORE...316
INT 10H, SERV. 0EH: SCRIVE UN CARATTERE IN MODO TTY 316
INT 10H, SERV. 0FH: LEGGE IL MODO E LA PAGINA VIDEO ATTUALI................................316
INT 10H, SERV. 13H: SCRIVE UNA STRINGA CON ATTRIBUTO 317
INT 21H, SERV. 3CH: CREA UN NUOVO FILE O NE TRONCA UNO ESISTENTE.....................319
INT 21H, SERV. 3DH: APRE UN FILE ESISTENTE 320
INT 21H, SERV. 3EH: CHIUDE UN FILE APERTO..320
INT 21H, SERV. 3FH: LEGGE DA UN FILE APERTO..321
INT 21H, SERV. 40H: SCRIVE IN UN FILE APERTO 321
INT 21H, SERV. 41H: CANCELLA UN FILE...321
INT 21H, SERV. 42H: MUOVE IL PUNTATORE ALLA POSIZIONE NEL FILE..........................322
INT 21H, SERV. 2FH: OTTIENE DAL DOS L'INDIRIZZO DEL DTA ATTUALE 323
INT 21H, SERV. 1AH: COMUNICA AL DOS L'INDIRIZZO DEL DTA..323
SERVIZI DELL'INT 21H UTILIZZANTI IL DTA 323
INT 21H, SERV. 62H: OTTIENE DAL DOS L'INDIRIZZO DEL PSP ATTUALE 324
INT 21H, SERV. 50H: COMUNICA AL DOS L'INDIRIZZO DEL PSP..324
INT 21H, SERV. 44H, SUBF. 01H: MODIFICA GLI ATTRIBUTI DI UN DEVICE DRIVER..........355
STRUTTURA DEL DEVICE DRIVER HEADER..358
STRUTTURA DELLA DEVICE ATTRIBUTE WORD...359
STRUTTURA DEL DEVICE DRIVER REQUEST HEADER 360
STRUTTURA DELLA DEVICE DRIVER STATUS WORD...361
ELENCO DEI SERVIZI IMPLEMENTABILI DAI DEVICE DRIVER 362
DEVICE DRIVER, SERV. 00: USO DEL REQUEST HEADER...363
STRUTTURA DEL BPB..365
DEVICE DRIVER, SERV. 01: USO DEL REQUEST HEADER...366
DEVICE DRIVER, SERV. 02: USO DEL REQUEST HEADER...367
DEVICE DRIVER, SERV. 03: USO DEL REQUEST HEADER...368
DEVICE DRIVER, SERV. 04: USO DEL REQUEST HEADER...369

DEVICE DRIVER, SERV. 05: USO DEL REQUEST HEADER 370
DEVICE DRIVER, SERV. 06: USO DEL REQUEST HEADER 370
DEVICE DRIVER, SERV. 07: USO DEL REQUEST HEADER 371
DEVICE DRIVER, SERV. 13: USO DEL REQUEST HEADER 372
DEVICE DRIVER, SERV. 15: USO DEL REQUEST HEADER 373
DEVICE DRIVER, SERV. 16: USO DEL REQUEST HEADER 374
DEVICE DRIVER, SERV. 19: USO DEL REQUEST HEADER 375
DEVICE DRIVER, SERV. 23: USO DEL REQUEST HEADER 376
DEVICE DRIVER, SERV. 24: USO DEL REQUEST HEADER 376
INT 21H, SERV. 44H, SUBF. 0CH E 0DH: GENERIC IOCTL REQUEST 456
SUFFISSI PER LE COSTANTI MANIFESTE DEFINITE IN STDDEF.H 462
INT 21H, SERV. 60H: RISOLVE UN PATH IN PATHNAME COMPLETO 467
INT 21H, SERV. 37H, SUBF. 00H: GET SWITCH CHARACTER... 498
INT 21H, SERV. 37H, SUBF. 01H: SET SWITCH CHARACTER 498
INT 21H, SERV. 44H, SUBF. 09H: DETERMINA SE IL DRIVE È REMOTO 500
INT 21H, SERV. 0CH: PULISCE IL BUFFER DI TASTIERA E INVOCA UN SERVIZIO............... 527
INTERRUPT E SERVIZI DI CARICAMENTO E TERMINAZIONE DEI PROGRAMMI 550

INDICE ANALITICO

#

#define; 47; 61; 186; 294; 570
#ifdef; 49
#ifndef; 49
#include; 8
#pragma; 184; 190; 272
#undef; 49

_

__emit__(); 184
__IOerror(); 163; 534
_doserrno; 533
_envseg; 307
_errno; 420
_fmemcpy(); 357
_fmode; 127; 136
_fstrcpy(); 162
_heaplen; 605
_open(); 136
_osmajor; 320; 420; 476
_osminor; 320; 420; 476
_osversion; 420; 476
_psp; 300; 307; 347; 418; 476
_restorezero(); 321; 351; 355; 441–44; 441–44
_saveregs; 407
_setargv__(), _setenvp__(); 509
_version; 420; 476

3

3, opzione compilatore; 314; 661

A

A20 Line; 244; 261
accessibilità delle variabili; 36
address wrapping; 244
ALLOC.H; 117; 203
allocazione dinamica della memoria; 117–21
    nei TSR; 302
allocmem(); 203; 303; 590
allocUMB(); 265
altoparlante; 572
and
    logico; 75
    su bit; 76
ANSI; 7; 46
AreYouLast(); 354
argc, argv; 113; 508; 512
    nel toolkit per i device driver; 472
argv; 143
aritmetica dei puntatori; 34
array; 30–34; 117
    di strutture; 55
ASCII code; 327; 545; 561
ASCII, formato; 604
ASCIIZ; 227; 342; 392; 485
assegnamento; 78
assembler; 182
    registri; 175
    variabili e indirizzi; 178
Assembler; 169–85
associatività degli operatori; 65
atoi(); 114; 119
autodecremento; 69
    portabilità; 493
autoincremento; 28; 48; 69
    portabilità; 493
automatic (variabili); 37

B

backslash; 46; 48; 495
backwords; 23; 60; 180; 269; 365; 493
batch; 603
bdos(), bdosptr(); 152
BIOS; 9; 149; 315; 322; 379; 394; 403; 541
block device driver; 380
boot(); 184
bootstrap; 204; 379; 389; 580; 588; 599
BPB; 393
break; 84; 87; 88
BYTE; 475
BZDD.H; 454; 474

C

c, opzione compilatore; 166; 310; 410; 454
C, opzione del librarian; 454
c, opzione linker; 410
C++; 2; 51; 95; 135
caching; 132; 629
CALL; 286; 322; 331; 334
CALL, istruzione batch; 646
campi

di bit; 63
di struct; 52
di union; 61
case; 84
case sensitivity; 7; 165
cast; 70; 280; 300; 304; 332; 333; 357; 502; 589
cdecl; 99; 208
chainvector(); 283
char; 16
charcter device driver; 380
child; 139
child process; 137; 238; 329; 343; 543; 588
cicli; 86–90
CLI; 270
Clipper; 191–202
allocazione della memoria; 196
EXTEND.H; 191
passaggio parametri al C; 192
puntatori; 195
reference; 192
restituzione valori dal C; 194
CLIPPER, macro; 192
close(); 134
CMDSUBST; 648
CMOS; 250; 592
code segment
nei device driver; 416
code segment, dati nel; 590
COM, eseguibile; 155; 300; 409; 536; 541
COM1:; 124
comandi interni ed esterni; 603
command line; 348; 357; 389; 427; 508–33
gestione; 463; 511–33
sintassi; 512
switch character; 517
toolkit C; 514–33
nei device driver; 450–53
PSP; 510
COMMAND.COM; 205; 295; 380; 617
commenti; 14; 101
compact model; 158
compilatore; 3; 298; 301; 326; 335; 379; 403; 409
error; 5
opzioni per i device driver; 471
portabilità dei sorgenti; 493
warning; 5
compilazione condizionale; 166; 190; 228
complemento a due; 492
complemento a uno; 68; 492
comunicazione inter-process; 148
CON:; 124
CON\:; 385; 470
CONFIG.SYS; 380; 389; 410; 481
CONIO.H; 95
continue; 87; 88
conversioni di tipo; 70
COPY; 618
costanti; 45–50
character; 45
esadecimali; 45
floating point; 46
long; 45
manifeste; 47
simboliche; 50
unsigned; 45
CP/M; 495
CritErr flag; 316; 333; 348; 349
CS; 186
CS\:IP; 106; 161; 273
CTRL-ALT-DEL; 290; 322
CTRL-BREAK, CTRL-C; 287; 322; 336; 348; 350; 380; 395; 571
nei TSR; 329; 331
CTRL-Z; 117
CUT; 635

D

DATECMD; 606
dati; 11
DDHEADER.ASM; 415
DDSEGCOS.ASI; 413
DEBUG; 541
debugging; 50
default; 84
device driver; 214; 379–490
attribute word; 390; 399
build BPB (servizio 02); 393
character e block; 380; 385; 390; 394; 402
code segment; 416
command line; 450–53
device attribute word; 384
device close (servizio 14); 399
device open (servizio 13); 398
environment; 471
flush input buffers (servizio 07); 397
flush output buffers (servizio 11); 398
generic IOCTL request (servizio 19); 400
get logical device (servizio 23); 401
header; 382; 384–85; 417; 461
init (servizio 00); 389
input status (servizio 06); 396
interrupt routine; 382; 421; 446
IOCTL read (servizio 03); 393
IOCTL write (servizio 12); 398

media check (servizio 01); 391
nome logico; 384; 395; 396; 397; 398; 399
nondestructive read (servizio 05); 395
output status (servizio 10); 397
output until busy (servizio 16); 400
read (servizio 04); 394
removable media (servizio 15); 399
request header; 383; 386–402; 389; 392; 393; 395; 397; 398; 400; 401; 472
rilocazione stack; 445–50
servizi; 386–402; 428; 470
set logical device (servizio 24); 402
startup module; 461
status word; 386; 423; 470
Busy Flag; 387; 395; 396; 397; 399
Done Flag; 387
Error Flag; 387
strategy routine; 382; 386; 420
struttura; 381
tiny model; 413
toolkit per il C; 412
accesso al request header; 474–75
BZDD.H; 454
command line; 477; 479
costruzione della libreria; 453
discardDriver(); 453; 470; 474
DRVSET; 461–70
indirizzi delle parti di codice; 476; 479
init(); 426; 450; 470; 472–73
libreria di funzioni; 425
request header; 460
RequestHeaderFP; 475
setResCodeEnd(); 470; 474
setStack(); 415; 419; 445–50; 472
setupcmd(); 450–53
status word; 472; 474
uso degli stream; 480
uso di malloc(); 480
variabili globali; 476
write (servizio 08); 397
write with verify (servizio 09); 397
DGROUP; 273; 278; 414
dichiarazioni
di array; 31
di campi di bit; 63
di enum; 62
di funzione; 170
di puntatori; 18
di stringhe; 27
di struttura; 53
di template; 52; 60; 62; 63
di union; 60
di variabili; 14
portabilità; 491
DIR.H; 500; 502
DISKCOPY; 540
diversità; 74
do...while; 87
DOLIST; 645
DOS
servizi; 331
DOS List of Lists; 208
DOS.H; 116; 203; 300; 307; 347; 504
double; 16
DS; 177
DS, SS; 159
DTA; 349; 511
nei TSR; 345–47
DUP; 580
dup(); 580
dup2(); 97
durata delle variabili; 36

E

EMM386.SYS; 214
EMMXXXX0; 219
EMPTYLVL; 604
EMS; 218–44
device EMMXXXX0; 219
handle; 223
mapping; 230; 236; 241; 304
nei TSR; 304
page frame; 222
pagine; 218; 223
trasferimenti senza page frame; 238
utilizzo; 229
versione; 220
entry point; 156; 245; 541
enum; 61
environment; 116; 139; 200; 305; 348; 403; 621
nei TSR; 570
envp; 113; 509
errno; 97; 476; 533
ERRNO.H; 97
errore critico; 332; 348; 571
errori di I/O; 533–35
ERRORLEVEL; 115; 296; 603; 620; 635; 636; 648
esadecimale; 495
eseguibile; 6
struttura; 298; 310
EXE, eseguibile; 155; 298; 409
exec...(); 143

exit(); 115; 351
EXTEND.H; 191
Extended Memory Block; 251
external; 42; 165

F

f, opzione compilatore; 315
far; 23; 107; 157; 158; 159; 165; 267; 277; 326; 384; 394; 504
farmalloc(); 121; 589
    nei device driver; 471
FAT; 392; 500; 571
fclose(); 128
FCNTL.H; 135
FCREATE; 618
fgetc(); 83
fgetchar(); 125
file; 123–37; 123–37; 145
    nei TSR; 341–45
FILE; 126; 341
File Control Block; 347
FIND; 605; 637
FINDFIRST, FINDNEXT; 111; 500
FirstFit, BestFit, LastFit; 212; 590
flag DOS (InDOS, CritErr); 316
float; 16
floating point; 12
fopen(); 126; 146; 341
for; 31; 88–90; 108
FOR; 603; 645
forward reference; 302
FP_OFF(); 26; 504
FP_SEG(); 26; 504
fprintf(); 125
fread(); 129; 341
free(); 117
freemem(); 203; 303; 353
freeUMB(); 266
fscanf(); 130
fseek(); 129
funzioni; 8; 91–107; 160
    chiamata; 91
    corpo; 94
    definizione; 93
    di libreria; 163–67
        nei device driver; 404; 470
        nei TSR; 310; 341
    interrupt; 185; 272; 326
    interrupt dichiarate far; 277–80
    prototipo; 9; 93–99; 94
    valori restituiti; 94
fwrite(); 128

G

geninterrupt(); 184; 485
getch(); 92
getEMMversion(); 220
getenv(); 116; 142
getfirstmcb(); 210; 356
getInDOSaddr(); 317
gets(); 118
getvect(); 269; 544
getXMMaddress(); 246
getXMMversion(); 247
giuliano, numero; 599
goto; 85
GOTO; 603

H

handle; 134; 341; 347
    EMS; 223
heap; 112; 120; 156; 157; 159; 605
HIMEM.SYS; 214
HMA; 244; 258
huge; 24; 165; 267
huge model; 162

I

IBMBIO.COM; 204
IBMDOS.COM; 205; 320
IF; 603; 650
if...else; 81–83
include file; 8; 99; 164; 413
indirezione; 18
indirizzo; 17
    lineare; 18
indirizzo di (operatore); 19
InDOS flag; 207; 316; 348; 349; 580
inline assembly; 171; 183; 184; 185; 283; 341
    portabilità; 493
int; 16
INT; 320; 591
int 05h
    nei TSR; 321
int 08h
    nei TSR; 322
int 09h
    nei TSR; 322
int 10h; 337
    nei TSR; 325
int 13h
    nei TSR; 326
int 16h
    nei TSR; 326

int 1Bh
nei TSR; 329
int 1Ch
nei TSR; 330
int 21h
nei TSR; 331
int 23h; 357
nei TSR; 331
int 24h; 357
nei TSR; 332
int 28h
nei TSR; 333
int 2Fh; 553; 570
nei TSR; 334
int86(), int86x(); 150
intdos(), intdosx(); 152
integral; 12
interprete; 3
interrupt; 185; 269; 272; 350; 580; 588
dichiaratore di funzione; 175
gestione; 269–87; 543
nei TSR; 315–36
utilizzo; 123–53
intr(); 152
IO.H; 97; 580
IO.SYS; 205; 379
IOCTL; 385; 393; 400; 482; 487
IRET; 175; 273; 279; 322; 326; 329; 333; 353; 423; 543
isA20enabled(); 262
isDriveRemote(); 535
isHMA(); 258
istruzioni; 8

J

JMP; 281; 331; 334
JOIN; 501
jolly, funzioni; 305; 308; 403; 544; 553; 570; 587
nei device driver; 446; 471
nei TSR; 301; 303; 347; 351; 352; 590

K

k, opzione compilatore; 190; 272; 275; 280; 410; 544; 587
keep(); 296; 304; 321; 351; 570
nei device driver; 471
KISS, regola; 2; 15; 75

L

lanciare programmi da programmi; 139
large model; 159
librerie; 9; 91
costruzione; 166; 453; 533
modello di memoria; 166
LIFO; 98; 177
LIM 4.0; 218
LIMITS.H; 655
linker; 4; 298; 403; 409
opzioni per i device driver; 471
long
double; 16
int; 16
LPT1:; 124

M

macro; 48
__TURBOC__; 189
maggiore di, minore di; 74
main(); 8; 112–16; 143; 166; 199; 351; 472; 508
argomenti; 113
valori restituiti; 115
malloc(); 117; 302; 589
nei device driver; 420
max(), min(); 48
MAXPATH; 502
medium model; 157
memoria
allocazione dinamica
in Clipper; 196
allocazione dinamica; 117–21
nei device driver; 480
nei TSR; 302
convenzionale; 204–13
espansa; 218–44
estesa; 245–57
fisica; 249
ottimizzazione nei TSR; 301
segmentazione; 356
upper; 213–18; 356; 587
Memory Control Block; 205; 214; 299; 303; 306; 355; 364; 591
strategia di allocazione; 212
MK_FP(); 26; 116; 220; 267; 300; 348; 511; 591
ml, opzione assemblatore; 425; 453
modelli di memoria; 107; 121; 155–62; 180; 412; 504; 589
compact; 158
huge; 162
large; 159; 191
medium; 157
small; 157
tiny; 156; 410

MORE; 606
MOV; 179
MSDOS.SYS; 205; 320; 380
mt, opzione compilatore; 454
multiplex; 334
MYOB, regola; 31

N

N_LDIV@, N_LMOD@, N_LMUL@; 313
near; 24; 107; 158; 165; 419; 446
newint17h(); 293; 658; 659
NOP; 542
not logico; 220
Notazione Ungherese; 163
NUL; 385
NULL; 30; 41; 46; 84; 89; 118; 516
numeri casuali; 655
    funzioni di libreria; 655
    gestione del range; 657
    pseudocausalità; 655
numero giuliano; 599

O

object file; 5; 164; 454
offset; 17
oggetti; 51
open(); 341
operatori; 65–79
    aritmetici; 72
    assegnamento; 78
    autodecremento; 69
    autoincremento; 69
    condizionale; 78
    di resto di divisione intera; 73
    di separazione di espressioni; 79
    logici; 74
    precedenza e associatività; 65
    su bits; 76
or
    logico; 75
    su bit; 77

P

page frame; 222
pagine EMS; 218
parametri
    attuali; 94
    formali; 94; 273; 335
        funzioni fittizie (jolly); 188
        funzioni interrupt; 275
    in numero variabile; 97
    per valore, per riferimento; 95
parent; 139
parsemcb(); 211
PARSEOPT.H; 514
parseoptions(); 516
pascal; 98
PATH, variabile d'ambiente; 617
pathname; 495; 499; 501; 570
pathname(); 504
perror(); 534
pipe; 148
piping; 605; 617
POP; 485
portabilità; 491–97; 657; 660
    compilatori; 493
    hardware; 491
    puntatori; 494
    sistema operativo; 495
Portabilità; 146
precedenza degli operatori; 65
preprocessore; 2; 8
    direttive; 8
printf(); 16; 27; 38; 59; 60; 98; 124
PROCESS.H; 543
protezione da copia; 540
prototipo di funzione; 94; 591
pseudocasualità; 655
pseudoregistri; 182
PSP; 116; 156; 200; 298; 303; 306; 332; 333; 345; 349; 355; 364; 403; 510; 591
    nei TSR; 347–48
puntatori; 16–27; 95; 179
    a funzioni; 99–107; 591
    a funzioni interrupt; 286
    a puntatori; 35
    a struct; 55
    a union; 61
    aritmetica dei; 34
    in Clipper; 195
    nei vettori di interrupt; 588–92
    portabilità; 494
    void; 36
punto e virgola; 7
PUSH; 174; 485
PUSHF; 322
putenv(); 116; 142

Q

QEMM386.SYS; 216

R

RAM; 11

indirizzamento; 17
segmentazione; 22; 155; 204; 356
rand(); 655
random(); 657
randomize(); 656
randoml(); 658; 659
rd, opzione compilatore; 553
read(); 341
readcmos(); 593
realloc(); 117; 303
redirezione; 125; 603; 617; 629; 635
register (variabili); 38
registri; 17; 175
releaseEnv(); 306
relocation table; 155; 298; 382
RequestHeaderFP; 475
resto, operatore; 73
RET; 173; 175; 423; 544
RET n; 175; 279; 543
return; 94; 114
ricorsione; 108; 660
riga di comando; 141

S

S, opzione compilatore; 170; 173; 311; 661
scan code; 323; 327; 545; 561
scanDirectory(); 109
screen saver; 366
SEEK_SET, SEEK_CUR, SEEK_END; 129
segmento; 17
segno, estensione del; 74
SELSTR; 628
setblock(); 203; 303
setvbuf(); 133
setvect(); 269; 326
shell; 617
shift; 73
short int; 16
SI, DI; 176
side-effect; 48; 63; 494
signed; 11; 495
significatività; 23; 151; 179; 199; 247
sizeof(); 16; 58; 72; 492
small model; 157
SMARTDRV.EXE; 133
sorgente; 3
SP; 174
spawn...(); 141
spawnl(); 543; 588
sprintf(); 146
srand(); 656
SS\:SP; 309
sscanf(); 146
stack; 172–75
stack; 71; 98; 120; 152; 155; 157; 159; 273; 283; 485; 570
attivazione locale; 421
frame; 174
funzioni interrupt; 275
funzioni interrupt dichiarate far; 280
indirizzi di ritorno; 106
nei TSR; 307–10
nella ricorsione; 109
rilocazione; 308; 403
nei device driver; 419; 445–50; 479
variabili automatiche; 502
Standard Auxiliary; 124
Standard error; 635
Standard Error; 124
Standard input; 604; 629; 635
Standard Input; 124; 336
Standard output; 605; 629; 635; 649
Standard Output; 124
Standard Printer; 124
standard stack frame; 174; 190; 272; 275; 278; 353; 423; 544
startup module; 113; 300; 307; 351; 356; 409; 509
per i device driver; 412–25
STAT.H; 136
static; 165
puntatori; 27
variabili; 40
stdaux; 124
stderr; 124
stdin; 124; 604
STDIO.H; 30; 94; 124; 133
STDLIB.H; 533
stdout; 124
stdprn; 124
STI; 270
strcat(); 30
strcmp(); 30
strcpy(); 30
stream; 123; 341; 496; 629
nei device driver; 447
nomi standard; 123
STRING.H; 100
stringhe; 27–30
di formato; 16; 130
dichiarazione; 27
strlen(); 100
struct; 51–59
struct ffblk; 500
struct MCB; 210; 354

struct OPT; 516
struct REGPACK; 152; 219
struct SREGS; 151
struct VOPT; 515
SUBST; 501
switch; 83–85
switch character; 512; 517
sys_errlist; 533
SYSINIT; 379; 476
system(); 139

T

t, opzione linker; 410
tag; 51; 62
tastiera; 289; 290; 322; 326; 571
    buffer; 328; 410; 554; 562
        puntatori al; 329; 563
    nei TSR; 336
    ridefinizione dei tasti; 544
    shift status byte; 324; 328; 570
    status byte; 324
tavola dei vettori; 204; 269–71; 355
template; 51
Terminate and Stay Resident; 295–365
ternario, operatore; 78
testEMM(); 220
TIMEGONE; 620
timer; 322; 330
tiny model; 156
tipi di dato; 11
    conversione; 70
TLIB; 167; 454
    response file; 454
Tm, opzione compilatore; 302
token; 68
TSR; 141; 295–365; 379
    allocazione dinamica della memoria; 302
    disattivazione e disinstallazione; 350–57; 546; 580
    DTA; 345–47
    environment; 305
    file; 341–45
    funzioni di libreria; 310
    funzioni fittizie (jolly); 301
    gestione degli interrupt; 315–36
    I/O; 336–47
    installazione; 296
    int 2Fh; 334; 352
    memoria EMS; 304
    ottimizzazione della memoria; 301
    PSP; 347–48
    stack; 307–10
    struttura; 295
TYPE; 606
typedef; 126; 474; 571

U

uguaglianza; 74
union; 59–61; 475
union REGS; 151; 502
Unix; 2; 9; 123; 146; 149; 495; 511; 533; 635; 649
    fork(); 147
    waitpid(); 147
unsigned; 11; 495
    char; 16
    int; 16
    long int; 16
    short int; 16
Upper Memory Block; 214; 264

V

va_arg(), va_start(), va_end(); 97
variabili; 13–16
    accessibilità e durata; 36
    automatic; 37
    globali (external); 42
        nei device driver; 471
        nei TSR; 297
    register; 38
    static; 40
VDISK; 250; 267
vettori; 185; 588
    usati come puntatori; 588–92
vettori di interrupt; 269; 322; 350; 580
    utilizzo; 280
video; 325; 336–41; 491
    buffer; 564
Vienna (virus); 536
virgola mobile; 12
virgola, operatore; 79
virus; 536
void; 13; 16; 94; 276

W

while; 86; 511
WILDARGS.OBJ; 510; 521
wildcard; 109; 346; 510; 521
word; 11
writcmos(); 594

X

XMS; 218
High Memory Area; 257–64
A20 Line; 261
utilizzo; 258
memoria estesa; 245–57
utilizzo; 251
Upper Memory Block; 264–68
utilizzo; 264

NOTE: